O'REILLY®

Безопасность разработки в Agile-проектах

Лаура Белл, Майкл Брантон-Сполл,
Рич Смит, Джим Бэрд

ДМК ПРЕСС

Л. Белл, М. Брантон-Сполл, Р. Смит, Д. Бэрд

Безопасность разработки в Agile-проектах

Обеспечение безопасности в конвейере непрерывной поставки

Agile Application Security

Enabling Security in a Continuous Delivery Pipeline

Laura Bell, Michael Brunton-Spall, Rich Smith, and Jim Bird

Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo O'REILLY®

УДК 004.42
ББК 32.973
Б43

Б43 Белл Л., Бр нтон-Сполл М., Смит Р., Бэрд Д.

Безоп сность р зр ботки в Agile-проект х. / пер. с нгл. А. А. Слинкин. – М.: ДМК Пресс, 2018. – 448 с.: ил.

ISBN 978-5-97060-648-3

В большинстве орг низ ций стремительно приним ют н вооружение гибкие (agile) методики р зр ботки. Они позволяют своевременно ре гировать н изменение условий и зн чительно сниж ть стоимость р зр ботки. Одн ко исторически безоп сность и гибкие методики никогд не дружили между собой.

Эт книг поможет в м р зобр ться в том, что т кое безоп сность, к кие существуют угрозы и н к ком языке специ листы-безоп сники описыв ют, что происходит. Вы н учитесь моделиров ть угрозы, измерять степень риск , созд в ть ПО постоянно помня о безоп сности.

Изд ние будет полезно всем специ лист м, в чьи обяз нности входит обеспечение информ ционной и опер ционной безоп сности орг низ ций, р зр ботчик м, применяющие гибкие методы р зр ботки приложений, для которых Scrum и к йдзен не пустые слов .

УДК 004.42
ББК 32.973

ISBN 978-1-49193-884-3 (нгл.)
ISBN 978-5-97060-648-3 (рус.)

Оглавление

Предисловие

Программы правят миром. Разработчики стали новыми «делателями королей». Благодаря интернету вещей компьютер скоро появится в каждой лампочке.

Все это означает, что скоро программы настолько глубоко проникнут в нашу жизнь, что для большинства людей до ближайшего компьютера будет буквально рукой подать, и постоянное взаимодействие с предметами и окружением, управляемыми компьютерами, войдет у нас в привычку.

Но в таком мире нас подстерегают опасности. В старом добром мире безопасность всерьез рассматривалась только в банковских и правительственных системах. Однако повсеместное распространение компьютеров повышает потенциальную выгоду от их неправомерного использования, что, в свою очередь, создает стимулы для такого использования и, стало быть, повышает риски, с которыми сталкиваются системы.

В большинстве организаций стремительно принимают на вооружение гибкие (agile) методики разработки. Они позволяют своевременно реагировать на изменение условий и значительно снижать стоимость разработки, поэтому задают стандарт, который, как мы ожидаем, будет распространяться все шире, и в конечном итоге большая часть программ будет создаваться с помощью гибких методик.

Однако исторически безопасность и гибкие методики никогда не дружили между собой.

Специалисты по безопасности по горло заняты уже упомянутыми системами для государственных учреждений, банков и электронной коммерции – их архитектурой, тестированием и защитой, – и все это перед лицом непрерывно совершенствующихся угроз. То, что часто представляют самым интересным и захватывающим в области безопасности, то, о чем пишут в технических блогах и говорят в ночных новостях, делается командами профессиональных хакеров, специализирующихся на исследовании уязвимостей, разработке эксплойтов и ошеломительных взломах.

Наверняка вы сможете назвать несколько уязвимостей, бывших на слуху в последнее время: Heartbleed, Logjam или Shellshock (или еще каких-нибудь, с которыми на приведи бог столкнуться), или припомнить названия команд, которым удалось «разлочить» последние модели

устройств на основе iPhone и Android. Но когда в последний раз вы слыхали о новом средстве или методике защиты – с запоминающимся, интересным для СМИ названием? А знаете ли вы имя создателя этой методики?

Профессионалы в области безопасности отстают в понимании и опыте применения гибких методик, и этот разрыв несет серьезные риски для нашей отрасли.

С другой стороны, гибкие команды отбросили стесняющие оковы прошлого. Нет больше детальных технических требований, нет системного моделирования, не стало каскадной модели (модели «водопада») с ее традиционными этапами и контролем их выполнения. Беда в том, что гибкие команды вместе с водой выплеснули и ребенка. Старые методики, пусть даже иногда медленные и негибкие, доказали свою ценность. Они создавались не без причины, а отказавшись от них, гибкие команды рискуют забыть, в чем состояла эта ценность, и выбросить все достижения на свалку.

Поэтому гибкие команды редко относятся к безопасности, как должно. Некоторые гибкие практики позволяют получить чуть более безопасные системы, но зачастую это благоприятный побочный эффект, а не цель. И лишь очень немногие гибкие команды понимают угрозы, с которыми может столкнуться система, а большинство не понимает рисков, не отслеживает их и не делает ничего, чтобы управлять ими. Они даже слабо представляют, кто вообще атакует плоды их труда.

Кому стоит прочитать эту книгу?

Мы не знаем, кто вы: руководитель гибкой команды или разработчик, желающий больше узнать о безопасности. Быть может, вы – «безопасник», который только что узнал о существовании доселе неведомой ему группы разработчиков и хотел бы разобраться, чем она занята.

При написании этой книги мы имели в виду три потенциальные группы читателей.

Разработчики, применяющие гибкие методы

Вы жизни не мыслите без гибкой разработки. Scrum и кайдзен – для вас не пустые слова, разработка через тестирование вошла в плоть и кровь. Не важно, какую конкретную роль вы играете – Scrum-мастер, разработчик, тестировщик, тренер, владелец продукта или представитель заказчика, – вы владеете гибкими практиками и понимаете, в чем их ценность.

Эта книга поможет вам разобраться в том, что такое безопасность, какие существуют угрозы и на каком языке специалисты-безопасники описывают, что происходит. Мы научим вас моделировать угрозы, измерять степень риска, создавать ПО, помня о безопасности, безопасно устанавливать ПО и оценивать, какие проблемы (в плане безопасности) могут возникать в процессе его эксплуатации.

Специалист по безопасности

Если вы риск-менеджер, специалист по обеспечению целостности и безопасности информации (information assurance) или по анализу операционной безопасности, то что такое безопасность, вы понимаете. Наверное, вы с осторожностью подходите к использованию онлайновых служб, постоянно думаете об угрозах, рисках и формах их проявления, возможно, даже сами находили новые уязвимости и разрабатывали эксплойты для них.

Эта книга поможет вам понять, как работают гибкие команды и что, наконец, такое спринты и истории, о которых они все время толкуют. Вы научитесь видеть порядок в хаосе, и это поможет взаимодействовать с командой и влиять на нее. Эта книга покажет, где следует вмешаться или внести свой вклад, чтобы он оказался наиболее ценным для команды и дал наилучший эффект.

Специалист по гибким методикам обеспечения безопасности

Вы знаете все – от рисков до спринтов. Будь вы разработчиком инструментальных средств, который хочет помочь команде правильно подойти к безопасности, или консультантом, оказывающим команде платные услуги, эта книга и для вас тоже. Главное в ней – уяснить то, что авторы называют возрастанием доли правильной практики. Эта книга поможет вам узнать, кто работает с вами по соседству, а также составить представление об идеях, размышлениях и концепциях, которые, как мы видим, прорастают в организациях, работающих над этой проблематикой. Мы дадим широкий обзор предмета, чтобы вы смогли решить, чем заняться или что изучить дальше.

О структуре книги

Книгу можно читать от корки до корки, по одной главе за раз. Собственно, так мы и рекомендуем поступить; мы приложили немало трудов и

надеемся, что в каждой главе любой читатель найдет что-то полезное для себя, пусть даже это будут просто наши незатейливые шутки и забавные рассказы!

Но, честно говоря, мы полагаем, что некоторые главы окажутся для вас полезнее, чем другие.

Книга разбита на три большие части.

Часть 1. Основы

Гибкая разработка и безопасность – очень широкие дисциплины, и мы не знаем, что вам уже известно. А если вы специализируетесь в одной области, то вполне возможно, что знаний и опыта из другой вам недостает.

Если вы специалист по гибкой разработке, то рекомендуем сначала прочитать *главу 1 «Начала безопасности»*, из которой вы получите базовые сведения о безопасности.

Если раньше вы вообще не занимались гибкой разработкой или делаете первые шаги на этом пути, то, прежде чем переходить к введению в эту тему, рекомендуем прочитать *главу 2 «Элементы гибких методик»*.

В *главе 3 «Революция в методах разработки – присоединяйтесь!»* описывается история гибкой разработки программного обеспечения и различные подходы к ней. Эта глава будет особенно интересна специалистам по безопасности и людям, еще не имеющим практического опыта.

Часть 2. Гибкая разработка и безопасность

Затем мы рекомендуем всем без исключения прочитать *главу 4 «Работа с существующим гибким жизненным циклом»*.

В этой главе производится попытка соединить рассматриваемую нами практическую безопасность с реальным жизненным циклом гибкой разработки и объяснить, как они сочетаются.

В *главах 5–7* разбирается управление требованиями и уязвимостью, а также управление рисками. Это общие принципы, лежащие в основе управления продуктом и планирования разработки.

В *главах 8–13* рассматриваются различные аспекты жизненного цикла разработки безопасного программного обеспечения: оценка угроз, инспекция кода, тестирование и операционная безопасность.

Часть 3. Собираем все вместе

В *главе 14* рассматривается соответствие нормативным требованиям и их учет в средах гибкой разработки и DevOps.

Глава 15 посвящена культурным аспектам безопасности. Да, вы могли бы реализовать все описанные в книге практики, и в предшествующих главах описаны разнообразные средства, позволяющие затвердить новые подходы. Но гибкие методики – это в первую очередь о людях, и то же самое верно в отношении эффективных путей обеспечения безопасности: безопасность – это внутренне осознанное изменение культуры поведения, и в этой главе вы приведем примеры приемов, которые оказались эффективны на практике.

Чтобы изменить отношение компании к безопасности, необходимы взаимная поддержка и уважение безопасников и разработчиков. Чтобы создать безопасный продукт, требуется их тесное взаимодействие. Этого невозможно добиться только путем внедрения набора средств и практик, изменения должны пронизывать всю организацию.

Наконец, в *главе 16* рассматривается вопрос о том, что разные люди понимают под «гибкой безопасностью», и подводится итог опыту авторов: что у них получалось и что не получалось при попытке создать команды, сочетающие одновременно гибкость и безопасность.

Графические выделения

В книге применяются следующие графические выделения:

Курсив

Новые термины, URL-адреса, адреса электронной почты, имена и расширения имен файлов.

`Моноширинный шрифт`

Листинги программ, а также элементы кода в основном тексте: имена переменных и функций, базы данных, типы данных, переменные окружения, предложения и ключевые слова языка. Знак ↪ в конце строки кода означает, что код продолжается на следующей строке.

`Моноширинный полужирный шрифт`

Команды и другой текст, который пользователь должен вводить буквально.

`Моноширинный курсив`

Текст, вместо которого следует подставить значения, заданные пользователем или определяемые контекстом.

Так обозначается совет или рекомендация.

Так обозначается примечание общего характера.

Так обозначается предупреждение или предостережение.

Как с нами связаться

Вопросы и замечания по поводу этой книги отправляйте в издательство:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (в США и Канаде)
707-829-0515 (международный или местный)
707-829-0104 (факс)

Для этой книги создана веб-страница, на которой публикуются сведения о замеченных опечатках, примеры и разного рода дополнительная информация. Адрес страницы *http://bit.ly/agile-application-security*.

Замечания и вопросы технического характера следует отправлять по адресу *bookquestions@oreilly.com*.

Дополнительную информацию о наших книгах, конференциях и новостях вы можете найти на нашем сайте по адресу *http://www.oreilly.com*.

Читайте нас на Facebook: *http://facebook.com/oreilly*.

Следите за нашей лентой в Twitter: *http://twitter.com/oreillymedia*.

Смотрите нас на YouTube: *http://www.youtube.com/oreillymedia*.

Благодарности

Прежде всего мы благодарны нашим замечательным редакторам: Кортни Аллен, Вирджинии Уилсон и Нэн Барбер. Мы не смогли бы довести это дело до конца без вас и всего коллектива издательства O'Reilly.

Мы также выражаем признательность техническим рецензентам за их терпение и полезные советы: Бену Аллену (Ben Allen), Джеффу Кратцу (Geoff Kratz), Питу Макбрину (Pete McBreen), Келли Шортридж (Kelly Shortridge) и Ненаду Стояновски (Nenad Stojanovski).

И наконец, спасибо нашим друзьям и семьям, выдержавшим *еще один* безумный проект.

Глава 1

Начала безопасности

Итак, что же такое безопасность?

Обманчиво простой вопрос, на который куда как сложно ответить.

Начиная путешествие в мир безопасности, трудно не то что разобраться, но хотя бы понять, куда смотреть первым делом. В новостях об успешных *взломах* рисуют картину злоумышленника типа Нео, который располагает чуть ли не безграничным арсеналом для проведения сложнейших атак. При такой точке зрения обеспечение безопасности может показаться безнадежным делом, выходящим за рамки человеческих возможностей.

Да, действительно, безопасность – сложная, постоянно изменяющаяся дисциплина, но верно и то, что существует ряд довольно простых базовых принципов, поняв которые, будет куда легче систематизировать знания, приобретенные впоследствии. Рассматривайте обеспечение безопасности как движение вперед, а не как конечную цель – движение, начинающееся с небольшого числа фундаментальных понятий, отталкиваясь от которых, можно постепенно строить все здание.

Поэтому важно, чтобы все мы, независимо от прежнего опыта, первым делом уяснили некоторые основополагающие принципы. Мы также рассмотрим традиционные подходы к безопасности и объясним, почему они перестали быть эффективными теперь, когда гибкие методы разработки стали вездесущими.

С точки зрения команд разработчиков, под безопасностью понимается информационная безопасность (а не физическая безопасность, воплощенная в дверях и стенах, а также в организации охраны, например процедурах досмотра персонала). Информационная безопасность включает практические приемы и процедуры в начале работы над проектом, в процессе реализации системы и в ходе ее эксплуатации.

Хотя в этой книге мы будем говорить главным образом *об информационной безопасности*, для краткости будем употреблять просто слово *безопасность*. Если понадобится упомянуть о каком-то другом аспекте безопасности, например физическом, то это будет оговорено явно.

Инженеры часто обсуждают, какую технологию выбрать для систем и сред. Но вопросы безопасности вынуждают нас выходить за рамки технологий. Быть может, безопасность лучше всего представлять как область на пересечении технологии с людьми, которые эту технологию используют для повседневных надобностей (см. рис. 1.1).

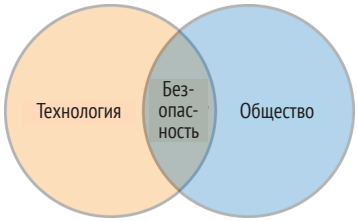

Рис. 1.1. Чем меньше общество зависит от технологии, тем меньше потребность в безопасности

Что можно сказать, глядя на эту картинку? Она показывает, что безопасность – нечто большее, чем просто технология, и, по самому своему определению, *должна* включать людей.

Людям не нужна технология, чтобы совершать дурные поступки и обманывать друг друга; этим они занимались задолго до того, как в нашу жизнь вошли компьютеры, для этого даже слово есть – *преступление*. Тысячи лет люди совершенствовались в искусстве лгать, обманывать и красть на пользу себе и своему сообществу. Но когда человек начинает взаимодействовать с технологией, это превращается в многообещающую комбинацию мотивов, целей и возможностей. В таких случаях некоторые мотивированные группы людей проводят согласованные действия в обход технологических ограничений во имя достижения конечной цели – вполне в духе человеческой природы. Именно такую деятельность и призвана предотвратить безопасность.

Следует, однако, отметить, что технический прогресс создал условия для расширения братства людей, способных на такие преступления: как благодаря доступности инструкций, так и вследствие появления всемирных сервисов, к которым может попытаться получить доступ мотивированный преступник. При наличии Интернета, мировых си-

стем связи и других достижений прогресса атаковать вас намного проще, чем раньше, притом что для преступника риск попасться гораздо ниже. Интернет и сопутствующие ему технологии сделали мир совсем маленьким, а асимметрия при этом стала даже более разительной – затраты снизились, прибыль возросла, а шансы быть пойманными резко уменьшились. В этом новом мире географическое расстояние до вожделенной богатой добычи для атакующего свелось к нулю, но действует прежняя юридическая система, основанная на межгосударственных соглашениях и процедурах межъюрисдикционных расследований и экстрадиций. И это мы еще не говорим о различиях принятых в разных странах определений существа компьютерного преступления. Технологии и Интернет также помогают взломщику избежать идентификации: вам больше не нужно явиться в банк, чтобы ограбить его, – вы можете находиться на другом конце света.

Замечание по поводу терминологии
При обсуждении небезопасности мы сознательно употребляем фразу «*в обход*», чтобы избежать неявных моральных суждений.

Чем больше в нашей жизни технологий, тем больше у нас возможностей использовать их во благо. Но у этой медали есть и оборотная сторона: чем выше зависимость общества от технологий, тем больше возможностей и стимулов использовать их во зло и тем выше доход от этого. Чем сильнее мы зависим от технологии, тем больше потребность в том, чтобы она была стабильна, защищена и всегда доступна. Если стабильность и безопасность оказываются под вопросом, страдает и бизнес, и общество. Тот же рисунок, что и выше, иллюстрирует взаимозависимость между уровнем распространения технологий в обществе и необходимостью обеспечить безопасность во имя достижения стабильности и защищенности (рис. 1.2).

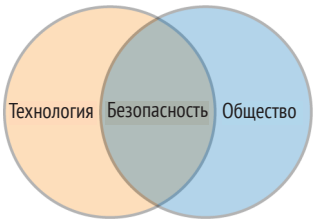

Рис. 1.2. Чем сильнее зависимость от технологии, тем выше потребность в безопасности и серьезнее последствия ее отсутствия

По мере проникновения технологии в ткань общества все большую важность приобретают подходы к осмыслению ее безопасности.

Фундаментальный недостаток классических подходов к информационной безопасности – непонимание того, что люди не менее важны, чем технология. В этой книге мы хотим предложить свежий взгляд именно на эту проблему.

Не только для технарей

Было время, когда безопасность беспокоила только государство да двинутых технарей. Но теперь, когда Интернет прочно вошел в жизнь людей по всему миру, обеспечение безопасности лежащих в его основе технологий тревожит часть общества, куда большую, чем когда-либо прежде.

Если вы пользуетесь технологиями, то безопасность имеет для вас значение, потому что любой изъян в ней может нанести прямой ущерб как вам, так и вашему окружению.

Если вы создаете технологию, то должны стремиться сделать ее стабильной и безопасной, чтобы с ее помощью мы могли улучшить бизнес и общество. Безопасность уже не является вопросом, который можно оставить где-то на периферии сознания.

- Вы несете ответственность за безопасность технологии.
- Вы заботитесь о том, чтобы люди прониклись идеей безопасности в повседневной жизни.

Если вы не берете на себя такую ответственность, значит, создаваемая вами технология фундаментально ущербна и не выполняет одну из своих главных функций.

Безопасность и риск неразделимы

Безопасность, а точнее безопасность программного обеспечения, призвана минимизировать риск. Смысл ее в том, что мы пытаемся уменьшить вероятность того, что людей, системы и данные можно будет использовать таким способом, который причинит финансовый или материальный ущерб или нанесет урон репутации организации.

Уязвимость: вероятность и последствия

Большинство методов обеспечения безопасности направлено на предотвращение атак на наши системы и информацию. Но расчет риска – это не попытка предотвратить событие, а стремление понять, что и как

может случиться, с целью расположить меры по улучшению в порядке приоритетности.

Чтобы рассчитать риск, нужно знать, что может случиться с организацией и системой, насколько вероятны такие события и какова их цена.

Это позволит определиться с тем, сколько денег и усилий потратить на защиту от потенциального ущерба.

Все мы уязвимы

Уязвимость – это подверженность риску. Вне контекста безопасности мы понимаем под уязвимостью возможность понести физический или эмоциональный ущерб. Говоря о системах и безопасности, мы употребляем это слово для описания изъянов в системе, ее компонентах или процессах, которые открывают возможность нанести вред данным, системам или людям путем злонамеренного использования или раскрытия информации.

Вероятно, вам доводилось слышать фразы типа «в такой-то программе была обнаружена новая уязвимость» или «хакеры воспользовались уязвимостью в ...». Здесь под уязвимостью понимается изъян в конструкции, конфигурации или прикладной логике программы, который позволил атакующему сделать нечто такое, что не было предусмотрено или на что он не имел права. Эксплуатация уязвимости – это акт ее использования, т. е. способ, позволяющий воспользоваться ошибкой в программе на благо атакующему.

Не невозможно, просто маловероятно

Вероятность, или *шанс* – это способ измерить, насколько легко мотивированный атакующий сможет эксплуатировать уязвимость.

Вероятность – весьма субъективный критерий, при ее оценке нужно учитывать множество факторов. При простом расчете риска ее можно свести к одному числу, но для ясности перечислим некоторые вещи, которые следует принимать во внимание при оценке вероятности.

Технические знания, необходимые для эксплуатации уязвимости

Необходимо ли быть квалифицированным техническим специалистом или достаточно случайных поверхностных знаний?

Надежность

Работает ли эксплойт надежно? Подвержены ли уязвимости различные версии, платформы и архитектуры? Чем надежнее эксплойт, тем меньше шансов, что атака вызовет заметный по-

бочный эффект, а это делает эксплойт *более безопасным* с точки зрения атакующего, т. к. снижает риск обнаружения.

Автоматизация

В какой мере эксплуатация уязвимости поддается автоматизации? Это свойство дает возможность включить эксплойт в состав комплекта эксплойтов или в самораспространяющийся код (червь), что повышает вероятность оказаться мишенью неизбирательной атаки.

Доступ

Нужно ли располагать возможностью непосредственного взаимодействия с определенной системой (сетью) или иметь определенный набор прав пользователя? Необходимо ли для успешной эксплуатации уязвимости, чтобы одна или несколько частей системы уже были скомпрометированы?

Мотивация

Является ли конечный результат эксплуатации уязвимости настолько значимым, чтобы у противника был мотив потратить свое время?

Измерение затрат

Последствия – это эффект, который эксплуатация уязвимости, недопустимое использование системы или ее взлом может оказать на вас, ваших клиентов и вашу организацию.

Для большинства коммерческих предприятий последствия измеряются в сумме утраченных денежных средств. Это может быть как прямая кража денег (например, в результате кражи кредитной карты или мошенничества), так и стоимость восстановления после взлома. Стоимость восстановления часто включает не только устранение уязвимости, но и:

- реакцию на сам инцидент;
- восстановление других систем или данных, которые были повреждены или уничтожены;
- внедрение новых подходов к повышению безопасности системы, чтобы предотвратить новые попытки взлома;
- увеличение затрат на аудит, страхование и соответствие нормативным требованиям;
- затраты на маркетинг и пиар;
- увеличение операционных издержек или применение повышенных тарифов поставщиками.

Куда серьезнее последствия для тех, кто создает системы управления или приложения, оказывающие непосредственное влияние на жизни людей. В таких случаях оценка последствий уязвимости может включать учет гибели и травм отдельных лиц или групп лиц.

В мире, который быстро движется в направлении автоматизации вождения и многих физических ролей в обществе, где компьютеризованы медицинские устройства и чуть ли не все бытовые приборы в наших домах, защита от уязвимостей все чаще подразумевает защиту людей, а не только денег и репутации.

Риск можно свести к минимуму, но не устранить вовсе

Мы склонны считать, что любое несовершенство системы можно устранить. Ошибки можно исправить, а с неэффективностью справиться благодаря более изобретательной конструкции. И действительно, большинство предметов, которые мы сами создаем и контролируем, можно довести до совершенства.

Но с риском дело обстоит иначе.

Риск связан с внешними воздействиями на системы, организации и людей. Эти воздействия чаще всего нами не контролируются (экономисты говорят о внешних факторах, или *экстерналиях*). Это могут быть отдельные лица или группы лиц с собственной мотивацией и планами, производители и поставщики со своими подходами и ограничениями или факторы окружающей среды.

Поскольку мы не контролируем риск и его причины, то никогда не сможем полностью избежать его. Бессмысленно и бесплодно стремиться к этому. Вместо этого нужно сосредоточиться на понимании рисков, на минимизации рисков (и их последствий) там, где это возможно, и постоянном мониторинге нашей предметной области на предмет появления новых и видоизменения старых рисков.

В том, чтобы принять риск, тоже нет ничего страшного, при условии что это делается осознанно, с пониманием природы риска. А вот слепо мириться с риском – путь к катастрофе, нужно постоянно быть начеку, чтобы не пропустить такую опасность, а сделать это ой как легко.

Несовершенный мир – трудные решения

Хотя наша цель – минимизировать и смягчить риски, необходимо также помнить, что мы живем в мире, где есть ограничения, а ресурсы конечны. Нравится нам это или нет, но в сутках только 24 часа, и в течение этого времени нужно еще где-то поспать. У любой организа-

ции есть бюджет и ограниченное количество людей и ресурсов, которые можно выделить для решения проблем.

Поэтому лишь очень немногие организации могут позволить себе учет всех рисков, с которыми сталкиваются. А большинство способно лишь сгладить или устранить малую часть рисков. Истратив свои ресурсы, мы можем только составить список оставшихся рисков и делать все, что в наших силах, для мониторинга ситуации и уяснения того, какими последствиями грозит оставление рисков без внимания.

Чем меньше организация, тем острее эта проблема. Но напомним, что даже самая маленькая группа с крохотным бюджетом кое-что может сделать. Малая численность и нехватка ресурсов – это не оправдание ничегонеделания, а возможность сделать максимум, чтобы обезопасить системы, творчески распорядившись имеющимися технологиями и навыками.

Решить, каким рискам уделить внимание, трудно, это отнюдь не точная наука. В этой книге мы познакомим вас с инструментами и идеями, которые позволят точнее понять и измерить риски и наилучшим образом распорядиться тем временем и ресурсами, которые имеются в наличии.

Знай своего врага

Так от кого же мы защищаемся?

Конечно, всем нам хотелось бы верить, что нас и наши приложения атакует этакий мелкий суперзлодей из комиксов, но лучше бы смотреть правде в лицо.

Существует целый ряд отдельных лиц и групп, которые могли бы или захотели бы попытаться эксплуатировать уязвимости в ваших приложениях или процессах. У каждого из них своя история, мотивы и ресурсы, а мы должны знать, как все это может сойтись, и подвергнуть нашу организацию риску.

Враг найдется у каждого

Еще недавно мы употребляли слово *кибер* для описания любого противника, который приближается к нашим владениям по внутренним сетям или через Интернет. Это породило веру в то, что есть только один вид атакующего и что это, скорее всего, злоумышленник государственного масштаба, который находится «где-то далеко».

Что такое кибер?

Слово «кибер», хотя и звучит оно так, будто пришло из романов Уильяма Гибсона, – на самом деле часть терминологии, принятой в армии США. Военные считали, что существует четыре театра военных действий, на которых допустимо ведение войны между государствами: суша, море, воздух и космическое пространство. Когда Интернет начал использоваться государствами для взаимодействия и противодействия, пришло понимание, что появился новый театр военных действий – киберпространство. Отсюда и пошло название.

Коль скоро государство стало формулировать киберстратегии и говорить о киберпреступлениях, крупные производители, естественно, подхватили жаргон. Так мы и пришли к ситуации, когда отовсюду несутся слова о различных «киберах» и связанных с ними угрозах. К сожалению, слово «кибер» стало универсальным маркетинговым термином для описания угроз и патентованных решений для борьбы с ними. Коммерциализация и неумеренное использование привели к размыванию смысла термина, превратив его в посмешище в сообществе специалистов по безопасности. В частности, люди, технически подкованные или настроенные агрессивно, зачастую употребляют слово «кибер» как издевательство.

Хотя некоторые из нас (включая нескольких авторов) стараются не употреблять слова «кибер», нельзя отрицать, что оно хорошо знакомо людям, далеким от техники и проблем безопасности. Альтернативы: «информационная безопасность», «инфобезопасность», «цифровая безопасность» – для многих куда менее понятны. Так что если слово «кибер» помогает вам в общении с людьми, плохо знакомыми с тематикой безопасности или больше ориентированными на пиар и маркетинг, то так тому и быть. Ну а разговаривая на технические темы или общаясь с людьми, более близкими к хакерскому краю спектра, имейте в виду, что это слово может обесценить ваши слова или вообще лишить их смысла.

Так вот – это не так.

Есть много типов атакующих: молодые, лишенные запретов и неугомонные люди; автоматизированные скрипты и поисковые роботы, неустанно ищущие мишени; недовольные работники; организованная преступность и политические активисты. Разнообразие атакующих куда шире и сложнее, одним словом «кибер» его не опишешь.

Мотивы, ресурсы, доступ

Размышляя о том, каких противников может заинтересовать ваша организация, принимайте во внимание как работающие в организации системы, так и людей. При этом следует учитывать три аспекта противника.

1. Цели и мотивы (почему он атакует и что надеется приобрести).
2. Ресурсы (что он может сделать, чем он может воспользоваться, чтобы это сделать, и каким временем располагает).
3. Доступ (до чего он может добраться, из каких источников получить информацию).

Пытаясь понять, от какого противника защищать организацию, насколько вероятна атака со стороны противника каждого типа и каковы возможные последствия, мы должны проанализировать все эти атрибуты в контексте организации, ее ценностей, практики работы и видов деятельности.

Мы рассмотрим эту тему гораздо подробнее, когда будем изучать создание персон безопасности и интегрировать их с процессами сбора требований и тестирования.

Цели безопасности: защита данных, систем и людей

Мы имеем право ожидать, что, занимаясь повседневными делами и взаимодействуя с технологиями и системами, не попадем в беду, пока данные остаются неповрежденными и конфиденциальными.

Безопасность – это способ обеспечить такое положение вещей, а для ее достижения мы сформулируем ряд целей.

Понимание того, что мы пытаемся защитить

Прежде всего следует решить, *что именно* мы пытаемся защитить, каковы брильянты короны в нашем мире и где они хранятся. Удивительно, сколько людей совершают какие-то действия, не понимая этого, а в результате тратят уйму времени и денег, защищая совсем не то, что нужно.

Конфиденциальность, целостность и доступность

Конфиденциальность, целостность и доступность (CIA) – сокровище среди традиционных понятий безопасности. Этот акроним служит для

описания и запоминания трех краеугольных камней безопасных систем – тех аспектов, которые мы стремимся защитить.

Конфиденциальность: держи в секрете

В наши дни мало найдется систем, которые разрешали бы всем делать всё. Мы разделяем пользователей приложения на роли и обязанности. Мы хотим быть уверены, что доступ к данным и возможность манипулировать ими были разрешены только людям, которым мы доверяем, прошедшим аутентификацию и авторизацию.

Этот аспект контроля и составляет суть конфиденциальности.

Целостность: береги от повреждений

Наши системы и приложения строятся вокруг данных. В процессе работы мы сохраняем данные, обрабатываем их и обмениваемся ими десятками разных способов.

Принимая ответственность за данные, мы предполагаем, что они будут храниться в контролируемом состоянии. Что с того момента, как нам доверили данные, мы понимаем и можем контролировать способы их модификации (кто может изменять данные, когда и каким образом). Поддержание целостности данных не означает, что их надо сохранять «законсервированными», в неизменном виде; важно, чтобы они подвергались контролируемым и предсказуемым операциям, чтобы мы понимали и могли сохранить текущее состояние данных.

Доступность: держи двери открытыми, а свет включенным

Система, к которой нельзя обратиться или использовать так, как было задумано, бесполезна. Наш бизнес и сама жизнь зависят от способности получить доступ и взаимодействовать с данными и системами почти без перерывов.

Не слишком остроумные циники скажут, что для того чтобы как следует обезопасить систему, ее нужно выключить, заключить в бетонный куб и опустить на дно океана. Но это помешает удовлетворить требование доступности.

Безопасность требует защитить данные, системы и людей, не мешая взаимодействию с ними.

Это значит, что мы должны отыскать баланс между средствами (или мерами) контроля для ограничения доступами или защиты информации и функциональностью, которая предоставляется пользователям как часть приложения. Как мы увидим, именно отыскание баланса и составляет основную проблему в нашем постоянно подключенном к сети обществе, делящемся информацией.

Неотрицаемость

Неотрицаемостью называется доказательство происхождения и целостности данных, иначе говоря, уверенность в невозможности отрицать совершенное действие. Неотрицаемость – это дополнение к аудитопригодности, и вместе они дают основания утверждать, что любое действие в системе – любое изменение, любую выполненную задачу – можно проследить до конкретного лица или до авторизованной операции.

Этот механизм связывания действия с описанием факта использования или поведением физического лица дает нам возможность поведать всю историю данных. Мы можем воссоздать и проследить все изменения и операции доступа и выстроить хронологию событий. Эта хронология поможет выявить подозрительную активность, расследовать инциденты безопасности или случаи некорректного использования и даже отлаживать функциональные дефекты в системах.

Соответствие нормативным требованиям, регулирование и стандарты безопасности

Во многих организациях одна из главных движущих сил программы обеспечения безопасности – необходимость соответствовать требованиям законодательства или отраслевым нормам. Они определяют, как должно функционировать предприятие и как следует проектировать, строить и эксплуатировать системы.

Нормативные требования можно любить или ненавидеть, но они всегда были – и будут – катализатором изменений в системе безопасности и зачастую помогают получить одобрение и поддержку со стороны руководства, без которых невозможно продвигать инициативы и внедрять изменения в сфере безопасности. Непреложные нормативные требования, «надо, и все тут» – иногда единственный способ убедить людей делать неприятные вещи, необходимые для обеспечения безопасности и конфиденциальности.

С самого начала нужно ясно понимать, что соответствие требованиям и регулирование – вещи, связанные с безопасностью, но не тождественные ей. Система может соответствовать всем требованиям и быть небезопасной, или быть безопасной, но не соответствовать требованиям. В идеальном мире все системы были бы соответствующими требованиям и безопасными, но следует отметить, что одно необязательно влечет за собой другое.

Эти концепции настолько важны, что мы посвятили им целую главу 14 «Соответствие нормативным требованиям».

Типичные заблуждения и ошибки в области безопасности

При изучении любого предмета знать антипаттерны не менее полезно, чем паттерны; понимание того, чем *не является* нечто, помогает двигаться в правильном направлении к пониманию того, чем же оно является.

Ниже приведено собрание (почти наверняка неполное) типичных заблуждений, касающихся безопасности. Стоит приглядеться, как становится ясно, что они возникают с раздражающей частотой, причем не только в технологических отраслях, но и в СМИ и даже на вашем рабочем месте.

Безопасность абсолютна

Безопасность – это не черное или белое, однако представление о том, что система является либо безопасной, либо нет, бытует повсеместно и опровергается бессчетное число раз на дню. Для любой достаточно сложной системы утверждение о ее абсолютной безопасности или небезопасности невероятно трудно, а то и вовсе невозможно доказать, поскольку все зависит от контекста.

Цель *безопасной* системы – гарантировать внедрение уровня контроля, адекватного угрозам, релевантным сценарию использования данной системы. Если сценарий изменяется, то должны измениться и средства контроля, необходимые для того, чтобы система оставалась безопасной. Аналогично, если изменяются угрозы системе, то и средства контроля должны эволюционировать соответственно.

Безопасность от кого? безопасность от чего? и *как можно было бы обойти принятые меры?* – вот вопросы, которые должны вертеться на языке при рассмотрении безопасности системы.

Безопасность – достижимое состояние

Никакая организация и никакая система никогда не будут «безопасными». Никто не повесит вам медаль на грудь, и никто не скажет, что работа сделана, система безопасна и можно идти домой. Безопасность – это культура, выбор стиля жизни, если хотите, это непрерывное стремление понять мир вокруг нас и реагировать на него. Мир постоянно изменяется, изменяется его влияние на нас, поэтому должны изменяться и мы сами.

Гораздо полезнее считать безопасность вектором, указывающим направление движения, а не точкой, в которую нужно прийти. У вектора

есть длина и направление, он говорит, куда и с какой скоростью двигаться в погоне за безопасностью. Но это дорога, по которой придется идти вечно.

Классический взгляд на безопасность можно проиллюстрировать старым анекдотом о двух охотниках, неожиданно повстречавших льва. Первый останавливается, чтобы получше завязать шнурки, второй оборачивается и кричит: «Ты спятил? Тебе же никогда не обогнать льва». Второй отвечает: «А мне и не надо обгонять льва, мне надо только обогнать тебя».

Ваши системы будут безопасны, если большинство противников решит, что получит большую выгоду, атаковав кого-нибудь другого. Как правило, организация не может повлиять на мотивы и поведение противника, поэтому ваша лучшая стратегия защиты – сделать атаку настолько трудной и дорогой, чтобы овчинка не стоила выделки.

Безопасность статична

Средства обеспечения безопасности, угрозы и подходы постоянно эволюционируют. Взгляните, как изменилась разработка программного обеспечения за последние пять лет. Подумайте, сколько появилось новых языков и библиотек, сколько проведено конференций и опубликовано статей для презентации новых идей. Безопасность в этом смысле ничем не отличается. И атакующая, и обороняющаяся стороны непрерывно модифицируют подходы и разрабатывают новые методы. Стоит атакующему обнаружить новую уязвимость и превратить ее в оружие, как защитник уже наготове и разрабатывает меры смягчения последствий и заплаты. В этой области, как и в области разработки ПО, никогда не перестаешь учиться и пробовать.

Для безопасности необходимо специальное [вставьте по своему усмотрению: пункт, устройство, бюджет]

Нет никакого недостатка в производителях и специалистах, готовых обеспечить безопасность вашей организации и систем, но правда в том, что, для того чтобы начать эту работу, ничего специального не нужно. Очень немногие из лучших специалистов по безопасности имеют какой-то сертификат или особый статус, подтверждающий сдачу некоторого экзамена; они просто посвящают своему делу все время. Обеспечение безопасности – это про отношение, культуру и подход. Не ждите, пока появится подходящее время, инструмент или учебный курс. Просто начните что-то делать.

В своем путешествии в мир безопасности вы обязательно столкнетесь с производителями, которые захотят продать вам всевозможные решения, которые позаботятся о безопасности. Да, существует немало инструментов, способных внести ощутимый вклад в общую безопасность, но не попадайтесь в ловушку – не громоздите гору лишнего. Сложность – враг безопасности, а больше почти всегда означает сложнее (даже если речь идет о большем количестве средств обеспечения безопасности). Один из авторов книги следует такому эвристическому правилу: не добавлять новое решение, если оно не позволяет вывести из эксплуатации два других. Помните об этом.

Начнем, пожалуй

Если вы заинтересовались этой книгой, то, скорее всего, вы либо разработчик, который хочет побольше узнать о безопасности, либо безопасник, желающий разобраться, что это за зверь – гибкие методики, – о котором трещат все разработчики. (Если вы не попадаете ни в одну из этих категорий, то будем считать, что у вас имеются какие-то свои, чертовски веские причины прочитать книгу о гибком обеспечении безопасности, и поставим на этом точку.)

Одна из причин, побудивших нас написать эту книгу, состоит в том, что, несмотря на очевидную необходимость глубокого взаимопонимания между разработчиками и «безопасниками», за много лет наблюдений мы такого понимания (смеем даже сказать, эмоциональной близости) почти не встречали. Более того, зачастую стороны не только не понимают друг друга, но и активно стремятся свести взаимодействие к минимуму или, хуже того, подрывают все усилия другой стороны.

Мы надеемся, что некоторые воззрения и опыт, нашедший отражение на страницах этой книги, помогут хотя бы частично устранить недопонимание и, быть может, даже недоверие между разработчиками и «безопасниками» и прольют свет на то, *что и почему делает другая сторона*.

Глава 2

Элементы гибких методик

Значительная часть этой книги призвана помочь специалисту по безопасности освоиться в мире гибкой разработки. Нам доводилось работать в организациях, которые успешно внедрили гибкие методики, но мы также работаем с компаниями, которые пока еще только пытаются овладеть гибкими методиками и DevOps.

Многие рассматриваемые в этой книге методы обеспечения безопасности применимы вне зависимости от того, ведется разработка гибко или нет, и для них не важно, насколько эффективно организация внедрила гибкие методики. Однако существуют типы поведения и приемы, следование которым позволит команде извлечь максимум пользы из внедрения гибкой разработки и рассматриваемых в книге методов обеспечения безопасности.

Все эти техники, инструменты и паттерны, способствующие успешному решению задач, типичны для высокоэффективных гибких организаций. В этой главе мы дадим обзор каждой техники и опишем, как их иерархическая организация улучшает гибкие методики разработки и поставки. В последующих главах вы найдете дополнительную информацию по этой теме.

Сборочный конвейер

Первым и, пожалуй, самым важным элементом с точки зрения разработки является *сборочный конвейер* (build pipeline). Это надежный, автоматизированный и повторяемый способ получения взаимосогласованных артефактов, допускающих развертывание.

Основное свойство сборочного конвейера состоит в том, что при любом изменении исходного кода можно инициировать процесс сборки, который надежно дает согласованные повторяемые результаты.

Некоторые компании доводят повторяемость сборки до такого уровня, что один и тот же процесс сборки, запущенный в разное время на разных машинах, дает в точности одинаковые двоичные файлы. Другие же просто выделяют одну или несколько машин специально для надежной сборки.

Это так важно, потому что вселяет в команду уверенность в целостности всех изменений кода. Мы знаем, каково работать без сборочного конвейера, когда разработчики создают выпускные сборки на своих ПК, а не включенные по забывчивости изменения, внесенные коллегой, приводят к ошибкам в регрессионных тестах.

Если вы хотите продвигаться вперед быстрее и развертывать систему чаще, то должны быть абсолютно уверены, что каждый раз корректно собирается весь проект.

Сборочный конвейер играет также роль единого места для контрольно-пропускных инспекций. Еще до пришествия гибких методик во многих компаниях контрольно-пропускные инспекции (gateway review) производились путем установки и ручного тестирования системы. При наличии сборочного конвейера эти процессы гораздо проще автоматизировать, заставив компьютеры производить проверки вместо вас.

Еще одно достоинство сборочного конвейера состоит в том, что он позволяет вернуться назад во времени, извлечь старые версии продукта и надежно собрать их. Следовательно, мы получаем возможность протестировать конкретную версию системы, в которой обнаружились известные проблемы, и применить к ней заплаты.

Автоматизация и стандартизация сборочного конвейера снижают риск и стоимость внесения изменений в систему, в т. ч. исправлений и обновлений системы безопасности. Это означает, что окно, в течение которого система остается уязвимой, можно закрыть гораздо быстрее.

Однако из того, что систему можно быстро и повторяемым образом откомпилировать и собрать, вовсе не следует, что она работает надежно. Для этого необходимо автоматизированное тестирование.

Автоматизированное тестирование

Тестирование – важная составная часть большинства процессов контроля качества ПО. Это также источник затрат, задержек и непроизводительных потерь во многих традиционных процессах.

На проектирование и написание тестовых скриптов уходит время, а еще больше времени тратится на их прогон. Во многих организациях для тестирования требуется несколько дней или недель и дополнитель-

ное время – для исправления найденных ошибок и повторного тестирования. Только после этого можно выпускать версию.

Если на тестирование уходит несколько недель, то вновь написанный код невозможно передать для тестирования раньше, чем закончатся предыдущие тесты. Это означает, что изменения накапливаются, так что новая версия становится больше и сложнее, а это требует дополнительного тестирования, и, значит, время тестирования еще увеличится… Спираль раскручивается, и становится только хуже.

Однако большая часть тестов, выполняемых в процессе типичного приемочного тестирования человеком – по контрольному списку или с помощью скриптов, – не несут большой ценности и могут (и даже должны) быть автоматизированы.

Автоматизированное тестирование обычно устроено по принципу *пирамиды тестов*, когда большинство тестов низкого уровня выполняется быстро, легко автоматизируется и может быть без труда изменено. Это снижает зависимость команды от сквозных приемочных тестов, которые нужно долго настраивать, а работают они медленно, с трудом поддаются автоматизации и с еще большим трудом – сопровождению.

В главе 11 мы увидим, что к услугам современных команд разработчиков имеется широкий ассортимент средств и методов автоматизированного тестирования: *разработка через тестирование* (TDD), *разработка на основе поведения* (BDD), интеграционное тестирование посредством виртуализации служб и полномасштабное приемочное тестирование. Каркасы автоматизированного тестирования, многие из которых поставляются с открытым исходным кодом, позволяют оформить в виде кода правила поведения системы.

Для типичной системы десятки тысяч автоматизированных автономных и функциональных тестов можно выполнить буквально за секунды, а более сложное интеграционное и приемочное тестирование занимает несколько минут.

Тесты каждого типа направлены на достижение различных уровней уверенности в правильности программы.

Автономные тесты (unit test)

В этих тестах применяется тестирование методом белого ящика для проверки того, что отдельные модули работают, как задумано. Работающая система для них не нужна, обычно тестируется, что входные данные порождают ожидаемые выходные или что имеют место ожидаемые побочные эффекты. Хорошие автоном-

ные тесты также проверяют граничные условия и возникновение ожидаемых ошибок при определенных условиях.

Функциональные тесты

Эти тесты проверяют целые группы функций. Зачастую для этого не нужна работающая система, но требуется некоторое время для подготовки, поскольку нужно связать сразу несколько модулей. Если система состоит из нескольких подсистем, то в таких тестах проверяется одна подсистема с целью убедиться, что каждая подсистема работает, как должно. Идея функциональных тестов – в том, чтобы попытаться смоделировать реальные сценарии с применением известных тестовых данных и типичных действий, которые будут выполнять пользователи.

Интеграционные тесты

Эти тесты знаменуют начало становления полной системы. Они проверяют, что все соединения сконфигурированы правильно и что подсистемы взаимодействуют, как положено. Во многих организациях интеграционное тестирование выполняется только для внутренних служб, а внешние системы заменяются *заглушками*, которые ведут себя предсказуемым образом. Таким образом удается достичь большей повторяемости результатов.

Комплексное тестирование (system testing)

Производится на полной системе, интегрированной с внешними компонентами и учетными записями. Задача этих тестов – убедиться в том, что система в целом вместе со всеми своими функциями работает, как ожидается, от начала и до конца.

Автоматизировать тесты тем сложнее, чем они ближе к концу этого перечня, но у такой организации тестирования есть свои преимущества.

Скорость

Автоматизированные тесты (особенно автономные) часто не нуждаются ни в пользовательском интерфейсе, ни в медленных сетевых вызовах. Их также можно выполнять параллельно, так что для завершения нескольких тысяч тестов нужны какие-то секунды.

Согласованность

При ручном тестировании, даже по контрольному списку, можно пропустить какой-нибудь тест или выполнить тесты неправильно или несогласованно. При автоматизированном тестировании всякий раз выполняются одни и те же действия в одном и том

же порядке. Это означает, что резко уменьшается вариативность и, следовательно, вероятность получения ложноположительных (или, что еще хуже, ложноотрицательных) результатов, возможных при ручном тестировании.

Повторяемость

В силу быстроты и согласованности автоматизированных тестов разработчик может положиться на них при внесении изменений. В некоторых вариантах гибкой разработки даже предписывается сначала писать тест, который заведомо закончится неудачно, а затем реализовать функцию так, чтобы тест прошел. Это позволяет предотвратить появление новых ошибок в уже проверенных частях программ (регрессию), а в случае разработки через тестирование – определить видимое извне поведение, что и является основной целью теста.

Аудитопригодность

Автоматизированные тесты необходимо кодировать. Этот код можно хранить в системе управления версиями, так что к нему применяются обычные механизмы управления изменениями. Это означает, что для прослеживания некоторого изменения в поведении системы нужно заглянуть в историю тестов и посмотреть, почему было внесено изменение.

В совокупности эти свойства дают высокую степень уверенности в том, что система делает именно то, что задумывали разработчики (хотя необязательно то, о чем просили или чего хотели пользователи, и именно поэтому так важно скорее передать систему в эксплуатацию и получить отзывы). Кроме того, с определенной уверенностью можно утверждать, что никакие изменения кода не окажут непредвиденное влияние на другие части системы.

Автоматизированное тестирование – не замена другим методам контроля качества, но оно значительно повышает уверенность команды в том, что внесенные изменения не «поломают» системы, и потому позволяет быстрее продвигаться вперед. К тому же при ручном тестировании и инспекциях кода можно сосредоточиться на приемочных критериях, значимость которых особенно высока.

Наконец, хорошо написанные и сопровождаемые тесты представляют собой ценную документацию того, что система должна делать.

Естественно, автоматизированное тестирование комбинируется со сборочным конвейером, так чтобы каждая полученная сборка была

полностью протестирована. Но чтобы в полной мере воспользоваться плодами обоих методов, их необходимо связать воедино, добившись *непрерывной интеграции*.

Автоматизированное тестирование безопасности

Естественно и просто предположить, что тестирование безопасности можно автоматизировать с помощью таких же методов и процессов.

Но хотя можно – и должно – автоматизировать тестирование безопасности в сборочном конвейере (и в главе 12 мы объясним, как это сделать), это отнюдь не такая простая задача, как описанное выше тестирование.

Существуют хорошие средства тестирования безопасности, которые можно встроить в процесс сборки, но большинство инструментов, относящихся к безопасности, трудно использовать эффективно, сложно автоматизировать, и работают они гораздо медленнее прочих средств тестирования.

Мы не рекомендуем начинать с одних лишь автоматизированных тестов безопасности, разве что у вас имеется опыт успешной автоматизации функциональных тестов и вы хорошо знаете, как пользоваться своими инструментами проверки безопасности.

Непрерывная интеграция

Имея сборочный конвейер, гарантирующий, что все артефакты создаются согласованно, и систему автоматизированного тестирования, реализующую базовый контроль качества, можно эти системы объединить. Обычно это называется *непрерывной интеграцией* (continuous integration – CI), но этот термин нуждается в уточнении.

Ключевое слово – «непрерывная». Идея CI-системы в том, что она постоянно следит за состоянием репозитория кода и, обнаружив изменение, автоматически запускает сборку, а затем и тестирование артефакта.

В некоторых организациях для сборки и тестирования артефакта хватает нескольких секунд, но если система велика или сборочный конвейер сложен, то это может занять несколько минут. По мере увеличения времени обычно выделяют группы тестов и шагов, которые можно запускать параллельно и тем самым быстрее получить обратную связь.

Если все тесты проходят, то на выходе процесса непрерывной интеграции получается артефакт, который можно развернуть на ваших серверах, и это делается после каждой фиксации кода разработчиком. Таким образом, разработчик почти мгновенно видит, что не совершил ошибки и не «поломал» чужую работу.

Заодно оказывается, что у команды в любой момент времени имеется работоспособный готовый к развертыванию артефакт, а значит, если возникает необходимость срочно внести исправление, в частности закрыть уязвимость, то это можно сделать легко и быстро.

Однако при выпуске артефакта нужно гарантировать, что его окружение согласовано и работает правильно. Что подводит нас к понятию *инфраструктуры как кода*.

Инфраструктура как код

Приложение или продукт можно собирать и тестировать на регулярной основе, но для системной инфраструктуры это делалось куда реже – еще недавно.

Традиционно инфраструктура системы закупалась за несколько месяцев и оставалась относительно неизменной. Но с пришествием эры облачных вычислений и *программируемого управления конфигурацией* стало возможным – и даже довольно распространенным – хранить инфраструктуру в репозитории кода.

Это можно делать разными способами, но чаще всего в репозитории определено желаемое состояние системы в виде кода. Сюда входит информация об операционных системах, имена хостов, определения сетей, правила брандмауэра, наборы установленных приложений и т. д. Этот код можно выполнить в любое время, чтобы перевести систему в желаемое состояние, а система управления конфигурацией произведет необходимые изменения в инфраструктуре.

Это означает, что внесение изменения в систему, будь то открытие правила брандмауэра или обновление версии какой-то программной части инфраструктуры, выглядит как изменение кода. Оно может быть сохранено в репозитории (который предоставляет средства управления изменениями и их отслеживания), а затем надежным и повторяемым способом воспроизведено.

Этот код имеет номер версии, он может быть подвергнут инспектированию и тестированию точно так же, как код приложения. В результате мы получаем такую степень уверенности в корректности изменений инфраструктуры, как и для изменений кода приложения.

Большинство систем управления конфигурацией регулярно сравнивают систему и инфраструктуру, и если обнаруживают различия, то могут либо выдать предупреждение, либо заблаговременно перевести систему в желаемое состояние.

При таком подходе мы можем контролировать среду выполнения, анализируя репозиторий кода, вместо того чтобы вручную просматривать и оценивать инфраструктуру. Заодно мы можем быть уверены в повторяемости среды. Как часто в вашей практике случалось, что программа работает в среде разработки, но отказывает в производственной среде, потому что кто-то вручную внес изменение в среду разработки и забыл повторить его в производственной?

Если значительная часть инфраструктурного кода является общей для среды разработки и производственной среды, то мы можем найти малейшие расхождения между средами и гарантировать, что такое никогда не случится.

Управление конфигурацией не заменяет мониторинг безопасности!

Система управления конфигурацией прекрасно справляется с задачей поддержания операционной среды в согласованном и желательном состоянии, но она не предназначена для мониторинга или оповещения об изменениях среды, связанных с действиями противника или происходящей атакой.

Средства управления конфигурацией периодически (скажем, раз в 30 минут) сравнивают фактическое состояние системы с ожидаемым. Поэтому остается временное окно, в течение которого противник может изменить конфигурацию, воспользоваться этим, а затем вернуть все, как было, – и система управления конфигурацией об этом никогда не узнает.

Механизмы мониторинга/оповещения в случае нарушения безопасности и управления конфигурацией предназначены для решения разных задач, и важно не путать их.

Конечно, можно – и высококвалифицированные команды так и делают – применить сборочный конвейер, автоматизированное тестирование и непрерывную интеграцию к самой инфраструктуре, повысив степень уверенности в том, что после изменения инфраструктура работает, как должно.

Имея согласованную и стабильную инфраструктуру, в которой производится развертывание, нужно гарантировать, что сам акт выпуска ПО повторяемый. Это подводит нас к управлению *релизами*.

Управление релизами

Типичная проблема управления проектом состоит в том, что инструкции по процессам выпуска и развертывания, призванным разместить код в производственной среде, могут составить небольшую книжку, в которой перечисляются отдельные шаги и проверки, подлежащие выполнению в строго определенном порядке.

Такие перечни шагов зачастую обновляются в последнюю очередь и потому могут содержать ошибки и упущения. Поскольку заглядывают в них редко, улучшение процесса не считается приоритетной задачей.

Чтобы процесс выпуска был менее подвержен ошибкам, гибкие команды стараются выпускать релизы чаще. Если некая процедура выполняется регулярно, то больше шансов, что она будет аккуратно сопровождаться. Кроме того, выпуск релиза – очевидный кандидат на автоматизацию, в результате чего процессы выпуска и развертывания становятся еще более согласованными, надежными и эффективными.

Включение в релиз небольших изменений часто снижает эксплуатационные риски и риски безопасности. Ниже мы объясним, что небольшие изменения проще понять, инспектировать и тестировать, а это уменьшает шансы на просачивание серьезных ошибок в производственную систему.

Эти процессы следует выполнять во всех средах, чтобы гарантировать надежную работу, а если они автоматизированы, то их можно включить в систему непрерывной интеграции. Если это сделано, то мы можем двигаться в направлении непрерывной поставки или непрерывного развертывания, когда изменение, помещенное в репозиторий кода, проходит через сборочный конвейер и стадии автоматизированного тестирования, а затем автоматически развертывается, быть может, даже в производственной среде.

Один из авторов работал в команде, которая производила развертывание изменений больше ста раз в день, а от изменения кода до его появления в производственной среде проходило меньше 30 секунд.

Однако это уже крайность, доступная опытной команде, работавшей так в течение нескольких лет. Большинство команд, с которыми мы общались, согласны на время обращения меньше 30 минут при числе развертываний от 1 до 5 в день или даже 2–3 в месяц.

Даже если вы не хотите заходить так далеко и непрерывно развертывать каждое изменение, все равно автоматизация процесса выпуска релизов позволит исключить человеческий фактор и получить повторяемость, согласованность, быстроту и аудитопригодность.

Что такое непрерывная поставка

Между непрерывной поставкой (continuous delivery) и непрерывным развертыванием (continuous deployment) есть тонкое различие.

Непрерывная поставка гарантирует, что изменения всегда готовы к развертыванию в производственной среде, – благодаря автоматизации и аудиту шагов сборки, тестирования, упаковки и развертывания, которые согласованно выполняются при каждом изменении.

В случае непрерывного развертывания изменения автоматически проходят через те же стадии сборки и тестирования, и если все хорошо, то автоматически и немедленно передаются в производственную среду. Именно так Amazon, Netflix и подобные компании достигают высокой скорости внедрения изменений.

Если вы хотите досконально разобраться во всех деталях непрерывной поставки и понять, как правильно настроить соответствующий конвейер, обратитесь к книге Dave Farley, Jezz Humble «Continuous Delivery» (издательство Addison-Wesley).

Это дает уверенность в том, что развертывание нового релиза программы не приведет к проблемам в производственной среде, поскольку и сборка, и процесс выпуска протестированы, все шаги проверены и заведомо работают.

Кроме того, благодаря встроенной аудитопригодности вы точно видите, кто решил выпустить некое изменение и что в это изменение включено. А это значит, что в случае ошибки ее будет гораздо проще найти и исправить.

Это решение гораздо надежнее и в критических ситуациях. Если необходимо срочно исправить ошибку в системе безопасности, то какой вариант вселил бы в вас большую уверенность: заплата, сделанная в обход ручного тестирования и развернутая кем-то, кто не занимался ничем подобным уже много месяцев, или заплата, собранная и протестированная так же, как все прочие программы, и развернутая скриптом, который выполняет десятки развертываний в день?

Сделав *исправление в системе безопасности* (security fix) ничем не отличающимся от любого другого изменения кода, мы много выигрываем в плане быстроты разработки и развертывания исправлений, и ключом к этому рывку вперед является автоматизация.

Но теперь, когда мы можем выпускать релизы легко и часто, надо позаботиться о том, чтобы команды не мешали друг другу. Для этого необходимо *визуальное прослеживание* (visible tracking).

Визуальное прослеживание

Когда располагаешь автоматизированным конвейером к производственной среде, становится особенно важно знать, что именно по нему движется, и предотвращать вмешательство одной команды в работу другой.

Несмотря на тестирование и автоматизацию, ошибки все же возможны, и особенно часто они вызваны зависимостями между компонентами. Может случиться, что некоторый компонент зависит от компонента, разрабатываемого другой командой. В таких случаях не исключено, что первый компонент интегрируется слишком рано и результат доходит до производственной системы раньше, чем готов компонент, от которого он зависит.

Практически в любой гибкой методике взаимодействию команд придается первоочередная важность, а самым распространенным механизмом является *визуальное прослеживание* работы. Это могут быть отрывные листочки или каталожные карточки, наклеенные на стену или на канбан-доску. Это может быть *трекер историй*. Но в любом случае, имеется ряд общих требований.

Видимость

Любой член команды и смежных команд должен сразу видеть, над чем ведется работа и что находится на пути в производственную среду.

Актуальность и полнота

Чтобы информация была полезной и надежной, она должна быть полной и актуальной. Все элементы проекта – журнал пожеланий, список дефектов, уязвимости, незавершенные работы, контрольные события и скорость работы, время цикла, риски, текущее состояние сборочного конвейера – все должно находиться в одном месте и обновляться в реальном времени.

Простота

Это не система для отслеживания всех детальных требований к каждой части работы. Каждый элемент должен кратко представлять часть работы и показывать несколько основных атрибутов, владельца и текущее состояние.

Разумеется, возможность видеть, кто над чем работает, бесполезна, если сама работа не имеет ценности. И тут мы подходим к *централизованной обратной связи*.

Централизованная обратная связь

Итак, у нас есть эффективный путь к производственной среде, и мы умеем автоматически тестировать, что изменения не «поломали» продукт. Теперь нужно как-то контролировать эффективность внесенных изменений. Чтобы оценить изменения, мы должны наблюдать за системой, а особенно за тем, как она работает.

Это не то же самое, что мониторинг системы, когда мы проверяем, работают ли машины. В данном случае мы наблюдаем за *цепочкой ценностей*: показателями, важными для команды, для пользователей системы и для предприятия, например: коэффициент обращаемости посетителей в покупателей, время пребывания на странице, коэффициент отклика на рекламу.

Причина заключается в том, что по-настоящему эффективные гибкие команды постоянно изменяют продукт, замыкая петлю обратной связи. Но для оптимизации времени цикла необходимо знать, какие данные собирать, и, в частности, оценивать, произвела ли выполненная работа полезный эффект.

Какой смысл в том, что команда выполнила 10, 100 или 1000 изменений, если невозможно связать их с какой-то работой, полезной для организации?

Индикаторы сильно зависят от контекста и службы, но вот несколько типичных примеров: эффективность использования ресурсов, доход в расчете на одну сделку, коэффициенты обращаемости, время пребывания на странице, среднее время до активации. Все эти значения можно отслеживать и отображать на информационных панелях, которые показывают исторические и текущие значения.

Знание о том, имеет ли ваша программа реальную ценность для бизнеса и отражается ли она на его показателях, помогает понять, что *хороший код – развернутый код*.

Хороший код – развернутый код

Программная инженерия и гибкая разработка бесполезны сами по себе. Они приобретают ценность, только если помогают компании достичь поставленных целей, будь то увеличение прибыли или изменение поведения пользователей.

Строка кода, не пошедшая в производство, не только абсолютно бесполезна для организации, но и является чистым долгом, потому что замедляет разработку и увеличивает сложность системы. То и другое негативно сказывается на безопасности системы в целом.

Гибкие методики помогают сократить путь до производственной системы, поскольку признают тот факт, что быстрая оборачиваемость – лучший способ извлечь ценность из написанного кода.

Конечно, все это сходится воедино при рассмотрении безопасности процесса гибкой разработки. Любой процесс в системе обеспечения безопасности, который замедляет путь до производственной системы, не принося ощутимой выгоды предприятию, является чистым долгом организации, и ориентированные на ценность команды должны искать обходные пути.

Безопасность играет важнейшую роль в определении того, что для гибкой команды означает «Сделано»: команда обязана корректно учитывать соображения безопасности. Но безопасность – это лишь один голос в хоре, который несет ответственность за то, чтобы команда осознавала риски и давала предприятию возможность принимать обоснованные решения об этих рисках.

Мы надеемся, что эта книга поможет вам понять место безопасности в этом потоке и подбросит идеи, как правильно реализовать ее в своей организации.

Работать быстро и безопасно

Что случится, если следовать всем этим рекомендациям невозможно?

Существуют организации, в которых нормативные требования запрещают вносить изменения в производственную систему без юридического оформления, а юристы вряд ли согласятся выполнять эту процедуру несколько раз в день или даже раз в неделю. В некоторых системах хранятся строго секретные данные, к которым разработчики не имеют доступа, поэтому ваши возможности принимать участие в поддержке и эксплуатации системы ограничены. А быть может, вы работаете с унаследованными корпоративными системами, которые невозможно изменить, так чтобы они поддерживали непрерывную поставку или непрерывное развертывание.

Ни один из описанных методов не является неотъемлемым от гибкой разработки, и, чтобы воспользоваться изложенными в книге идеями, необязательно слепо следовать всем рекомендациям. Но если вы не следуете большинству из них в той или иной степени, то должны понимать,

что частично утрачиваете уверенность в безопасности изменений, позволяющую работать быстро.

Можно быстро продвигаться вперед и без такой высокой степени уверенности, но тогда в краткосрочной перспективе вы принимаете на себя ненужные риски, например риск выпустить программу с критическими ошибками или уязвимостями, а в долгосрочной почти наверняка создаете технический долг и операционные риски. Кроме того, вы теряете некоторые важные преимущества от исключения человека из жизненного цикла разработки, например способность минимизировать время разрешения проблемы и длительность открытого окна уязвимости.

Важно также понимать, что, скорее всего, сразу внедрить все эти методы в командах с установившимся стилем работы не получится, не стоит и пытаться. Есть много книг, посвященных внедрению гибких методик, в которых подробно рассказано о том, как постепенно вводить культурные и организационные перемены. Но мы рекомендуем работать с командой, помогая ей свыкнуться с новыми идеями и методами, понять, как они работают и в чем их ценность, и внедрять новшества постепенно, постоянно анализируя достигнутые результаты и внося усовершенствования по мере продвижения вперед.

Описанные в этой главе методы зависят друг от друга и позволяют ускорить цикл разработки и получения обратной связи.

1. Путем стандартизации и автоматизации сборочного конвейера мы создаем фундамент для последующих технологий.
2. Автоматизация тестирования гарантирует правильности сборок.
3. Непрерывная интеграция позволяет автоматически производить сборку и тестирование после каждого изменения, благодаря чему разработчик сразу видит результат.
4. Непрерывная поставка дополняет непрерывную интеграцию упаковкой и развертыванием, что, в свою очередь, требует стандартизации и автоматизации.
5. Идея инфраструктуры как кода означает применение тех же инженерных методов и процессов к внесению изменений в конфигурацию инфраструктуры.
6. Чтобы замкнуть петлю обратной связи, необходимо определить показатели эффективности и наблюдать за всеми стадиями – от разработки до производственной системы и обратно.

По мере внедрения и совершенствования этих методов ваша команда сможет продвигаться вперед быстрее и с большей уверенностью.

Кроме того, эти методы образуют каркас управления, которым можно воспользоваться для стандартизации и автоматизации безопасности и соответствия нормативным требованиям. Этим мы и будем заниматься в оставшейся части книги.

Глава 3

Революция в методах разработки – присоединяйтесь!

Вот уже несколько лет, как компании-стартапы и команды веб-разработчиков исповедуют гибкие методы разработки программного обеспечения. Сравнительно недавно мы стали свидетелями того, как на гибкие методики переходят государственные учреждения, предприятия и даже режимные организации. Но что это такое? Как вообще можно создать безопасное ПО, даже не имея четко прописанного проекта или требований?

Возможно, вы, любезный читатель, давно и профессионально трудитесь в области безопасности, но никогда не работали с гибкой командой разработчиков. А возможно, вы инженер по безопасности, работающий с командой, практикующей гибкие методики или DevOps. А быть может, разработчик или руководитель группы, который хочет понять, как подходить к безопасности и к соответствию нормативным требованиям. В любом случае, прочитав эту главу, вы будете хорошо ориентироваться во всем, что авторы знают о гибких методиках, так что мы будем на одной волне.

Гибкая разработка: взгляд с высоты

Слово «гибкий» (agile) несет разный смысл для разных людей. Вряд ли найдутся две гибкие команды, работающие в точности одинаково. Отчасти это объясняется разнообразием гибких методик, а отчасти – тем, что любая гибкая методика поощряет адаптацию и улучшение процесса под нужды конкретной команды и контекст.

Слово «гибкий» охватывает все многообразие методик итеративной и инкрементной разработки ПО. Как термин оно появилось, когда небольшая группа авторитетных экспертов в 2001 году удалилась от мира

на лыжном курорте в Сноубэрде, штат Юта, чтобы обсудить проблемы современной разработки ПО. Крупные проекты раз за разом выходили за рамки бюджета и сроков, и даже после добавления времени и денег большинство проектов все равно не отвечало требованиям бизнеса. Ребята, собравшиеся в Сноубэрде, понимали это, и каждый из них имел за плечами успешные эксперименты по изобретению более простых, быстрых и эффективных способов поставки ПО. «Манифест гибкой разработки программного обеспечения» (Agile Manifesto, *http://agilemanifesto.org/*) – одна из немногих вещей, по которым всем 17 участникам удалось прийти к единому мнению.

Манифест гибкой разработки программного обеспечения

Мы постоянно открываем для себя более совершенные методы разработки программного обеспечения, занимаясь разработкой непосредственно и помогая в этом другим. Благодаря проделанной работе мы смогли осознать, что:

- **люди и взаимодействие** важнее процессов и инструментов;
- **работающий продукт** важнее исчерпывающей документации;
- **сотрудничество с заказчиком** важнее согласования условий контракта;
- **готовность к изменениям** важнее следования первоначальному плану.

То есть, не отрицая важности того, что справа, мы всё-таки больше ценим то, что слева.

В этом манифесте самым важным является то, что все его пункты – это просто утверждения о ценностях. Подписавшиеся вовсе не считали, что работающее программное обеспечение – это «наше всё». Но, по их мнению, при любой методике разработки в любой части процесса аспекты, указанные слева, должны иметь больший вес, чем указанные справа.

Например, согласование условий контракта – вещь важная, но только если способствует сотрудничеству с заказчиком, а не подменяет его.

За утверждениями о ценностях стоят 12 принципов (*https://www.agilealliance.org/ agile101/12-principles-behind-the-agile-manifesto/*), образующих костяк большинства гибких методик.

Из этих принципов следует, что смысл гибких методик – в поставке программного обеспечения регулярно и постепенно, что к изменению требований следует относиться как к естественному процессу, а не пытаться заморозить их раз и навсегда, и что нужно приветствовать принятие решений внутри команды.

Принципы гибкой разработки

Мы следуем таким принципам:

1. Наивысшим приоритетом для нас является удовлетворение потребностей заказчика благодаря регулярной и ранней поставке ценного программного обеспечения.
2. Изменение требований приветствуется, даже на поздних стадиях разработки. Гибкие процессы позволяют использовать изменения для обеспечения заказчику конкурентного преимущества.
3. Работающий продукт следует выпускать как можно чаще, с периодичностью от двух недель до двух месяцев.
4. На протяжении всего проекта разработчики и представители бизнеса должны ежедневно работать вместе.
5. Над проектом должны работать мотивированные профессионалы. Чтобы работа была сделана, создайте условия, обеспечьте поддержку и полностью доверьтесь им.
6. Непосредственное общение является наиболее практичным и эффективным способом обмена информацией как с самой командой, так и внутри команды.
7. Работающий продукт – основной показатель прогресса.
8. Инвесторы, разработчики и пользователи должны иметь возможность поддерживать постоянный ритм бесконечно. Гибкие методики помогают наладить такой устойчивый процесс разработки.
9. Постоянное внимание к техническому совершенству и качеству проектирования повышает гибкость проекта.
10. Простота – искусство минимизации лишней работы – крайне необходима.
11. Самые лучшие требования, архитектурные и технические решения рождаются у самоорганизующихся команд.
12. Команда должна систематически анализировать возможные способы улучшения эффективности и соответственно корректировать стиль своей работы.

Так на что же похожа гибкая разработка?

Большинство людей, которые, по их собственным словам, практикуют гибкую разработку, применяют Scrum, экстремальное программирова-

ние (XP), канбан, бережливую разработку (Lean development) – или нечто, основанное на одной или нескольких из этих наиболее известных методик. Многие команды заимствуют лучшие, на их взгляд, приемы и идеи из разных методик (чаще всего Scrum с толикой XP) и, естественно, вносят коррективы со временем. Обычно коррективы обусловлены тем, что контекст и характер разрабатываемого ПО со временем изменяются, и методика должна изменяться соответственно.

Существует ряд других гибких методов и подходов, например: SAFe, LeSS или DAD для крупных проектов, Cynefin, RUP, Crystal и DSDM. Но пристальное изучение некоторых наиболее популярных подходов поможет понять, чем различаются гибкие методики и чем все они похожи.

Scrum, самая популярная из гибких методик

На момент написания книги Scrum, безусловно, является самой популярной гибкой методикой. Существует много сертифицированных Scrum-мастеров, а учебные курсы постоянно выпускают все новых практиков и тренеров Scrum. Концептуально методика Scrum отличается простотой и может быть интегрирована во многие существующие системы управления проектами и программами. Поэтому она весьма популярна у менеджеров и директоров компаний, поскольку им кажется, что будет легко понять, чем занимается команда и когда она сдаст работу.

Scrum-проекты разрабатываются небольшими многопрофильными командами (обычно от 5 до 11 человек), которые берут задачи из общего журнала пожеланий. Как правило, в команду входят разработчики, тестировщики и дизайнеры, менеджер по продукту, или *владелец продукта*, и кто-то, играющий роль *Scrum-мастера* – лидера-слуги и тренера команды.

Спринты и журналы пожеланий

Журнал пожеланий продукта (product backlog), или журнал пожеланий Scrum, – это собрание историй, т. е. требований к продукту очень высокого уровня. Владелец продукта постоянно упорядочивает задачи из журнала пожеланий по важности и контролирует их актуальность и пригодность. Этот процесс называется «уходом за журналом пожеланий» (backlog grooming).

Этап работы Scrum-команд называется *спринтом* (sprint), традиционно спринт продолжается один месяц, хотя во многих современных Scrum-командах применяются спринты, длящиеся всего одну или две недели. Каждый спринт строго ограничен во времени: по его заверше-

нии команда прекращает работу, оценивает, что и насколько хорошо сделано, и приступает к следующему спринту.

В начале спринта вся команда, включая владельца продукта, просматривает журнал пожеланий продукта и выбирает истории с наивысшим приоритетом, над которыми будет работать.

Команду просят совместно оценить стоимость завершения каждой единицы работы, и истории помещаются в журнал пожеланий спринта в порядке приоритета. Scrum-команды могут использовать любые средства оценивания по своему выбору. Иногда применяются физические единицы времени (данная работа займет три дня), но во многих командах применяют относительные абстрактные единицы измерения (например, размеры футболок: S (малый), M (средний), L (большой) – или животных: улитка, перепёлка, кит[1]. Измерение размера в абстрактных единицах дает неформальные оценки: команда просто говорит, что одна история больше другой, а Scrum-мастер контролирует способность команды реализовать эти истории.

После того как истории помещены в журнал пожеланий спринта, Scrum-команда дает согласие завершить всю эту работу в течение спринта. Часто в таких случаях команда выбирает истории из журнала, но может взять и какие-то дополнительные истории. Так бывает, когда имеется много больших историй: команда не уверена, что успеет завершить большую историю, и вместо нее берет несколько историй поменьше.

Это соглашение между командой и владельцем продукта критически важно: владелец продукта назначает историям приоритеты, но именно команда отбирает истории в журнал пожеланий спринта.

На протяжении спринта Scrum-команда обычно относится к его журналу пожеланий, как к святыне. В течение этого времени истории никогда не помещаются в журнал спринта, но могут быть включены в более полный журнал пожеланий продукта, где им будет назначен приоритет.

Это часть контракта между Scrum-командой и владельцем продукта: владелец продукта не изменяет журнал пожеланий в ходе работы над спринтом, а команда надежно и повторяемо выполняет все запланированное в каждом скрипте. Тем самым мы жертвуем частью гибкости (отказываемся от немедленного внесения изменений в ответ на пожелания заказчика) в обмен на постоянство темпов поставки.

По возможности члены команды находятся в одном помещении, сидят рядом и могут обсуждать работу в течение дня – это способствует

[1] По-английски эти слова (snail, quail, whale) рифмуются. По-русски приблизительным эквивалентом было бы «крот, удод, бегемот». – *Прим. перев.*

сплочению команды. Если члены группы безопасности работают совместно с гибкой командой разработчиков, то важно, чтобы и сидели они вместе. Устранение барьеров для общения помогает передавать знания по безопасности и строить доверительные отношения в противовес умонастроению «мы и они».

Планерки

Команда начинает день с планерки (stand-up): короткого собрания, на котором каждый смотрит на доску или другое место, где записаны истории, планируемые в спринте, и рассказывает о планируемой на день работе. В некоторых командах для отслеживания историй используется физическая белая доска, на которой истории представлены карточками, которые перемещаются по «плавательным дорожкам» по мере изменения состояния. Другие команды применяют электронные системы, где истории представлены карточками на виртуальной стене.

В каждой команде доска устроена по-своему, но чаще всего истории перемещаются слева направо, переходя из состояния «Готова к воплощению» в состояние «В работе» и, наконец, в «Сделано». Некоторые команды добавляют дополнительные дорожки для таких задач, как дизайн или тестирование, и дополнительные состояния, в которых история стоит в очереди перед переходом в следующее состояние.

Куры и свиньи

Курица и свинья идут по дороге.
Курица говорит: «Послушай, свинья, я тут подумала, что надо бы нам открыть ресторан!»
Свинья отвечает: «Гм, возможно. А как мы его назовем?»
Курица: «Как насчет “Яичница с беконом?”»
Свинья, немного подумав: «Так не пойдет. Ведь тогда мне придётся полностью посвятить себя проекту, а ты будешь вовлечена только частично».

Все члены команды должны присутствовать на ежедневной планерке. Присутствующие делятся на «кур» и «свиней». Свиньи создают продукт, а куры только наблюдают. Курам не разрешено ни говорить, ни прерывать планерку.

В большинстве команд присутствующие выступают по очереди и отвечают на следующие вопросы:

- Что я делал вчера?
- Что я собираюсь делать сегодня?
- Что мне мешает?

Член команды, завершивший работу над историей, перемещает ее карточку в следующий столбец и зачастую удостаивается аплодисментов. Цель команды – завершать истории, как было договорено, а все, что этому мешает, называется *препятствием*.

Основная работа Scrum-мастера – устранить препятствия на пути команды. История может быть заблокирована, потому что не была готова к воплощению, но чаще всего препятствием является зависимость от какого-то стороннего ресурса, не контролируемого командой. Scrum-мастер выявляет и решает такого рода проблемы.

Циклы обратной связи в Scrum

Scrum опирается на циклы обратной связи. После каждого спринта команда собирается, проводит ретроспективную оценку спринта и решает, что может сделать лучше в ходе следующего спринта.

Такие циклы обратной связи могут стать ценным источником информации для групп безопасности. Тут можно многое узнать о проекте непосредственно от разработчиков, а также оказывать поддержку на всем протяжении процесса разработки, а не только во время контрольно-пропускных встреч или инспекций, посвященных безопасности.

Один из основных вопросов, который должна решить Scrum-команда, – является ли группа безопасности курицей (пассивным внешним наблюдателем) или свиньей (активным участником обсуждений и принятия решений). Включение эксперта по безопасности в регулярный жизненный цикл критически важно для построения доверительных отношений и для открытого, честного и продуктивного диалога по вопросам безопасности.

Ядро Scrum – взаимодействие в команде, владение продуктом, короткие итеративные циклы и циклы обратной связи – позволяет легко усвоить методику членам команд (и менеджерам), а затем и приладиться к ней. Но имейте в виду, что многие команды и организации отклоняются от чистой методики Scrum, а это значит, что вы должны понять детали конкретной интерпретации или реализации.

Кроме того, необходимо понимать, какие ограничения Scrum налагает на порядок работы. Например, Scrum запрещает или, по крайней мере, сильно ограничивает изменения в ходе спринта, чтобы команда могла всецело посвятить себя достижению целей спринта. Кроме того,

не поощряется прямое взаимодействие с членами команды разработчиков, все вопросы следует адресовать владельцу продукта.

Экстремальное программирование

Экстремальное программирование (Extreme Programming – XP) – одна из самых ранних гибких методик и, пожалуй, одна из самых гибких, но она и больше других отличается от традиционной разработки ПО.

Сейчас команды, практикующие XP, встречаются сравнительно редко, т. к. эта методика требует невероятно высокой дисциплины и сосредоточения. Но поскольку многие заимствованные у нее технические приемы активно используются другими гибкими командами, хорошо бы понимать, откуда они взялись.

Перечислим основные идеи экстремального программирования.

- У команды есть заказчик, который всегда рядом.
- Команда обязуется поставлять работающий код регулярно и небольшими релизами.
- Для создания качественного продукта команда применяет в работе определенные технические приемы, в т. ч. разработку через тестирование, парное программирование, рефакторинг и непрерывную интеграцию.
- Вся подлежащая выполнению работа возлагается на команду в целом, залогом чему являются коллективное владение кодом, единые стандарты кодирования и метафора системы.
- Коллектив работает в рассчитанном на длительный срок темпе, предотвращающем выгорание разработчиков.

Игра в планирование

Команда экстремального программирования, как и Scrum-команда, работает на основе журнала пожеланий продукта и на каждой итерации практикует игру в планирование, цель которой – выбрать истории с наивысшим приоритетом.

Команда совместно оценивает истории, добиваясь, чтобы каждый член был согласен с оценкой и готов сделать все для выполнения запланированной работы. Часто это происходит в виде коллективной игры – *покера планирования*, в которой стоимость истории оценивается по некоей абстрактной шкале, скажем, в баллах.

После того как команда придет к единому мнению о стоимости каждой истории и относительных приоритетах, начинается работа; каждый

этап продолжается обычно одну или две недели. Чаще всего команда контролирует ход работ и отчитывается о выполнении с помощью *диаграммы сгорания* (burn-down chart), на которой показано, сколько баллов осталось до завершения работы, или *обратной диаграммы сгорания* (burn-up chart), на которой показано, сколько баллов уже выполнено.

Заказчик всегда рядом

Обычно XP-команда сидит в одном помещении с представителем заказчика, который отвечает на вопросы и вправе принимать текущие решения о функциональности и пользовательском интерфейсе продукта. Как правило, команда соглашается с тем, что «карточка истории – это лишь повод для беседы» с заказчиком. Такие команды не записывают требования или истории во всех деталях, а просто все участники работы над историей демонстрируют сделанное находящемуся рядом заказчику – быть может, несколько раз в день.

От заказчика ожидают заинтересованности и даже активного изменения своего мнения об истории после ознакомления с предъявленной работой, а команда должна четко указать стоимость внесения изменений, если такое происходит.

Трудности XP-команды часто связаны с тем, что находящемуся рядом заказчику запрещено принимать решения без консультации с другими членами организации, а это резко снижает скорость команды.

Парное программирование

В отличие от Scrum, когда команда может «пересматривать и адаптировать» технические приемы так, как считает удобным для себя, XP-команды привержены строгой дисциплине разработки ПО. Наиболее известны и понятны два приема: *парное программирование* и *разработка через тестирование*.

При парном программировании каждая история выбирается двумя разработчиками, которые обычно сидят за одним компьютером и пишут код совместно. У пары одна клавиатура на двоих, и они попеременно играют роль «водителя» и «штурмана».

Водитель

«Водителем» называется тот, кто в данный момент владеет клавиатурой и вводит код.

Штурман

«Штурман» держит в голове структуру будущей программы и думает о структуре кода, контекстах приложения и прочих требованиях.

Обычно пара регулярно меняется ролями, через каждые 15–60 минут, чтобы оба привыкали к смене контекста.

Парное программирование позволяет разработчикам отделять контекст от таких несущественных деталей реализации, как синтаксис языка и детали API. Кроме того, при таком подходе на каждую строку кода смотрят две пары глаз, и в идеале один из членов пары держит в голове тестопригодность, удобство сопровождения и другие нефункциональные свойства кода, в т. ч. и безопасность. Инженеры по безопасности могут (и должны) объединяться в пары с разработчиками при работе над функциями, каркасами и прочим кодом, имеющим прямое отношение к безопасности.

Разработка через тестирование

Основная идея разработки через тестирование (test-driven development – TDD) состоит в том, чтобы сначала написать автоматизированный тест, а только потом – код, который этим тестом проверяется. Разработчики уже давно применяли автоматизированное тестирование, но только XP и TDD довели эту практику до логического предела, выступая за стопроцентную тестопригодность.

Обычно этот подход выражают тремя словами: «*красное, зеленое, рефакторинг*». Разработчик пишет набор тестов, выражающих, что должен делать код. Затем создаются методы-заглушки, необходимые, чтобы тесты компилировались. При запуске тестов должен зажечься красный свет, означающий, что сборка не проходит из-за сбойного теста.

Затем разработчик пишет код таким образом, чтобы тест прошел. После этого тест прогоняется снова, и на этот раз зажигается зеленый свет, означающий, что тест прошел. Далее разработчик анализирует написанный код, пытаясь улучшить его, устранить дублирование и упростить структуру.

Этот процесс называется *рефакторингом*, под этим понимается изменение внутреннего устройства кода без изменения его поведения. Разработчик может вносить структурные изменения в код, будучи уверенным в его работоспособности, поскольку набор тестов отловит все ошибки и несовместимости.

TDD отлично работает в сочетании с парным программированием, поскольку оно поощряет «пинг-понговый» стиль работы, когда один программист пишет тест и передает его другому, а тот должен реализовать соответствующую функциональность, написать следующий тест и «вернуть мячик» назад.

Поскольку тесты должны быть акцентированы на том, что делает код, а не как он это делает, они дают возможность обсуждать вопросы проектирования API, именования методов и инвариантов методов на этапе создания теста, и именно тесты определяют эти аспекты кода.

Метафора системы

XP включает и другие практики, например концепцию *метафоры системы*, согласно которой все члены команды должны пользоваться единым языком описания системы, поскольку это способствует коллективному владению кодом и общему пониманию задач.

Метафора системы нашла отражение уже в первом XP-проекте, где система формирования платежной ведомости была реализована как поточная линия, а различные поля платежной ведомости соответствовали рабочим местам на этой линии.

В настоящее время эта практика уступила место *предметно-ориентированному проектированию* (domain-driven design), когда от команды разработчиков требуется понимать и использовать язык предметной области, что способствует распространению знаний о ней среди членов команды.

Важное достоинство экстремального программирования состоит в том, что оно позволяет очень быстро откликаться на изменения, поскольку заказчик каждый день, а то и каждый час видит, как создается продукт и что получается. Неослабное внимание XP к технической дисциплине гарантирует высокое качество кода; методика называется *экстремальным программированием* не без причины. Многие технические приемы XP (включая непрерывную интеграцию, TDD, рефакторинг и даже парное программирование) вошли в общую практику и используются командами, которые не применяют XP, в т. ч. и такими, которые до сих пор работают по старинке.

Серьезная проблема экстремального программирования заключается в том, что сложно заранее предсказать эффективность команды и срок готовности продукта. К тому же, как оказалось, методика с большим трудом масштабируется на крупные проекты и на территориально распределенные команды.

Канбан

Канбан отличается от Scrum и XP прежде всего тем, что это не методика создания программного продукта, а метод организации работы высокоэффективных команд. Система канбан стала результатом работы

У. Эдвардса Деминга в компаниях Toyota и Toyota Production System, и суть ее заключалась в новом революционном подходе к перемещению работы по территории цеха.

Большая часть работы в цеху производится на отдельных рабочих местах, и на большинстве производств с каждым рабочим местом ассоциирована входная и выходная очередь. Работа берется из входной очереди, а результат помещается в выходную очередь.

Деминг заметил, что работа часто проходит по всей системе, т. е. заказанные детали поступают в начало конвейера, а затем продвигаются по системе. И самое главное – он обратил внимание, что частично сделанная работа проводит большую часть времени в одной из очередей на рабочем месте или в пути от одного рабочего места к другому (или еще хуже – отправляется на склад).

Взамен он предложил систему, основанную на подаче работы к каждому рабочему месту точно в срок. Это значит, что каждое рабочее место запрашивает (или *вытягивает*) работу у предыдущего рабочего места в момент, когда оказывается готово взять следующую работу.

Таким образом, количество незавершенной работы на каждом рабочем месте строго ограничено, и поток работ в системе оптимизируется. Деминга в первую очередь интересовали непроизводительные потери в системе – места, где работа была частично выполнена, но не востребована, или выполнена слишком рано.

В системах канбан назначаются приоритеты *потоку*, или *времени цикла* – показателю, измеряющему скорость прохождения единицы работы от начала до конца системы, а также количество точек контакта с ней по пути.

В ИТ-процессах количество точек контакта на пути от зарождения идеи до получения готового продукта может исчисляться десятками, а то и сотнями. У многих точек контакта есть очереди, где работа пребывает в ожидании обработки. Каждая такая очередь задерживает поток.

Типичный пример – совет по контролю изменений, который занимается анализом и принятием или отклонением предлагаемых изменений. Поскольку неэффективно созывать такой совет ежедневно, он обычно собирается на заседания раз в неделю или раз в месяц. Работа, которую необходимо сделать, должна оставаться на стадии предложения, пока совет не рассмотрит и не одобрит запрос.

Поскольку в очереди находится много запросов на изменение, совет иногда не успевает рассмотреть все и вынужден откладывать предложения до следующего заседания, что лишь увеличивает задержку. При-

нятое изменение передается в группу реализации, у которой таких изменений много, поэтому они тоже ставятся в очередь. Ну и так далее.

Системы канбан основаны на трех ключевых принципах, которые мы подробно рассмотрим в следующих разделах.

Канбан-доска: сделать работу видимой

Прежде всего используется канбан-доска, на которой каждая колонка соответствует «рабочему месту» и показывает входную очередь, работы в процессе и выходную очередь. В команде разработчиков ПО могут существовать, например, такие рабочие места: анализ, разработка, тестирование и развертывание. Это позволяет визуализировать поток работ в команде, так что одного взгляда достаточно, чтобы понять, где накапливаются работы и возникают «затыки».

Kanban строго ограничивает количество незавершенных работ на каждом рабочем месте. Если количество работ превышает возможности рабочего места, то оно не может начать новую работу, пока не закончит предыдущую.

Рабочее место с заполненной входной очередью не дает предыдущему рабочему месту переместить работу в выходную очередь. Это вызывает эффект ряби по всем предшествующим рабочим местам. И наоборот, когда последнее рабочее место завершает работу и готово принять следующую, все рабочие места передают законченные ими работы дальше.

Постоянная обратная связь

На первый взгляд, это чудовищно неэффективно, но на самом деле означает, что процесс не может продвигаться быстрее, чем самое медленное рабочее место. Понятно, что если организация пытается работать быстрее, чем это рабочее место, то она только впустую растрачивает ресурсы или создает задержки где-то в другом месте.

И это второй принцип канбана: постоянная обратная связь. Наглядно представив незавершенные работы, команда получает обратную связь о потоке и его возможностях.

Но на этом канбан не останавливается. Канбан-команды доверяют друг другу в деле формирования обратной связи. Такая полная прозрачность гарантирует, что спонсоры и заинтересованные стороны могут увидеть текущее состояние системы, понять, сколько времени потребуется новым запросам для прохождения по ней, и соответственно расставлять приоритеты запросов на новые работы.

Непрерывное улучшение

Вот мы и подошли к третьему принципу канбана: непрерывному улучшению. На каждой поточной линии, на каждом рабочем месте работников поощряют вносить рационализаторские предложения для ускорения процесса обработки.

Поскольку выявить узкие места в канбан-процессе просто, любое усовершенствование должно сразу увеличить пропускную способность системы в целом. Очевидно, что этот подход эффективнее прочих, где зачастую пытаются навести порядок на наиболее «крикливых» местах, хотя это ни к чему не приведет, если время впустую расходуется где-то еще.

Существенный побочный эффект канбана состоит в том, что в большинстве организаций процессы становятся более предсказуемыми. Процесс, в котором прохождение запроса новой функции занимало от 7 до 90 дней в зависимости от степени благоприятности условий для ее разработки, становится предсказуемым, и на реализацию новых функций уходит стандартное время.

Это означает, что заказчики функций имеют больше оснований для управления спросом. Команда разработчиков может передать задачу назначения приоритетов потребителям заказанных функций, и пусть они ведут трудные переговоры между собой.

Важно отметить, что канбан не настаивает ни на каком конкретном методе разработки. Канбан – это методика управления командой и организацией ее работы, чаще всего она применяется командами, отвечающими за несколько продуктов, когда предсказать сроки выполнения значительно труднее.

Поэтому канбан больше походит на работу групп ИТ-поддержки и эксплуатации (и групп безопасности!), чем на труд команд разработчиков, однако во многих организациях он используется на всем цикле разработки.

Канбан показывает, что уменьшение размера работы обычно повышает пропускную способность: любой процесс происходит быстрее, если нужно обрабатывать меньше, поскольку в этом случае больше работы можно проделать за меньшее время. Именно такое понимание природы вещей заставляет команды, практикующие DevOps, поставлять изменения небольшими порциями, но часто.

Бережливая разработка

Бережливая разработка и более поздняя модель бережливого стартапа (Lean startup model) Эрика Риса (*http://theleanstartup.com/*) также

основаны на системе бережливого производства, берущей истоки в канбане и компании «Toyota Production System». Однако со временем методика бережливой разработки стала развиваться немного в другом направлении.

Одно из основных отличий бережливой разработки – упор на анализ сделанного и извлечение уроков для следующих итераций.

В методике бережливой разработки на первое место ставится цикл *сборка → измерение → обучение*.

Считается, что итеративное обучение – ключ к созданию успешного продукта и что для успешного итерирования нужно не просто собирать и собирать, а потратить время и силы на измерение эффекта каждого изменения, а затем извлечь уроки из результатов измерений.

Явно выделяя части *измерение* и *обучение* на каждой итерации, мы смещаем акцент на построение поддающихся измерению систем и включаем в проект средства мониторинга и анализа.

Бережливые команды практикуют *разработку на основе гипотез*, а не описание единиц работы в терминах ценности для заказчика (по типу пользовательских историй). Они формулируют гипотезу о том, как изменение повлияет на меру ценности для бизнеса.

История считается *законченной* не тогда, когда ее код написан и развернут в производственной среде, а когда собраны и проанализированы данные, подтверждающие или опровергающие гипотезы о ценности.

Во-вторых, бережливые команды стремятся использовать экспериментальные модели, допускающие статистическое сравнение, например системы A/B-тестирования, которые позволяют предъявить заказчикам несколько реализаций для оценки.

Вот пример бережливой истории: «Размещение кнопки “Купить” на каждой странице, а не только на странице корзины побудит пользователей делать больше покупок». Эту историю можно было бы реализовать с помощью A/B-теста, показывая кнопку «Купить» на каждой странице 20% пользователей, а для остальных 80% оставив прежнее поведение. Результаты следует собрать и сравнить, чтобы понять, была ли работа проделана не зря.

Бережливая разработка также уделяет особое внимание ранней поставке минимального жизнеспособного продукта (Minimum Viable Product – MVP). Вначале команда проектирует и поставляет только минимальный набор функций, необходимый для того, чтобы начать сбор данных о том, чего пользователи хотят и за что готовы платить, а затем быстро выполняет итерации и развивает продукт на основе отзывов реальных пользователей.

Гибкие методы в целом

Вне зависимости от выбранной гибкой методики (а многие команды применяют гибридные методики, начиная с какой-нибудь хорошо известной и приспосабливая ее к своим потребностям) обнаруживается, что все гибкие команды считают ценными и практикуют следующие принципы.

Высокий приоритет обратной связи

Гибкие команды придают первостепенное значение получению обратной связи о продуктах, причем как можно раньше. Обычно это значит, что увеличивается скорость доведения ПО до состояния, допускающего демонстрацию, и устраняются коммуникационные барьеры, мешающие лицам, принимающим решение, увидеть результаты своих решений.

Гибкие команды ценят обратную связь в методике разработки, поскольку она побуждает проводить ретроспективную оценку каждой итерации или внедрять механизмы непрерывного улучшения, гарантирующие, что сам процесс адаптивен и дает желаемые результаты.

Любой механизм ускорения обратной связи – фирменное клеймо гибко разрабатываемого ПО. Это может быть непрерывная интеграция, системы быстрого развертывания или средства контроля в процессе разработки.

Сокращение пути к производственной эксплуатации – ключ к максимально быстрому получению обратной связи.

Быстрая поставка небольшими порциями

В большинстве своем гибкие методы берут начало в бережливом производстве или, по крайней мере, учитывают эту систему, а главный вынесенный урок – сокращение размера партии. Почти все гибкие команды предпочитают поставлять ПО итеративно небольшими приращениями, а не в виде одного большого массива функциональности.

В некоторых случаях поставка может производиться не сразу в производственную систему, а на предпроизводственную, или промежуточную стадию, но команда всегда может сосредоточиться на разработке небольшого числа функций за один раз и двигаться дальше в зависимости от отзывов на сданную работу.

Гибкие команды измеряют свою эффективность в терминах *скорости* – количества функций, поставленных в производственную

систему. Команда готова тратить время на автоматизацию своих систем и инструментов, если это может увеличить скорость.

Итеративная разработка

Все гибкие методики ориентированы на ту или иную форму итерации – быстрые отзывы на поставленные функции полезны только в том случае, когда реакция тоже быстрая. В большинстве гибких методик имеются средства для уменьшения эффекта переделки, а также для сильной обратной связи с лицами, принимающими решения, касательно стоимости переделки. Например, в Scrum переделка включается в журнал пожеланий продукта, так что команда должна будет оценить ее стоимость и назначить приоритет – точно так же, как для любой другой работы.

Командное владение

Гибкие методики расширяют возможности команды разработки, перенося принятие решений на уровень команды, возлагая на команду ответственность за то, как именно она будет делать свою работу, а также за ее успех или неудачу. Команда имеет право самостоятельно искать способы улучшения процессов и практических приемов – это даже ожидается от нее.

Во многих гибких командах есть тренер, опытный специалист, который помогает команде разобраться в практике применения методики, знакомит ее с разными ритуалами и задним числом анализирует итоги. Задача тренера – помочь команде самоорганизоваться и стать высокоэффективной, а не понуждать людей работать определенным образом.

Если больно, делай чаще

Гибкая команда стремится найти болевые точки в процессах разработки, а обнаружив что-то трудное или болезненное, заставляет себя делать это чаще.

Это может показаться нелепым или лишенным всякого смысла, но мы понимаем, что многие действия представляются нам сложными или неудобными только по незнанию. Классический пример – развертывание. Если команда развертывает систему раз в полгода, то обнаруживается, что процесс не стыкуется с реальностью, и никто не выполнял его целиком больше одного-двух раз в жизни. Если же развертывание производится несколько раз в день, то всегда найдутся члены команды, прекрасно знакомые с процессом и контекстом.

Процесс, выполняемый редко, обычно бывает ручным, потому что затраты на его автоматизацию и ее сопровождение высоки, по сравнению с количеством выполнений. Но если вы начнете выполнять процесс на порядок чаще, то автоматизация уберет трудности и даст колоссальный выигрыш в плане затраты усилий. К тому же автоматизация повышает повторяемость и качество, что приносит команде дополнительные выгоды.

Инспектируй и адаптируй

Гибкие методики заставляют команду итеративно совершенствовать не только разрабатываемый продукт или службу, но и саму методологию и внутренние процессы. Для этого мы должны инспектировать процессы так же, как следим за эффективностью продукта, и определять их ценность.

Для оценки и корректировки процесса команда использует такие концепции, как картирование потока создания ценности, журналы затраченного времени, скорость и ретроспективный анализ. Культура непрерывного обучения и открытости к изменению процесса позволяет команде эффективно адаптироваться к изменению контекста организации.

А как насчет DevOps?

DevSecOps – звучит знакомо

Если ваша профессия – безопасность, то многое из того, что вы услышите об элементах DevOps, может показаться удивительно знакомым. Мы понимаем, что современное состояние безопасности очень похоже на положение дел в области эксплуатации, сложившееся в 2009 году, когда зародилось понятие DevOps.

Мы являемся свидетелями резкого роста интереса к DevSecOps, DevOpsSec и прочим наименованиям технологий, объединяющих гибкие методики, эксплуатацию и безопасность. А пока суть да дело, мы полагаем, что группы безопасности могут многому научиться, познакомившись с историей DevOps и направлениями ее развития.

Движение за гибкость – конечно, здорово, при условии что в результате действительно получается работающее ПО. Но по мере того как все

больше команд разработчиков по всему миру переходят на такой способ работы, начинают возникать проблемы с эксплуатацией.

Эффективность гибкой команды измеряется главным образом объемом поставленного работоспособного ПО – «скоростью» команды. Вместе с тем группы эксплуатации оцениваются и вознаграждаются, как правило, исходя из стабильности системы, которая может измеряться временем непрерывной работы без сбоев или количеством инцидентов.

Из-за разнонаправленных приоритетов команды разработчиков и эксплуатационников могут оказаться чуть ли не на ножах.

Разработчики зачастую вынуждены принимать решения, отдавая приоритет времени и стоимости поставки в ущерб долгосрочной эксплуатационной пригодности. Если разработчики не разделяют хотя бы в какой-то мере ответственности за эксплуатацию и поддержку, то возникает отчуждение, поощряющее сиюминутное мышление и срезание углов. А трата дополнительного времени на размышления о том, как повысить удобство эксплуатации, означает, что истории поставляются медленнее и разработчики несут финансовые потери.

Движение DevOps возникло из осознания этой проблемы, произошедшего одновременно с кардинальным структурным сдвигом в автоматизации крупных операционных платформ, в частности переходом к облачным вычислениям, виртуализации и появлением программируемых автоматизированных средств, ориентированных на группы эксплуатации.

Группы эксплуатации начали осознавать, что команды гибкой разработки рассматривают их как препятствие и что во многих случаях их роль в организации должна измениться: вместо того чтобы делать что-то самостоятельно, они должны содействовать работе других.

Организации, практикующие DevOps, выделяют три направления в работе групп инфраструктуры и эксплуатации:

- группы инфраструктуры, отвечающие за закупку и управление инфраструктурой;
- группы инструментальных средств, которые создают автоматизированные средства для подготовки и управления вышеупомянутой инфраструктурой;
- группы технической поддержки, реагирующие на инциденты.

Некоторые компании передали всю инфраструктуру крупным поставщикам облачных услуг, по существу выведя группы инфраструктуры за пределы штата и возложив большую часть их работы на группы

инструментальных средств, которые предоставляют разработчикам средства для самостоятельной подготовки, эксплуатации и сопровождения инфраструктуры.

Такие вещи, как настройка протоколирования, мониторинга, оповещений, применение исправлений и т. п., в большинстве организаций являются задачами, решаемыми однократно (для каждого поставщика инфраструктуры). А после этого разработчикам можно предоставить необходимые API и инструменты – и пусть они все делают сами.

Поддержка – куда более серьезная проблема, но некоторые организации быстренько решили, что разработчики должны сами поддерживать свои приложения. Такой выход из изоляции заставил команды разработчиков думать о том, с какими трудностями может быть сопряжена эксплуатация их продуктов, и устранил отчуждение результатов их труда. В результате службы стали более надежными.

Передача команде разработчиков ответственности за сопровождение и эксплуатацию созданных служб требует уровня организационной зрелости, которым могут похвастаться не все организации. Но организации, преуспевшие в этом, могут достигать невероятной скорости и эффективности.

Многие команды, практикующие DevOps, выпускают производственные релизы несколько раз в день, а в некоторых организациях удается достичь сотен и даже тысяч развертываний в день.

Вспоминая бережливую разработку и канбан, мы можем сказать, что если самая большая очередь в системе – ожидание обратной связи на развернутую систему, то устранение этого узкого места может принести бизнесу огромную выгоду.

Улучшение обратной связи и ускорение вывода на рынок – не единственные достоинства этого подхода. Мы также видим, что организации, способные выпускать релизы чаще, оказываются значительно надежнее.

Организации, которые производят развертывание редко, обычно обращают больше внимания на *среднюю наработку на отказ* (mean time between failures – MTBF). Такая стратегия позволяет избежать риска, но одновременно означает, что организация плохо подготовлена к устранению последствий отказа, когда он все же случается. Поэтому мы постепенно переходим на другой показатель – *среднее время восстановления* (mean time to recovery – MTTR) после эксплуатационного отказа. Организация, которая включает в каждый релиз небольшой объем изменений, способна установить причину проблемы гораздо быстрее и сразу выпустить исправление, что уменьшает MTTR.

Но экстремальная скорость изменений в DevOps создает серьезные проблемы с точки зрения безопасности и требует переосмысления наших подходов к безопасности. Мы будем говорить об этом на протяжении всей книги, а особенно в главах, посвященных эксплуатации, управлению рисками, соответствию нормативным требованиям и тестированию.

Гибкие методики и безопасность

Заставить гибкие команды успешно сотрудничать с безопасниками исторически было трудным делом. Отчасти проблема состоит в том, что процессы и практики в области безопасности создавались для крупных проектов, разрабатываемых по каскадной технологии с предварительной фиксацией требований, а не маленькими командами, работающими быстро и итеративно.

Многие профессионалы в области безопасности испытывали большие трудности, пытаясь адаптировать существующие методы к миру, где требования могут изменяться еженедельно, да к тому же никогда не формулируются в письменном виде. К миру, в котором решения, относящиеся к проектированию и управлению рисками, принимаются командой в последний момент, а не планируются и не спускаются сверху вниз. К миру, где ручное тестирование и контроль соответствия не поспевают за скоростью поставки.

Хуже того, слишком многие группы безопасности считают своей целью по возможности запрещать любые изменения, чтобы свести к минимуму изменения в профиле риска приложения или среды: если риск не изменился, то к группе безопасности не может быть никаких претензий за вновь возникшие угрозы.

Если группа безопасности стремится уменьшить риск ценой отказа от изменений, вместо того чтобы помогать группе разработки в реализации ее идей безопасным образом, то в гибком мире она обречена на забвение, просто все работы будут проводиться в обход ее. А в результате системы станут менее безопасными, менее защищенными, не соответствующими нормативным требованиям, поскольку забота о безопасности исключена из процесса разработки.

Несмотря на все это, есть команды, которые успешно применяют гибкие методики и при этом поставляют безопасное ПО, и далее в этой книге мы познакомимся с приемами и инструментами, которые хорошо работают в гибких командах, а также с практическими методами, способными улучшить процесс в целом.

Чтобы все это действовало, каждый должен выполнять свою часть работы.

Гибкие команды должны понимать и принимать меры обеспечения безопасности и брать на себя больше ответственности за безопасность создаваемых систем.

Владельцы продуктов должны давать своим командам больше времени на надлежащую проработку вопросов безопасности, они должны понимать требования к безопасности и соответствию нормативным требованиям и назначать им соответствующие приоритеты.

Безопасники должны научиться принимать изменения, работать быстрее и более итеративно, учиться думать о рисках безопасности и управлять этими рисками инкрементно. И самое главное – безопасность должна быть подспорьем, а не препятствием.

Гибкие методики и DevOps – не преходящее увлечение. В будущем ИТ-отделы должны работать быстрее, реагировать оперативнее, сотрудничать успешнее и повышать уровень автоматизации.

Система безопасности должна смотреть в лицо этим вызовам и позаботиться о том, чтобы будущее было не только быстрым, но и безопасным. В следующей главе мы начнем изучать, как и где можно включить безопасность в жизненный цикл гибкой разработки.

Глава 4

Работа с существующим жизненным циклом гибкой разработки

Итак, вы полны желания создавать более безопасное программное обеспечение, но в контрольных списках по безопасности и соответствию нормативным требованиям значится, что необходимы инспекция проекта и тестирование на проникновение, а вы не понимаете, в какое место жизненного цикла гибкой разработки их воткнуть.

Традиционные модели безопасности приложения

В традиционной модели безопасности приложения безопасность включается в цикл разработки ПО в основном в виде контрольно-пропускных пунктов, где продукт должен остановиться и быть пропущенным. Кое-какая работа по обеспечению безопасности производится параллельно с разработкой, и КПП – это возможность проверить, что безопасность и разработка не разошлись в стороны. Ниже перечислены типичные КПП.

Инспекция проекта или требований

Группа безопасности изучает список требований или первые варианты проекта и добавляет требования к безопасности, основанные на модели угроз и дереве атак.

Инспекция архитектуры

Группа безопасности изучает предложенную архитектуру, например инфраструктуру или информационные потоки, и предлагает

ряд средств контроля безопасности с целью свести риск к минимуму.

Инспекция кода

Группа безопасности изучает чувствительные участки кода и подтверждает, что требования к безопасности удовлетворены и что код соответствует архитектуре.

Тестирование безопасности

Группа безопасности или внешние эксперты проверяют тестовую версию продукта на соответствие требованиям безопасности с целью убедиться в безопасности и защищенности системы.

Идея этих КПП состоит в том, что результаты работы поставляются крупными пакетами. Основано это на старом правиле, гласящем, что чем раньше обнаружен дефект, тем дешевле его исправить; поэтому инспекцию безопасности нужно производить как можно раньше, чтобы выявить все дефекты, прежде чем дело зайдет слишком далеко.

Практики гибкой разработки возражают: да, это правило, вообще говоря, верно – позднее выявление дефекта *действительно* обходится дороже раннего, – но решение заключается не в том, чтобы пытаться сделать невозможное – отловить все дефекты на ранней стадии, – а в том, чтобы уменьшить стоимость исправления дефектов, сделав внесение изменений простым и безопасным делом.

То же самое справедливо в отношении функций и средств контроля безопасности. Мы хотим найти тонкий баланс между заблаговременным выявлением и исправлением (а лучше предотвращением) проблем с безопасностью в тех случаях, когда это имеет смысл, и уверенностью в том, что сможем исправить их быстро и дешево позднее, если что-то все-таки останется незамеченным.

Некоторые дефекты безопасности попадают в особую категорию дефектов: ошибки проектирования, приводящие к критическим неисправностям или фундаментальным проблемам в функционировании системы. В то время как многие ошибки сравнительно легко исправить позже и они не несут существенного риска системе, для исправления фундаментального изъяна в проектировании может потребоваться начать с самого начала, иначе придется исправлять ошибки безопасности, следующие одна за другой бесконечным потоком.

Например, выбор неподходящего языка или каркаса, а также слишком большое доверие к механизмам платформы PaaS или инфраструктуре в надежде, что они позаботятся обо всех проблемах, могут

стать причиной серьезных рисков для безопасности, а также фундаментальных проблем с надежностью и масштабируемостью на этапе выполнения.

В чем разница между дефектом и изъяном?

В разговорной речи слова «дефект» (bug) и «изъян» (flaw) используются как синонимы, но в области безопасности они описывают совершенно разные типы проблем.

Дефект – это низкоуровневая ошибка реализации, которая приводит к тому, что система работает не так, как задумано, что влечет за собой угрозу безопасности. Типичный пример дефекта – отсутствие проверки входных данных, поступивших от пользователя и используемых при обращении к базе данных. Это может привести к атаке путем внедрения SQL.

Знаменитый пример дефекта безопасности – уязвимость Heartbleed в OpenSSL (CVE-2014-0160) из-за того, что сервер не проверял совпадение объявленной длины данных в сообщении «я жив» с истинной. Благодаря этой ошибке противник мог заставить уязвимый сервер отправить ему лишние данные, которые могли содержать секреты.

Изъян – это ошибка проектирования. В этом случае система работает, как ожидалось, и именно поэтому оказывается уязвимой. Изъяны часто возникают в системах, которые проектировались без учета безопасности. Канонический пример изъяна – клиент-серверная система, которая производит аутентификацию пользователя на стороне клиента и отправляет серверу результат аутентификации. Это делает систему уязвимой к атакам с посредником.

Знаменитый пример изъяна безопасности – проблема Dirty Cow (CVE-2016-5195), затронувшая ядро GNU/Linux. При проектировании системы копирования при записи (COW) и управления памятью было допущено состояние гонки. При его возникновении непривилегированный пользователь мог записать в файл, принадлежащий пользователю root, а это позволяло повысить привилегии до уровня root.

Почему эти различия так важны? В общем случае исправить дефект проще и дешевле, чем изъян, поскольку это сводится всего лишь к исправлению конкретной ошибки разработчика. Тогда как для устранения изъяна может потребоваться существенная реструктуризация как кода, содержащего ошибку, так и всего зависящего от него кода. Реструктуризация уже реализованной системы может оказаться очень сложным делом, чреватым внесением новых дефектов и изъянов.

> Бывают случаи, когда изъян вообще невозможно устранить, и система остается уязвимой на протяжении всего времени жизни. Наилучший способ предотвратить изъяны – учитывать вопросы безопасности на этапе проектирования приложения. Чем раньше архитектура безопасности включена в жизненный цикл разработки, тем меньше вероятность просачивания изъянов в проект.

Поэтому даже в гибкой среде безопасность следует учитывать на ранних стадиях обдумывания продукта и обсуждения архитектуры, а не только на более поздних этапах жизненного цикла разработки.

Давайте возвратимся от кодирования и тестирования (которые в гибких методиках производятся одновременно) к планированию и проектированию и обсудим, какие действия в плане безопасности должны происходить на каждом этапе.

Ритуалы на каждой итерации

На протяжении одной итерации разработки продукта имеется ряд ритуалов, в которые вовлечена безопасность.

На ежедневной планерке, когда производится инспекция состояния историй, команда должна выслушивать доклад о возникших проблемах, которые могут затронуть безопасность и конфиденциальность. Если имеются истории, касающиеся важности безопасности, то следует особо контролировать ход работ по ним.

В процессе разработки было бы желательно, чтобы к паре, которая пишет код, требующий соблюдения безопасности, был прикреплен человек, хорошо знакомый с этой проблематикой. Особенно это относится к командам, постоянно практикующим парное программирование.

Если команда практикует коллективную инспекцию кода (а так и должно быть) или пользуется платформой для коллективной инспекции, то участие в этой процедуре специалиста по безопасности может помочь в выявлении частей кода, требующих повышенного внимания и дополнительного тестирования.

В начале каждой итерации на установочном совещании, посвященном планированию, многие команды совместно отбирают истории для этой итерации. Должен присутствовать специалист по безопасности, который проследит за тем, чтобы все понимали требования безопасности, применимые к каждой истории.

Кто воплощает безопасность?

Кого конкретно мы имеем в виду в организации, практикующей гибкие методики, когда произносим слово «безопасность»?

Ответ зависит от размера компании и от приоритетов команды. В стартапе или небольшой компании отдельного специалиста по безопасности может и не быть. Эта роль возлагается на какого-то члена команды, которого время от времени консультирует и контролирует внешний эксперт (см., однако, главу 14, где описаны проблемы и риски такой модели).

В более крупной организации может существовать специалист или группа, занятые только обеспечением безопасности компании. Но, как правило, группа безопасности отвечает за физическую безопасность, сетевую безопасность, соответствие нормативным требованиям и аудит, а не только за консультирование по вопросам безопасности приложений и их поддержку.

В некоторых организациях, где к безопасности и к соответствию нормативным требованиям относятся серьезно, к каждой команде гибкой разработки прикреплен отдельный человек, курирующий вопросы безопасности. Возможно, он уделяет этой работе не все время, а, скажем, 20%, а возможно, прикреплен сразу к нескольким командам. Бывает и так, что в команде имеется свой специалист по безопасности с полной занятостью.

Главное – что кто-то в команде должен принимать на себя роль «безопасника» на каждой итерации и следить за тем, чтобы на этапах сбора требований, проектирования, кодирования, тестирования и внедрения были учтены вопросы безопасности и риски, а система оценивалась с точки зрения возможного противника. Тот, кто несет эту ответственность, а равно и все прочие члены команды должны рассматривать безопасность как неотъемлемое звено, понимать и соглашаться с тем, что учет безопасности с самого начала и на всех этапах процесса способствует получению более успешного конечного продукта.

Иногда и в конце итерации следует привлекать специалиста по безопасности к инспекциям и ретроспективному анализу, чтобы он понимал, что команда сделала и с какими проблемами столкнулась.

В каждой из этих точек у специалиста по безопасности есть возможность пообщаться с разработчиками, помочь друг другу, поучиться друг у друга и завязать полезные личные отношения.

Устранять барьеры для взаимодействия с группой безопасности – ключ к тому, чтобы безопасность не мешала поставке. Группа безопасности должна быстро предоставлять неформальные консультации, отвечать на вопросы с помощью мессенджера или чата, по электронной почте, а по возможности и лично, так чтобы никто не рассматривал безопасность как препятствие. Если до лиц, отвечающих за безопасность, невозможно достучаться, то почти наверняка соображения безопасности будут отложены в сторону, поскольку разработка не должна тормозиться.

Инструменты, встроенные в жизненный цикл

Наилучший способ удостовериться в безопасности создаваемой системы – подвергнуть ее полному набору проверок до передачи в эксплуатацию.

Технологические достижения в области безопасности привели к созданию ряда инструментов такого рода:

- Gauntlt
- BDD-Security
- Snyk
- InSpec
- Brakeman
- ZAP
- OSQuery
- TruffleHog
- Dependency-Check
- Error-Prone

Эти и другие инструменты, которые мы будем рассматривать далее, автоматизируют многие процессы контроля качества, за которые традиционно отвечали ручные тестировщики. Они не полностью устраняют необходимость ручного тестирования, но помогают планировать уделяемые ему ресурсы, беря на себя рутинную, отнимающую много времени работу и повышая степень повторяемости и надежности тестов.

Владельцем этих инструментов должна быть группа безопасности, а команда разработчиков владеет их внедрением в конвейер.

Это значит, что разработчики отвечают за включение инструмента в свой конвейер, за его правильную настройку и за действия на основе полученных результатов.

Группа безопасности определяет, какими функциями должен обладать инструмент, отвечает за простоту его включения в конвейер и за покрытие инструментом тех проблем, которые вызывают наибольшее беспокойство у команды.

Деятельность до начала итераций

В большинстве гибких команд имеется не только группа разработчиков, непосредственно участвующая в итерациях, но и один или несколько проектировщиков, которые подключаются раньше разработчиков и занимаются проблемами проектирования, созданием прототипов и обсуждением архитектуры. Результаты их работы поступают непосредственно в журнал пожеланий продукта, так что у команды есть начальный набор историй, готовых для предстоящей итерации.

Нам встречалось несколько способов организации этой работы: от отдельной команды проектировщиков, опережающих разработчиков на одну итерацию, до ежемесячных совещаний по проектированию продукта, на которых согласуется пакет задач на несколько спринтов вперед.

Безопасность имеет первостепенное значение на этапе проектирования и выбора архитектуры. Именно на этой стадии следует думать не о версии исправлений библиотеки и не о требованиях к безопасному кодированию, а о безопасном дизайне службы, моделировании доверия и о безопасных архитектурных паттернах.

Важно отметить, что под дизайном в этом случае понимается не внешний вид системы – до виртуозов Photoshop'а нам сейчас дела нет. Мы имеем в виду проектирование внутреннего устройства системы, основных взаимодействий с пользователем, API и потока данных в системе.

Группа проектирования должна иметь доступ к учебным курсам по безопасности или к экспертам-безопасникам, которые помогут удостовериться в том, что проектируемая служба отвечает требованиям безопасности на уровне взаимодействий с пользователем. Приведем несколько примеров этой работы: нужно ли запутывать сведения о пользователе при отображении, и если да, то как; как собирать изменения данных; какие требования к идентификации предъявляются при совершении различных действий.

Архитектурная группа должна также иметь доступ к архитектору безопасности на этапе обсуждения сложной архитектуры. Разработка изначально безопасной архитектуры сильно отличается от написания безопасного кода и от контроля отсутствия дефектов в продукте.

Архитекторы должны тщательно обдумать модели угроз (или *счет угроз* для команд, не практикующих формальное моделирование) и границы доверия в системе (мы будем говорить об этом в главе 8).

Инструменты планирования и обнаружения

Группа безопасности должна предоставить инструментальные средства, процессы и консультации, которые помогли бы менеджерам, архитекторам и разработчикам соблюдать общепринятые правила безопасности при проектировании новой системы.

Это может быть просто вики-сайт с перечнем стандартных правил безопасности, уже используемых в организации, или средства моделирования угроз, или контрольные списки либо анкеты для оценки технического риска – все, что помогает архитекторам разобраться в проблемах безопасности и с самого начала понять, как их решать.

Деятельность после итерации

Гибкие команды, воспринявшие культуру DevOps, нуждаются в быстрых автоматизированных системах для надежной и повторяемой сборки продукта и его развертывания в производственной среде.

Есть несколько причин, по которым безопасность имеет значение в процессе сборки и развертывания.

1. Уверенность в том, что собрано и развернуто именно то, что нужно.
2. Уверенность в том, что собранная и развернутая система безопасна.
3. Уверенность в том, что всякий раз система собирается и развертывается безопасным образом.

Контроль безопасности на этой стадии должен быть автоматизируемым, надежным, повторяемым и понятным команде – только тогда она примет его на вооружение.

Ручные процессы – полная противоположность этим требованиям: по большей части они ненадежны (т. е. не обеспечивают уверенное обнаружение одних и тех же ошибок), неповторяемы (т. е. не гарантируют, что некая ошибка каждый раз будет обнаруживаться) и непонятны команде.

В идеале группа безопасности уже давно вовлечена в процессы эксплуатации: она принимает участие в составлении планов по обеспечению непрерывности бизнеса, планов реагирования на инциденты, а также отвечает за мониторинг и аудит подозрительной активности в системах.

Но используется ли ее время эффективно? Группа безопасности должна знать, какие функции включены в последний релиз, и позаботиться об их отражении в средствах протоколирования, обнаружения мошенничества и в других системах безопасности.

Понятно, что за функциями, несущими высокий риск, следует наблюдать более пристально. Можно, в принципе, временно смириться с риском на протяжении итерации в надежде, что вероятность его реализации до включения надлежащих средств контроля спустя несколько итераций очень мала.

Но группа безопасности должна знать об этих рисках и следить за ними, пока они не будут смягчены. Это означает, что на ранних стадиях разработки системы должны быть реализованы эффективные механизмы протоколирования и аудита, гарантирующие, что такого рода вещи не пройдут незамеченными.

Мы вернемся к этому вопросу в главе 12, когда будем говорить о безопасности эксплуатации.

Инструментальные средства в помощь команде

Помимо предоставления средств автоматизированного тестирования безопасности, легко встраиваемых в технологические процессы разработки, группа безопасности должна искать способы, которые позволили ли бы облегчить работу команде разработчиков, чтобы она могла поставлять ПО быстрее и в то же время безопаснее.

Например, группа безопасности может помочь разработчикам в создании эффективного конвейера сборки и развертывания и предложить простой процесс и инструменты для компиляции, сборки, тестирования и автоматического развертывания системы, так чтобы на всех этапах имел место контроль безопасности.

Группа безопасности может также предложить средства для внутреннего обучения, например проект WebGoat (*https://www.owasp.org/index.php/Category:OWASP_WebGoat_Project*) сообщества OWASP[1], проект Damn Vulnerable Web Services (*https://github.com/snoopysecurity/dvws*) или другие заведомо уязвимые приложения, на которых разработчики могут учиться находить проблемы безопасности и устранять их последствия.

Группа безопасности должна приложить все усилия к тому, чтобы самый простой способ создать какую-нибудь систему внутри организации был безопасным. Для этого можно знакомить команды с безопасными HTTP-заголовками, рецептами и сценариями создания особо стойких конфигураций на этапе выполнения, а также заведомо не содержащими уязвимостей сторонними библиотеками и образами, которые можно просто взять и сразу использовать. Всё это мы рассмотрим в последующих главах.

[1] Открытый проект обеспечения безопасности веб-приложений.

Когда группа безопасности перестает все время говорить нет и становится помощником в деле поставки надежного кода, – вот тогда можно говорить об истинной гибкой безопасности.

Инструменты проверки соответствия нормативным требованиям и аудита

А что делать после передачи системы в эксплуатацию? Как и в случае инструментов для простого тестирования на уязвимость, хорошая группа безопасности знает, что должна подтверждать соответствие нормативным требованиям и проводить аудит. Так почему бы не автоматизировать большую часть этого процесса и не дать командам разработчиков доступ к тем же инструментам?

Создайте инструмент, который сверяет список пользователей, имеющих права доступа к системе, с базой данных отдела кадров. Это позволит проверять, что учетные данные уволившихся сотрудников отозваны.

А как насчет инструмента, который имеет API для подтверждения того, что все узлы в облачной инфраструктуре построены на основе безопасного базового образа, что на них установлены все необходимые исправления и что они правильно помещены в группы безопасности?

Такие инструменты аудита и подтверждения соответствия помогают и группе безопасности, и эксплуатационникам своевременно обнаруживать ошибки, высвобождая высококвалифицированных специалистов для аудита действительно интересных и трудных проблем.

Задание контрольного уровня безопасности

Как узнать, делают ли инструменты именно то, что нужно? Как убедиться, что продукт действительно безопасен?

На самом деле в безопасности продукта никогда нельзя быть уверенным на все сто процентов, но можно утверждать, что продукт соответствует некоторому контрольному уровню безопасности.

Включив в гибкий жизненный цикл точки соприкосновения с безопасностью и пользуясь имеющимися инструментами и шаблонами, можно определить контрольный уровень безопасности, которому должен отвечать продукт, и проверять, что каждая новая сборка соответствует этому уровню.

Это придаст вам уверенности при формулировании гарантий качества продукта, и вы будете знать, что таким уровнем качества обладает не только данная сборка, но и все последующие.

А что будет при масштабировании?

Описанная модель хорошо работает, когда команд разработчиков немного и имеется небольшая группа инженеров по безопасности, которых можно равномерно распределить между командами. Это ситуация, в которой находится большинство читателей. Например, если есть шесть команд разработчиков и два инженера-безопасника, то рабочего времени группы безопасности хватит для решения большинства возникающих вопросов.

Но что, если число продуктов или служб продолжает расти?

Если следовать принятой в Amazon модели двух пицц[2], то в организации, где работает 200 человек, должно быть 30 или больше команд, а это значит, что для их поддержки нужно, по крайней мере, 10 инженеров-безопасников. Если следовать принятой в Netflix модели инженерных групп из двух человек, то такая модель безопасности не масштабируется вовсе. Чем больше команд разработчиков, тем меньше шансов, что вы сможете выделить каждой специалиста по безопасности.

В крупных организациях обеспечение безопасности приложений следует рассматривать как конвейерную задачу. Вместо того чтобы пытаться решить проблему в той точке, где уже невозможно справиться с объемом работы, инженеры-безопасники должны подключаться на более поздней стадии конвейера и создать условия, при которых разработчики могут принимать решения о безопасности самостоятельно.

Создание содействующих групп безопасности

Группа безопасности должна не *заниматься безопасностью* самостоятельно, а *содействовать безопасности*. Под этим мы понимаем, что основная задача группы состоит в том, чтобы создавать инструменты, документировать практические методы и обеспечивать возможность разработки и развертывания безопасных служб. По-настоящему гибкие команды безопасности измеряют успех в зависимости от того, что они могут обеспечить, а не в количестве проблем с безопасностью, которым они не позволили выйти за порог организации.

Создание среды, в которой безопасный способ работы является и самым простым, – великая цель, к которой должна стремиться любая группа безопасности. У нее есть и дополнительный бонус – все участники непосредственно вовлечены в повышение безопасности процесса

[2] Не должно быть команд, которых нельзя было бы накормить двумя пиццами, т. е. в команде должно быть от 5 до 7 человек – в зависимости от аппетита.

разработки. Это единственный масштабируемый способ, который позволяет группе безопасности если не повысить, то хотя бы сохранить свое влияние в растущей организации, сталкивающейся с неприглядной реальностью, когда нет ни финансирования, ни достаточного числа квалифицированных профессионалов, чтобы закрыть все потребности в безопасности.

Создание инструментов, которыми будут пользоваться

Под созданием инструментов мы понимаем разработку инструментальных средств обеспечения безопасности, с помощью которых команды разработчиков смогут убедиться в безопасности своих продуктов. Это могут быть средства для управления рисками, анализа дерева атак и учета гибких историй. Или средства автоматического тестирования, встраиваемые в сборочный конвейер, а также средства автоматической проверки зависимостей. Или библиотеки и микросервисы, содержащие средства безопасности, которыми команды могут пользоваться для решения задач в области криптографии, многофакторной аутентификации и аудита. Это может быть также инструментарий для безопасного аудита и исправления конфигурации основных систем безопасности, например брандмауэров, или сторонних служб, предоставляемых, например, компаниями AWS или GCP.

Наш опыт показывает, что попытка заставить команду пользоваться конкретным инструментом ведет к культуре контрольных списков, когда инструмент рассматривается как чужеродный предмет, который никто толком не понимает и которым пользуются неохотно. Команде разработчиков должна быть предоставлена свобода в выборе инструментов, соответствующих ее потребностям и профилю рисков разрабатываемой системы. Инструментов, которые они понимают, которые хорошо ложатся на технологические процессы и которыми они станут владеть.

Важно отметить, что эти инструменты должны не только выявлять дефекты безопасности, но и позволять легко исправить их силами команды. Поэтому инструмент, который просто сканирует систему на предмет безопасности и формирует отчет, ничем не поможет средней команде. Он должен связывать результаты со стандартными мерами по исправлению ошибок, которые заведомо работают. Только такой инструмент имеет ценность.

Инструменты, требующие значительных усилий от разработчиков, неизбежно не будут использоваться. Примером может служить чрезмерно подробный отчет, в котором разработчик вынужден долго

разыскивать интересующую его информацию. Или когда для того, чтобы выяснить, важна некоторая информация или нет, разработчик должен выполнять дополнительные действия или обращаться к внешним источникам.

Крайне важно, чтобы инструменты безопасности допускали расширение посредством API или благодаря применению основополагающего принципа Unix: решить одну задачу и передать результат следующему инструменту. Рассматривайте свой инструментарий как расширяемый набор, который пополняется со временем и позволяет комбинировать инструменты различными способами, вместо того чтобы каждый раз переписывать с нуля.

Методы документирования системы безопасности

Не так уж много написано о хорошо зарекомендовавших себя на практике способах обеспечения безопасности. Безопасность по-прежнему считается темным искусством, которое тайно практикуют в пыльных углах посвященные мастера. Для многих разработчиков безопасность сводится в основном к непонятным ошибкам и методам защитного кодирования, которое, по их мнению, нужно только в особых случаях.

Инженеры-безопасники должны обучать разработчиков правильным методам, применимым к данной организации. Это может быть настройка типичного веб-приложения, советы по эффективной и безопасной работе с поставщиком облачных служб, рекомендации по безопасному кодированию, контрольные списки к инспекциям кода в контексте используемых языков и каркасов, а также перечни типичных рисков в приложениях того типа, который они разрабатывают.

Главное – чтобы все эти методы были применимы, актуальны и релевантны. Рекомендации NIST (*https://csrc.nist.gov/publications/search?requestserieslist=1& requeststatuslist=1,3&requestdisplayoption=brief&itemsperpage=all&requestsortorder=5*) или руководство по надлежащей практике правительства Великобритании, а также другие руководства, публикуемые государством или регуляторами, обычно составлены в таком общем виде и написаны таким бюрократическим языком, что для большинства команд бесполезны.

Поскольку гибкие команды разработчиков ценят работающее ПО превыше документации, то код всегда побивает бумагу. Всюду, где возможно, встраивайте рекомендации по безопасности и контрольные списки непосредственно в код: безопасные заголовки, рецепты и сценарии безопасной настройки, облачные шаблоны, каркасы, в которых средства безопасности включены по умолчанию, а также автома-

тизированные тесты безопасности и проверки соответствия, которые можно включить в конвейеры и запускать в производственной системе. Код, который разработчики могут легко найти и использовать без замедления работы.

Сухой остаток

Гибкая разработка ставит новые вызовы перед группами безопасности. Вот ключи к успеху гибких программ безопасности:

Вовлеченность

В гибком жизненном цикле немало возможностей для совместной работы специалистов по безопасности и разработчиков – мест, где они могут учиться друг у друга и помогать друг другу. Лицо, взявшее на себя заботу о безопасности в команде (инженер-безопасник, приписанный к команде, или разработчик, отвечающий за безопасность), может и должен принимать участие в совещаниях по планированию, утренних планерках, ретроспективном анализе и пошаговых разборах.

Обеспечение

Гибкие команды продвигаются быстро и непрерывно учатся и совершенствуются, а группа безопасности должна помогать в этом, а не ставить препятствия на их пути.

Автоматизация

Проверки и тесты безопасности должны быть автоматизированы способом, который позволяет легко и прозрачно вставлять их в технологические процессы разработки и сборочные конвейеры.

Гибкость

Группа безопасности не должна уступать в гибкости команде разработчиков. Она должна думать и действовать быстро и итеративно, реагировать оперативно, учиться и совершенствоваться вместе с разработчиками.

Глава 5

Безопасность и требования

Разработка любой системы начинается со сбора требований. И система безопасности – не исключение.

В этой главе мы рассмотрим создаваемые сообществом инструменты, в частности стандарт подтверждения безопасности приложений OWASP (Application Security Verification Standard – ASVS), в котором перечислены стандартные механизмы обеспечения безопасности, и список относящихся к безопасности историй, составленный группой SAFECode, который можно использовать при обдумывании и формулировании требований с целью убедиться, что безопасность учтена.

Мы также рассмотрим некоторые простые приемы гибкого определения требований к безопасности и вопрос о том, где и как следует подключать группу безопасности к составлению и управлению требованиями.

Учет безопасности в требованиях

Традиционная разработка в соответствии с каскадной моделью или V-моделью предполагает, что все требования к системе можно собрать и проанализировать заранее, а затем передать полный список команде разработчиков для проектирования, реализации и тестирования. Любые изменения требований считаются исключительной ситуацией.

В методиках гибкой разработки предполагается, что требования или потребности можно понять только при личном общении, поскольку многие функциональные требования выглядят как обсуждение произведений искусства: «Я узнаю это, как только увижу».

Точнее говоря, практики гибкой разработки полагают, что пользователям и заказчикам трудно точно сформулировать требования, потому что язык – это механизм коммуникации с потерями и потому что зачастую слова пользователя о том, что он хочет, – совсем не то, что он хочет на самом деле.

Поэтому в гибких методиках требования собираются итеративно и конкретно, опираясь на личное общение и прототипы, а затем реализуются мелкими и частыми шагами для демонстрации и получения обратной связи.

Но какой бы механизм определения требований ни использовался, часто бывает трудно установить атрибуты безопасности того программного обеспечения, которое мы проектируем и разрабатываем.

Пользователи способны объяснить, как они представляют себе определенные операции программы, но ни один пользователь никогда не скажет, что ему нужны маркеры безопасности на сеансовом уровне для защиты от межсайтовой подделки запросов (CSRF), – да никто и не ожидает, что пользователь знает, что это такое.

В методиках разработки ПО подобные требования объединяют в группу межфункциональных или нефункциональных требований, куда входят безопасность, удобство сопровождения, производительность, стабильность и другие аспекты системы, которые команда должна учитывать при проектировании, кодировании и тестировании.

Но гибкие методики сталкиваются с трудностями при определении межфункциональных или нефункциональных требований, поскольку их сложно связать с конкретными потребностями пользователей, а заказчику или его представителю трудно оценить их приоритет по сравнению с функциями, ориентированными непосредственно на пользователя.

Безопасность и надежность системы часто зависит от фундаментальных решений, принимаемых на ранних стадиях архитектурного и системного проектирования, поскольку их нельзя добавить позже, не выкинув в корзину уже написанный код, чего, конечно, никто не хочет.

Те, кто выступает против гибких методов разработки, указывают, что именно здесь вся методика терпит крах. Отказ от предварительного планирования, заблаговременного определения требований и детального проектирования, акцент на быструю поставку функциональности может привести к зияющему пробелу в важных нефункциональных требованиях к системе, который обнаружится, когда будет уже поздно.

Наш опыт показывает, что гибкий не значит незапланированный или небезопасный. Гибкость – это открытость для изменений и улучшений, и потому мы считаем, что возможно гибко создавать ПО, не упуская из виду требования к безопасности.

Начнем с краткого объяснения того, как подходят к требованиям при гибкой разработке – и почему так делается.

Гибкие требования: рассказывание историй

Большая часть требований в гибких проектах собирается в виде *пользовательских историй*: неформальных описаний того, что и почему хочет пользователь. История – это конкретное описание потребности или определенного решения задачи, в котором четко указано, что пользователю нужно сделать и какой цели он хочет достичь, обычно с точки зрения одного пользователя системы или пользователей одного и того же типа. Истории записываются на простом языке, понятном членам команды и пользователям.

По большей части истории начинаются «эпическим» зачином: длинным и туманным предложением о некоторой потребности системы, которое постепенно распадается на конкретные истории, до тех пор пока члены команды не будут ясно понимать, что им нужно сделать, и более-менее представлять, когда это понадобится.

Истории всегда простые и короткие, они содержат ровно столько информации, чтобы команда могла начать работу, и побуждают команду задавать вопросы и выпытывать у пользователей системы детали. Это заставляет членов команды прилагать усилия, чтобы понять, чего и почему хочет пользователь, позволяет заполнять пробелы и вносить коррективы по ходу реализации решения.

В отличие от проектов, разрабатываемых по каскадной модели, когда руководитель проекта стремится определить объем и содержание полностью и еще до начала работы, а всякие изменения рассматривает как исключения, гибкие команды считают, что изменения неизбежны и готовы к тому, что требования будут изменяться в ответ на новую информацию. Они хотят поставлять работающее ПО быстро и часто, чтобы получать полезные отзывы и реагировать на них.

Это критически важный момент, поскольку означает, что, в отличие от каскадных проектов, где все планируется заранее, гибкие команды стремятся не создавать взаимосвязанных требований. Каждая пользовательская история или фрагмент функциональности должны оставаться автономными, если команда решит в какой-то момент прекратить поставку. Такой подход с фиксированным временем и бюджетом и переменным наполнением является типичным для гибких проектов.

Как выглядят истории?

В большинстве гибких команд используется простой шаблон пользовательской истории, пропагандируемый Майком Коном (*https://www.mountaingoatsoftware.com/ books/user-stories-applied*) и другими:

В качестве {тип пользователя}
я хочу {сделать то-то и то-то},
с тем чтобы {я мог достичь такой-то цели}.

История записывается на каталожной карточке, липкой бумажке или электронном аналоге таковых.

Условия удовлетворенности

По каждой истории команда работает с владельцем продукта для восполнения недостающих деталей функции или изменения и письменно формулирует условия удовлетворенности (*https://www.mountaingoatsoftware.com/blog/clarifying-the-relationship-between-definition-of-done-and-conditions-of-sa*), или критерий приемки. Если истории записываются на карточках, то эти детали записываются на оборотной стороне карточки. *Условия удовлетворенности* – это конкретная функциональность, которую команда должна продемонстрировать в доказательство того, что работа над историей закончена.

Условия удовлетворенности служат команде руководством по проектированию функциональности и составлению списка тестов, которые должны пройти для конкретной истории. В этих критериях описывается, что система должна делать при различных обстоятельствах: какие варианты действий есть у пользователя, как система должна реагировать на действия пользователя и какие ограничения налагаются на действия пользователя.

По большей части следует использовать утвердительные предложения и уделять основное внимание успешным сценариям функции или взаимодействия. А это значит, что большинство тестов будут позитивными, призванными доказать, что сценарий завершается успешно.

При формулировании условий удовлетворенности обычно обращают мало внимания на то, что должно происходить в случае ошибки при выполнении действия, исключения или другого негативного развития событий. В главе 11 мы увидим, что это серьезная проблема с точки зрения безопасности, поскольку противник не ограничивается успешными путями выполнения системного кода. Он ведет себя не так, как обычный пользователь. Он стремится использовать систему некорректно, ищет слабые места и упущения разработчиков, которые дадут ему такой доступ к возможностям и информации, которого быть не должно.

Учет историй и управление ими: журнал пожеланий

Записанные истории добавляются в журнал пожеланий продукта или проекта. *Журнал пожеланий* (backlog) – это список историй, упорядоченный по приоритету, в котором определена вся подлежащая поставке функциональность, а также известные на данный момент изменения и исправления. Команда выбирает из журнала истории с наибольшим приоритетом и планирует работу над ними.

В системе канбан и других моделях с *непрерывным потоком* отдельные члены команды выбирают историю с наивысшим приоритетом из очереди. В Scrum и XP истории выбираются из общего журнала пожеланий продукта на основе приоритета и разбиваются на более мелкие задачи, помещаемые в журнал пожеланий спринта, где определена работа, которую команда собирается выполнить на очередном временном отрезке.

В некоторых гибких средах каждая история записывается на карточке или липкой бумажке. Журнал историй вывешивается на стене, чтобы все члены команды видели, что предстоит сделать.

В других командах, особенно в крупных компаниях, учет историй ведется в электронном виде в таких системах, как Jira, Pivotal Tracker, Rally или VersionOne. Электронная система учета дает некоторые преимущества, особенно с точки зрения соответствия нормативным требованиям.

1. Электронный журнал пожеланий автоматически запоминает историю изменений требований и проекта, позволяет получить контрольный журнал, в котором указано, когда изменение было предложено, кто его одобрил и когда оно было реализовано.
2. С пользовательскими историями можно автоматически связать другие технологические процессы. Например, запись в репозиторий можно пометить идентификатором истории, что позволяет легко проследить всю работу над историей, включая изменения, внесенные в код, инспекции, автоматическое тестирование и даже развертывание функциональности с помощью сборочного конвейера.
3. Можно без труда найти истории, относящиеся к безопасности и подтверждению соответствия нормативным требованиям, истории, связанные с конфиденциальностью информации или исправлениями критических ошибок. И всё это пометить для будущей инспекции.

4. Можно также пометить специальными признаками проблемы безопасности и соответствия нормативным требованиям, чтобы затем проанализировать общую картину и понять, какого рода проблемы возникали в разных проектах и как часто. Эту информацию можно будет использовать для целенаправленного обучения безопасности и другим заблаговременным действиям со стороны группы безопасности.
5. Информацию, хранящуюся в онлайновых системах, проще сделать общей для нескольких команд, особенно в распределенной среде.

Истории в журнале пожеланий продукта постоянно инспектируются, обновляются, уточняются, переставляются в другом порядке, а иногда и удаляются владельцем продукта или другими членами команды. Эта деятельность называется «уходом за журналом пожеланий».

Отношение к дефектам

Считать ли дефекты историями? Как учитывать дефекты? Некоторые команды вообще не ведут учет дефектам. Они либо исправляют их немедленно, либо не исправляют вовсе. Другие команды учитывают только дефекты, которые не смогли исправить немедленно, и добавляют их в журнал пожеланий как технический долг.

А как насчет уязвимостей в системе безопасности? Следует ли учитывать их как дефекты? Или они должны учитываться группой безопасности в рамках программы управления уязвимостями? Об этом мы поговорим подробнее в главе 6.

Включение вопросов безопасности в требования

Для групп безопасности скорость принятия решений в гибких средах разработки и примат работающего программного обеспечения над документацией означает, что нужно быть рядом с командой разработчиков, чтобы понимать, над чем она работает, и не упускать из виду изменение требований и приоритетов.

Выше при обсуждении распределения сотрудников группы безопасности между командами в условиях масштабирования мы говорили, что нужно решить, когда и как мы можем позволить себе привлекать группу безопасности, а когда не можем.

Группа безопасности должна принимать участие в планировании релизов, спринтов и других совещаниях по планированию, чтобы помочь в инспекции и уточнении деталей историй, связанных с безопасностью, соответствием требованиям, и других высокорисковых историй. Участие в планировании позволяет группе безопасности лучше понять, что важно для организации, и дает ей шанс помочь владельцу продукта и другим членам команды правильно расставить приоритеты задачам безопасности и подтверждения соответствия.

По возможности группа безопасности должна также участвовать в ежедневных планерках команды разработчиков, чтобы устранять препятствия и быть в курсе внезапной смены направления.

Группу безопасности необходимо привлекать к инспекциям и обновлениям журнала пожеланий (*уходу за журналом*), она должна также пристально следить за появлением историй, содержащих риски для безопасности, конфиденциальности и соответствия нормативным требованиям.

Группа безопасности не обязана ждать команду разработчиков. Она может сама добавлять в журнал пожеланий истории, касающиеся безопасности, конфиденциальности и соответствия нормативным требованиям.

Но самый лучший способ масштабировать возможности группы безопасности – обучить членов команды идеям и методам, описанным в этой главе, и помочь им в создании персон безопасности и деревьев атак, чтобы они понимали и могли бороться с рисками безопасности сами.

Истории, касающиеся безопасности

Как совместить требования к безопасности с историями?

Истории, касающиеся функций безопасности (создание учетной записи пользователя, изменение и восстановление пароля и т. д.), обычно прямолинейны:

> В качестве {зарегистрированного пользователя}
>
> я хочу {войти в систему},
>
> с тем чтобы {я мог видеть и делать то, на что имею право}.

Такие истории мало чем отличаются от прочих. Но из-за рисков, сопряженных с ошибкой в реализации этих функций, следует обращать повышенное внимание на критерии приемки. Приведем несколько примеров и тестовых сценариев.

Пользователь успешно входит в систему

Что должен увидеть пользователь, и что он может дальше делать? Какая информация должна быть записана и куда?

Пользователь не смог войти в систему из-за неправильных учетных данных

Какое сообщение (сообщения) об ошибке должен увидеть пользователь? Сколько попыток есть у пользователя, прежде чем он будет отключен, и на какое время он будет отключен? Какая информация должна быть записана и куда?

Пользователь забыл учетные данные

Это должно стать поводом для другой истории – о восстановлении пароля.

Пользователь не зарегистрирован

Это должно стать поводом для другой истории – о том, как пользователь регистрируется и получает учетные данные.

Применение стандарта OWASP ASVS для определения критерия приемки

Стандарт подтверждения безопасности приложений проекта OWASP (ASVS) – ценный ресурс при написании историй, касающихся безопасности, особенно для мобильных и веб-приложений.

Это открытый стандарт, предназначенный для аудиторов безопасности, но его могут использовать также разработчики и тестировщики при определении требований и в особенности критериев приемки – просто пропустите начальные слова об аудите и переходите сразу к контрольным спискам.

В контрольные списки ASVS включены пункты, позволяющие удостовериться, что управление пользователями поставлено правильно. Там же есть все необходимое для проверки того, как реализованы контроль доступа, аудит, протоколирование, криптографические функции и прочие вещи, относящиеся к безопасности. Мы рекомендуем, чтобы опытный специалист по безопасности просмотрел контрольные списки и выбрал критерии оценки, соответствующие вашему проекту.

Для приведенной выше истории о входе пользователя в систему в ASVS 3.0 перечислено 28 пунктов, которые нужно проверить, чтобы убедиться в правильности реализации.

1. Проверить, что по умолчанию все страницы и ресурсы требуют аутентификации – за исключением тех, что явно объявлены открытыми.

2. Проверить, что формы для ввода учетных данных не заполняются приложением автоматически. Предварительное заполнение приложением означало бы, что учетные данные хранятся в виде открытого текста или в обратимом виде, что, безусловно, запрещено.
3. Проверить, что аутентификация производится только на стороне сервера.
4. Проверить, что ошибка аутентификации реализована так, что не дает противнику никакой информации для входа в систему.
5. Проверить, что поля для ввода пароля позволяют и даже поощряют использование парольных фраз и не препятствуют использованию диспетчеров паролей, длинных парольных фраз или очень сложных паролей.
6. Проверить, что все функции аутентификации владельца учетной записи (обновление профиля, восстановление пароля, заблокированный или утраченный маркер безопасности, обращение в службу технической поддержки или к автоинформатору), которые могут восстановить доступ к учетной записи, не менее устойчивы к атакам, чем основной механизм аутентификации.
7. Проверить, что при изменении пароля система запрашивает старый пароль, новый пароль и подтверждение нового пароля.
8. Проверить, что все решения, относящиеся к аутентификации, протоколируются без сохранения секретной информации, в т. ч. идентификаторов сеансов и паролей. Это относится и к запросам релевантных метаданных, необходимых при расследованиях, связанных с безопасностью.
9. Проверить, что для паролей учетных записей вычисляется односторонний хеш с начальным значением (солью) и что фактор трудозатрат достаточно велик, чтобы предотвратить попытки взлома пароля полным перебором и атаки с восстановлением пароля по его хешу.
10. Проверить, что учетные данные передаются по зашифрованному каналу связи и что все страницы и функции, требующие от пользователя ввода учетных данных, применяют для этой цели зашифрованный канал.
11. Проверить, что функция «забыли пароль?» и другие способы восстановления не раскрывают текущий пароль и что новый пароль не передается пользователю в открытом виде.
12. Проверить, что при входе в систему и при восстановлении пароля невозможен последовательный перебор имен пользователей.
13. Проверить, что ни в самом приложении, ни в одном из его компонентов не существует паролей по умолчанию.
14. Проверить, что реализованы механизмы антиавтоматизации, препятствующие полному перебору паролей или пользователей, а также атакам с целью блокировки учетных записей.

15. Проверить, что все учетные данные для аутентификации при доступе к внешним службам зашифрованы и хранятся в защищенном месте.
16. Проверить, что в функции «забыли пароль?» и других способах восстановления передается не окончательный пароль, а одноразовый пароль с ограниченным сроком действия или сгенерированный программой маркер, используется передача на мобильное устройство или иной механизм офлайнового восстановления. Передача случайного пароля по электронной почте или в SMS должна быть средством на крайний случай, поскольку этот механизм заведомо уязвим.
17. Проверить, что есть два вида блокировки учетной записи: мягкая и жесткая, – и что они не являются взаимно исключающими. Если учетная запись временно заблокирована из-за атаки с полным перебором, то истечение срока блокировки не должно приводить к отмене жесткой блокировки.
18. Проверить, что если используются вопросы, основанные на знании общей информации («секретные вопросы»), то сами вопросы не нарушают законов о неприкосновенности частной жизни и достаточно специфичны, чтобы защитить учетные записи от злонамеренного восстановления пароля.
19. Проверить, что настройка системы допускает отмену конфигурируемого числа предыдущих паролей.
20. Проверить, что для операций высокой ценности применяется повторная аутентификация, например двухфакторная аутентификация или подписывание операций.
21. Проверить, что приняты меры против использования широко распространенных паролей и слабых парольных фраз.
22. Проверить, что среднее время всех попыток аутентификации, как успешных, так и неудачных, одинаково.
23. Проверить, что секреты, ключи API и пароли не включены в исходный код или онлайновые репозитории исходного кода.
24. Проверить, что если приложение допускает аутентификацию пользователей, то они могут использовать двухфакторную или другую стойкую аутентификацию либо иную подобную схему, препятствующую раскрытию пары имя пользователя + пароль.
25. Проверить, что интерфейсы администратора недоступны ненадежной стороне.
26. Применение имеющейся в браузере функции автозаполнения и интеграция с диспетчерами пароля разрешены, если явно не запрещены политикой управления рисками.

Вот сколько всего нужно проверить в казалось бы простой истории! И до скольких из этих пунктов вы могли бы додуматься сами?

Стандарт OWASP ASVS создавался в течение нескольких лет самыми толковыми людьми в сфере безопасности. Обязательно воспользуйтесь им. В следующей главе мы покажем, как это сделать в процессе инспекции кода.

Конфиденциальность, мошенничество, соответствие нормативным требованиям и шифрование

Помимо собственно безопасности, есть еще ряд требований, которые иногда необходимо учитывать.

Конфиденциальность

Определение конфиденциальной или секретной информации, которую необходимо защищать с помощью шифрования, генерации маркеров безопасности, контроля доступа и аудита.

Защита от мошенничества

Управление идентификацией, принудительное разделение обязанностей, проверка и одобрение всех шагов ключевых технологических процессов, аудит и протоколирование, выявление паттернов поведения, задание пороговых значений и оповещение в случае исключительных ситуаций.

Соответствие нормативным требованиям

Что необходимо включить в состав средств контроля (аутентификация, контроль доступа, аудит, шифрование), и наличие чего необходимо доказать при проверке процессов разработки и эксплуатации?

Нормативные требования налагают ограничения на способы работы команд, предписывают, что следует делать в процессе инспекции и тестирования, какие одобрения и виды надзора необходимы, а также какие доказательства совершения всех этих действий должна сохранять команда в процессе разработки и поставки системы. Вопросу о соответствии нормативным требованиям в организациях, практикующих гибкие методики и DevOps, будет посвящена отдельная глава.

Шифрование

У требований к шифрованию есть две стороны:

1. Понять, какая информация нуждается в шифровании.
2. Как именно следует шифровать: какие разрешены алгоритмы и методы управления ключами.

Вопрос о соответствии нормативным требованиям и конфиденциальности (и тут опять не обойтись без криптографии) мы будем рассматривать в главе 14.

Команда должна найти свой способ учета этих требований и ограничений: в виде инструкций по разработке, контрольных списков при на-

писании истории или в форме «определения готовности» – контракта, заключенного командой с самой собой и с другими частями организации, о том, что должно быть сделано, чтобы историю можно было считать законченной и готовой к поставке.

Криптографические требования: здесь водятся драконы

Шифрование – это та область, где необходима особая тщательность реализации, чтобы соблюсти требования. Некоторые требования исходят от регулирующих органов. Например, стандарт безопасности данных индустрии платёжных карт (Payment Card Industry Data Security Standard – PCI DSS), разработанный для систем обработки данных кредитных карт, предъявляет следующие явные требования к криптографии:

1. В разделе 3 перечисляются данные, которые должны быть представлены маркером безопасности, подвергнуты одностороннему хешированию или зашифрованы, а также формулируются требования к стойкой криптографии и управлению ключами (генерация и хранение ключей, распределение ключей, ротация и истечение срока действия ключей).
2. В глоссарии PCI DSS определено понятие «стойкая криптография» и приведены примеры приемлемых стандартов и алгоритмов. Затем дается отсылка к «текущей версии специальной публикации NIST 800-57, часть 1, где можно найти более подробную информацию по стойкости криптографических ключей и алгоритмов». В разделе глоссария «Генерация криптографических ключей» даются ссылки на другие руководства, в которых описано, как следует организовывать управление ключами.

Не сказать, чтобы это было просто и понятно, но криптография – вообще дело не простое. Это та область, в которой непосвященному лучше прибегнуть к помощи эксперта. И главное – никогда не пытайтесь изобрести собственный криптографический алгоритм или модифицировать опубликованный чужой алгоритм.

Соблюдение нефункциональных требований, в т. ч. к безопасности и надежности, – нерешенная проблема гибкой разработки. Эксперты спорят, что считать «правильным подходом» к этой проблеме. Но все согласны, что как-то ее надо решать. Важно, чтобы команда придумала свой способ выявления и учета этих требований, а затем придерживалась его.

Истории, касающиеся безопасности, с точки зрения SAFECode

Форум SAFECode (*https://safecode.org/*) (Software Assurance Forum for Excellence in Code) – отраслевая ассоциация, в которую входят, в частности, компании Adobe, Oracle и Microsoft, разрабатывающая рекомендации по безопасности и качеству ПО. В 2012 году она опубликовала документ «Practical Security Stories and Security Tasks for Agile Development Environments» (*http://safecode.org/publication/ SAFECode_Agile_Dev_Security0712.pdf*), в котором поделилась некоторыми идеями о том, как включить безопасность в процедуру гибкого планирования и реализации требований.

SAFECode предложила истории, призванные предотвратить типичные уязвимости, встречающиеся в приложениях: XSS, выход за пределы назначенного каталога, удаленное выполнение, CSRF, внедрение команды ОС, внедрение SQL и полный перебор паролей. Другие истории посвящены предотвращению раскрытия информации в сообщениях об ошибках, правильному применению шифрования, аутентификации и управлению сеансами, безопасности транспортного уровня, ограничению загрузок на сервер и переадресации на URL ненадежных сайтов.

Существуют также истории, вникающие в детали кодирования: сравнение с нулевым указателем, проверка выхода за границы, числовые преобразования, инициализация, синхронизации процессов или потоков, обработка исключений и использование небезопасных или ограниченных функций. Есть истории, где описывается безопасная практика разработки и эксплуатации: использование последней версии компилятора, своевременное применение исправлений к исполняющей среде и библиотекам, использование статического анализа, сканирование на наличие уязвимостей, учет и исправление ошибок в системе безопасности, а также более продвинутые методы, требующие помощи со стороны специалистов по безопасности: фаззинг (случайный поиск уязвимостей), моделирование угроз, тестирование на проникновение и повышение защищенности операционной среды.

В общем, это весьма полный список рисков, которыми следует управлять, и безопасных методов разработки, которых лучше придерживаться в большинстве проектов. Но, несмотря на высокое качество содержания, формат оставляет желать лучшего.

Чтобы понять, в чем дело, рассмотрим две истории, взятые из SAFECode.

> В качестве архитектора или разработчика я хочу гарантировать и в качестве контролера качества я хочу проверить, что для совершения некоторого действия выполняются одни и те же шаги в одном и том же порядке, без какого-либо отклонения, намеренного или случайного.
>
> В качестве архитектора или разработчика я хочу гарантировать и в качестве контролера качества я хочу проверить, что ущерб, причиненный системе или ее данным, будет ограничен, если неавторизованное лицо сможет получить контроль над процессом или иным способом повлиять на его поведение непредвиденным способом.

Как видим, истории SAFECode – это хотя и предпринятая с наилучшими намерениями, но очень корявая попытка представить нефункциональные требования в формате гибких пользовательских историй. Многие команды отвернутся от этого подхода, сочтя его искусственным, натянутым и совершенно чуждым тому, как они думают и работают.

Хотя истории SAFECode выглядят как настоящие, их нельзя выбрать из журнала пожеланий и поставить заказчику, как другие истории, и нельзя удалить из журнала по завершении, поскольку работа над ними никогда не «завершается». Команда обязана держать эти вопросы в фокусе внимания на протяжении всего проекта и жизни системы.

Каждой истории SAFECode соответствует перечень детальных задач в журнале пожеланий, которые команда должна рассматривать на этапе планирования спринта или во время работы над отдельными пользовательскими историями. Но по большей части эти задачи сводятся к напоминанию разработчикам о необходимости следовать наставлению по безопасному кодированию, проводить сканирование и другие проверки безопасности.

Команда может решить, что непрактично и даже вредно учитывать все эти повторяющиеся задачи в журнале пожеланий. Какие-то проверки следует включить в определение готовности историй или спринта в целом. Другие должны стать частью принятого командой наставления по кодированию и войти в контрольные списки для инспекций, либо быть добавлены в автоматизированный сборочный конвейер, либо рассматриваться при обучении команды безопасному кодированию, чтобы все члены команды с самого начала знали, как делать правильно.

Истории SAFECode, касающиеся безопасности, не следует рассматривать как инструмент, который нужно любой ценой впихнуть гибкой

команде. Но это способ выложить требования к безопасности на стол. Анализ и обсуждение этих историй поможет завязать с командой разговор о безопасных практиках и мерах контроля и побудит членов команды предлагать собственные идеи.

Бесплатное руководство по безопасности и учебный курс от SAFECode

SAFECode также бесплатно предлагает обучение и наставление по безопасному кодированию, которому могут следовать команды при создании безопасных систем.

Сюда входит бесплатное руководство по безопасной разработке, особенно полезное для пишущих на C/C++. В нем рассмотрены типичные проблемы безопасности и приведены многочисленные ссылки на инструменты, статьи и другие подсобные средства.

Для тех, кто не может позволить себе более полное обучение безопасной разработке, SAFECode предлагает набор вводных онлайновых учебных курсов по безопасному кодированию на C/C++ и Java, криптографии, моделированию угроз, безопасности облачных вычислений, тестированию на проникновение. Описано также, как защититься от конкретных атак, например внедрения SQL.

Персоны и антиперсоны безопасности

Персоны – еще один инструмент, который во многих командах безопасности применяется в процессе определения требований и проектирования функциональности. Персона – беллетристическое описание различных типов пользователей системы. Для каждой персоны создается вымышленная биография, в которой описаны образование и опыт работы, технические навыки, цели и предпочтения. Эти профили можно составить на основе собеседований, исследований рынка или во время мозговых штурмов.

Персоны помогают команде лучше понять образ мышления пользователей на конкретных примерах, уяснить, как и почему тот или иной человек может использовать некоторую функцию. Это бывает полезно для построения моделей взаимодействия пользователя с системой. Персоны используются также во время тестирования для придумывания различных тестовых сценариев.

Если команда уже использует персон, то имеет смысл предложить ей рассмотреть также *антиперсон*: пользователей, не соблюдающих правила работы с системой.

Проектировщики и команды ищут, как сделать работу с системой максимально простой и интуитивно понятной. Но в некоторых случаях группа безопасности может потребовать сознательного снижения скорости или применения других антипаттернов, поскольку все мы понимаем, что нашей системой будут пользоваться и противники, и хотим затруднить им достижение своих целей.

Мы рекомендуем создавать по одной персоне для каждой «категории» или «класса» пользователей. Мало что дает создание слишком большого числа персон или слишком сложных персон, если можно обойтись несколькими простыми.

Например, один из авторов работал над системой, в которой было 11 пользовательских персон и всего 5 антиперсон: член группы хакеров, мошенник, член организованной преступной группы, автор вредоносных программ и скомпрометированный системный администратор.

В ходе детализации антиперсоны важно учитывать мотивы противника, его технические возможности и пороги отказа от намерений. Важно понимать, что противниками могут быть также легальные пользователи, имеющие побудительные мотивы для взлома системы. Например, при разработке онлайновой системы обращений за страховой выплатой следует рассмотреть пользователей, готовых лгать для получения большей денежной суммы.

Персоны используются всей командой при проектировании всей системы. Не следует ограничивать их одним приложением, поскольку понимание того, как можно атаковать бизнес-процессы, третьи стороны и физические объекты, может оказаться важным. Может случиться, что команда создает компьютерное решение, являющееся лишь одной из частей более крупного бизнес-процесса, а персоны представляют людей, которые хотят атаковать этот процесс через приложение.

Вот несколько примеров простых антиперсон.

- Брайан – полупрофессиональный мошенник:
  - его интересуют атаки с рентабельностью не менее 10 000 фунтов стерлингов;
  - Брайан не хочет попасться и не будет делать ничего такого, что, по его мнению, может оставить следы;
  - у Брайана есть доступ к простым хакерским инструментам, но мало опыта работы с компьютерами, писать код он не умеет.

- Лаура имеет низкие доходы и живет на пособие:
  - Лаура не считает ложь системе социального страхования аморальным поступком и хочет получить настолько большое пособие, насколько удастся;
  - у Лауры есть друзья, хорошо разбирающиеся в том, как работает система льгот и пособий;
  - у Лауры нет технического опыта.
- Грэг – хакер-любитель из онлайновой группы хакеров:
  - Грэг хочет взламывать сайты или еще как-то заявить о себе;
  - Грэг мечтает взломать столько сайтов, сколько удастся, и ищет легкие жертвы;
  - у Грэга нет деловой хватки, он не знает, как эксплуатировать уязвимости ради извлечения прибыли;
  - Грэг немного умеет программировать, он способен написать скрипт и модифицировать код готовых инструментов.

Другие примеры антиперсон и сведения о том, как использовать персон и антиперсон при анализе требований к безопасности и моделировании угроз, см. в приложении C книги Adam Shostack «Threat Modeling: Designing for Security» (издательство Wiley).

Истории противника: надеваем черную шляпу

Другой способ включить безопасность в требования дают истории противника или сценарии злонамеренного использования (вместо сценариев использования). В этих историях команда размышляет о том, как противник или иной злонамеренный, а может быть, просто беспечный пользователь мог бы использовать некую функцию во вред. Это заставляет команду думать, от каких конкретно вещей она должна защищаться. Вот пример:

> В качестве {противника определенного вида}
> я хочу {сделать что-то плохое}
> с целью {украсть, или повредить секретную информацию,
> или получить что-то, не заплатив, или нарушить работу
> важной функции системы, или еще что-то в этом роде…}.

Такие истории более конкретны и тестопригодны, чем истории SAFECode. Вместо критерия приемки, который подтверждается успешными сценариями, в каждой истории противника имеется список «негативных критериев» или «критериев опровержения»: условий

или сценариев, которые нужно опровергнуть, чтобы история считалась готовой.

Возьмите пользовательскую историю и в процессе ее детализации и составления списка сценариев отступите назад и взгляните на нее сквозь призму безопасности. Думайте не только о том, что пользователь хочет и может сделать, но и о том, что мы хотим не позволить ему сделать. Пусть те же люди, что работают над историей, «наденут черные шляпы» и на время превратятся в злоумышленников, придумывающих негативные сценарии.

Для большинства разработчиков глядеть на мир глазами противника нелегко и неестественно, мы поговорим об этом в главе 8. Но стоит попрактиковаться – и это умение придет. Хороший тестировщик сможет предложить идеи и тестовые сценарии, особенно если у него есть опыт исследовательского тестирования. На крайний случай можно пригласить специалиста по безопасности, чтобы он помог разработать такие сценарии, особенно для функций, связанных с безопасностью. Можно также ознакомиться с типичными атаками и контрольными списками требований типа историй SAFECode или стандарта OWASP ASVS.

Для изучения сценариев злонамеренного использования могут очень пригодиться антиперсоны. После слов «в качестве» можно указать имя антиперсоны, это поможет разработчикам написать предложение после слов «с целью».

Истории противника и моделирование угроз

Написание историй противника, или сценариев злонамеренного использования, частично пересекается с моделированием угроз. То и другое означает взгляд на систему с точки зрения атакующего или иного злоумышленника. Оба метода помогают закрыть бреши в системе безопасности заблаговременно, но делается это на разных уровнях.

- Истории противника излагаются с точки зрения пользователя, когда технологические процессы и взаимодействия с пользователем описываются так, будто система – это черный ящик.
- При моделировании угроз система рассматривается как белый ящик – разработчик или проектировщик оценивает средства контроля и предположения о доверии с точки зрения человека, знакомого с внутренним устройством системы.

Истории противника поддаются автоматизированному тестированию. Это особенно полезно для команд, практикующих разработку через тестирование (TDD) или разработку на основе поведения (BDD), когда автоматизированные тесты для каждой истории пишутся еще до написания кода и используются для обдумывания дизайна. Включив тесты для историй противника, разработчики могут гарантировать, что средства контроля и безопасности невозможно отключить или обойти.

Написание историй противника

Истории противника – зеркальное отражение пользовательских историй. Необязательно писать истории противника для каждой пользовательской истории в системе. Но нужно написать их, по крайней мере, в следующих случаях:

- когда пишутся истории, касающиеся функций безопасности, например входа в систему;
- когда пишутся или изменяются истории, в которых речь идет о деньгах, персональных данных или важных административных функциях системы;
- когда история обращается к другим службам, имеющим дело с деньгами, персональными данными или важными административными функциями, чтобы гарантировать, что описываемая функция не станет черным ходом.

Это те типы пользовательских историй, которые особенно интересны для атакующих и мошенников. Именно в этих случаях нужно примерить личину атакующего и взглянуть на систему с точки зрения противника.

Как мы видели, противник необязательно является хакером или киберпреступником. Это может быть работник, затаивший обиду, эгоистичный пользователь, желающий поживиться за счет других, или конкурент, стремящийся украсть информацию о ваших клиентах или интеллектуальной собственности. Но *противником* может быть и администратор, от разрушительных ошибок которого необходимо защититься, или внешняя система, которая не всегда ведет себя корректно.

Подвергните сценарии пользовательской истории пристрастному изучению, задавая следующие вопросы.

1. Что могло бы пойти не так? Что случится, если пользователь не пойдет по основному пути сценария? Какие проверки необходимо добавить и что может случиться, если проверка не пройдет?

Обращайте внимание на граничные значения, редактирование данных, обработку ошибок и необходимые виды тестирования.

2. Спросите себя об идентификации пользователя и данных, которые предоставляются в сценарии. Можно ли им доверять? На чем основана ваша уверенность?
3. За какой информацией может охотиться противник? Какую информацию он уже видит и что может с ней сделать?
4. Ведется ли все необходимое протоколирование и аудит? Когда следует отправлять сигнал тревоги или другое уведомление?

Воспользуйтесь этим упражнением, чтобы сформулировать критерии опровержения (пользователь может сделать то, но не может сделать это; пользователь может увидеть то, но не это) вместо или вместе с условиями удовлетворения истории. Назначьте сценариям приоритеты в зависимости от риска и добавьте сценарии, которые считаете заслуживающими этого, в текущую историю или, если они достаточно велики, в качестве новых историй в журнал пожеланий[1].

В качестве атакующего я НЕ ДОЛЖЕН иметь возможность...

Есть и еще один вариант писать истории противника – описывать в истории, что атакующий не должен иметь возможности сделать:

> В качестве {противника определенного вида}
>
> я НЕ ДОЛЖЕН иметь возможности {сделать что-то плохое}
>
> с целью...

Иногда это проще, чем пытаться написать историю с точки зрения атакующего, потому что вы не обязаны ни понимать, ни описывать конкретные шаги предпринимаемой атаки. Вы просто акцентируете внимание на том, чего не хотите позволить противнику: вы не хотите, чтобы он мог видеть или изменять информацию о пользователе, выполнять операцию с высокой ценностью, не авторизовавшись, обойти проверку кредитного лимита и т. д.

Команда должна будет позже задать критерий приемки, указав, какие конкретно действия нужно проверять и что тестировать, но уже сама история делает требования наглядными – чем-то таким, чему нужно назначить приоритет и что необходимо включить в план.

[1] Это описание историй противника основано на работе Джуди Нэер (Judy Neher), независимого консультанта по гибким методикам безопасности. Смотрите беседу с ней по адресу https://www.youtube.com/watch?v=hxQ1j_HQhGw.

Истории противника хороши, когда нужно выявить уязвимости в бизнес-логике, проанализировать средства обеспечения безопасности (например, аутентификацию, контроль доступа, управление паролями и лицензирование) и средства против мошенничества, довести до ума обработку ошибок и валидацию данных, а также предотвратить нарушение законов о защите частной жизни. И еще они могут помочь команде в подборе дополнительных и улучшенных тестовых сценариев.

Написание таких историй хорошо стыкуется со способом мышления и работы гибкой команды. Они находятся на одном уровне с пользовательскими историями, в них применяется тот же язык и тот же подход. Это более конкретный способ рассуждения об угрозах, чем упражнение в моделировании угроз, и более полезный, чем попытка отслеживать длинный список вещей, которые нужно или, наоборот, не нужно делать.

В результате мы получаем конкретные, осуществимые тестовые сценарии, которые команда и, в частности, владелец продукта легко могут понять и оценить. Это особенно важно в методике Scrum, поскольку владелец продукта решает, какие работы делать и в каком порядке. А поскольку истории противника обрабатываются наряду со всеми остальными людьми, которые работают над историями (а не в ходе отдельных инспекций, которые еще нужно координировать и планировать), то они с большей вероятностью будут сделаны нормально.

Полчаса, потраченные на обдумывание части системы с этой точки зрения, должны помочь команде в поиске и заблаговременном предотвращении слабых мест. По мере возникновения новых угроз и рисков или появления новых видов атак и эксплойтов необходимо возвращаться к этому вопросу, пересматривать существующие и писать новые истории противника для закрытия возможных брешей.

Деревья атак

Дерево атак – сравнительно новая методология выявления путей атаки на систему.

Впервые этот подход был описан Брюсом Шнайером в 1999 году, когда он предложил структурный метод представления растущей цепочки атак (*https://www.schneier.com/academic/archives/1999/12/attack_trees.html*).

Чтобы построить дерево атак, мы начинаем с выписывания целей противника. Это может быть дешифрирование секретных данных, получение доступа с правами суперпользователя или подача мошеннической заявки на получение пособия.

Затем мы изображаем все возможные способы достижения этих целей. Канонический пример из работы Шнайера – чтобы открыть сейф, можно подобрать отмычку к замку, узнать шифр замка, разрезать сейф или допустить ошибку при его установке.

Затем мы обходим дерево атак, выделяя точки, в которых, как нам кажется, мы можем быть вовлечены; например, чтобы узнать шифр, можно было бы найти бумажку, на которой он записан, или попытаться выудить шифр у жертвы.

Современные деревья атак могут оказаться очень широкими или очень глубокими. Имея дерево, мы можем зайти в каждую вершину и оценить такие свойства, как вероятность, стоимость, легкость организации атаки, повторяемость, шанс быть пойманным и итоговый выигрыш противника. Какие свойства выбрать, зависит от понимания противников и от того, сколько времени и сил вы готовы потратить.

Нетрудно выявить вершины с повышенным риском, для этого нужно вычислить отношение «затраты–выгода» для атакующего.

Отыскав области с наибольшим риском, мы можем рассмотреть контрмеры, например: обучение персонала, патрулирование и охранные системы.

Если все сделано правильно, то мы сможем проследить все средства контроля, определить, почему некоторое средство было установлено в этом месте, и оценить его значимость. Если привлечь дополнительное финансирование, то сможем ли мы усилить защиту, поставив более дорогой брандмауэр или установив систему управления секретами? Можно также определить, какие средства контроля следует заменить. Так, если некоторое средство обеспечения безопасности непригодно в данной системе, то можно будет понять, какой эффект возымеет его удаление и какие при этом возникнут риски.

Опытные пользователи метода деревьев атак могут строить деревья для отдельной системы, а также на уровне отдела, подразделения или даже всей организации. Один из авторов с помощью этого метода смог полностью изменить структуру расходов организации на безопасность, направив их на устранение реальных рисков, а не на традиционные, с точки зрения службы безопасности, места приложения денег (подсказка: в частности, было потрачено гораздо меньше денег на навороченные брандмауэры).

Построение дерева атак

Построение дерева атак – интересный опыт. Это весьма действенная, но и весьма субъективная техника.

По нашему мнению, деревья лучше всего строить в ходе нескольких совещаний, в которых принимают участие специалисты по безопасности, технические специалисты и представители бизнеса.

На первом совещании в присутствии специалиста по безопасности следует прикинуть основные цели рассматриваемых деревьев. Например, на одном недавнем совещании, посвященном механизму федеративного входа в систему, мы пришли к выводу, что беспокоиться стоит только о трех целях: вход в систему, кража учетных данных и отказ от обслуживания.

Для анализа типичных угроз применяется акроним STRIDE:

- подлог удостоверения пользователя (Spoofing user identity):
- манипулирование – подделка программных средств и документов (Tampering);
- отрицание совершенных действий (Repudiation);
- разглашение информации (Information disclosure):
- отказ от обслуживания (Denial of service);
- повышение привилегий (Elevation of privilege).

Рассмотрите свою систему с точки зрения этих угроз и сформулируйте набор целей. Затем созовите совещание с участием специалистов разного профиля. Мы обнаружили, что на этом этапе ценнее всего представители бизнеса. Безопасники и технари проявляют изобретательность в способах достижения цели, а знатоки бизнеса лучше понимают сами цели и обычно знают об ограничениях системы (тогда как безопасники и технари склоны считать компьютерные системы идеальными и незапятнанными). Представители бизнеса гораздо более реалистично относятся к изъянам и ручным обходным решениям, благодаря которым система вообще работает.

По завершении построения деревьев специалисты-безопасники могут на славу поработать, измеряя такие свойства, как затраты и т. п.

Сопровождение и использование деревьев атак

Написанные и переданные командам деревья атак, конечно, со временем устаревают и перестают отражать действительность, поскольку и контекст, и ситуация меняются.

Деревья атак можно хранить в цифровой форме – в виде электронных таблиц или *диаграмм связей* (mind map); последний формат показался нам вполне эффективным.

Их следует регулярно пересматривать. В особенности нужно проверять, не произошло ли существенных изменений в профиле безопасности после модификации мер контроля безопасности.

По меньшей мере, один раз нам встретилась компания, которая хранила деревья атак на вики-странице, а все средства контроля были привязаны к цифровым карточкам историй, так что состояние каждой истории динамически отражалось в представлении. Тем самым группа безопасности всегда видела текущее состояние дерева атак и запланированные работы, которые могли повлиять на него. А уполномоченные по соответствию нормативным требованиям могли проследить весь путь заказа-наряда и понять, почему он был размещен и когда выполнен.

Вы сами должны решить, что будет работать для вашей команды, групп безопасности и уполномоченных по соответствию, но такого рода взаимосвязи очень ценны для высокопроизводительных команд, поскольку позволяют хорошо ориентироваться в обстановке и принимать правильные решения.

Требования к инфраструктуре и эксплуатации

Из-за высокой скорости, с которой современные гибкие – и особенно практикующие DevOps – команды передают системы в производственную среду, а также из-за темпа внесения изменений в уже используемые системы разработчики должны располагаться гораздо ближе к инфраструктуре и эксплуатационникам. Теперь нет специальных церемоний передачи в отдельные группы эксплуатации и обслуживания, когда разработчики переходят к другому проекту, как то было в каскадной модели. Модель работы команд основана на концепции оказания услуг – команда несет совместную, а иногда и полную ответственность за эксплуатацию и поддержку системы. Она связана с системой на всем протяжении срока ее службы.

Это означает, что команда должна думать не только о тех, кто будет пользоваться системой, но и о тех, кто будет ее эксплуатировать и поддерживать: инженерах службы обеспечения инфраструктуры и сети, эксплуатационниках и службе поддержки клиентов. Все они становятся заказчиками и партнерами по принятию решений о том, как следует проектировать и реализовывать систему.

NoOps и «кто написал, тот и эксплуатирует»

Несколько лет назад Netflix приняла решение отдать все свои ИТ-операции и инфраструктуру на аутсорсинг Amazon. Сегодня Netflix – один из самых крупных потребителей облачных служб и интернет-трафика. Позиция Netflix заключается в том, что управление центром обработки данных, подготовка инфраструктуры и сетевые работы – это «нераздельная трудная работа». Она важна и, если делается хорошо, обходится дорого, но лучше поручить ее кому-нибудь другому, а самой сосредоточиться на проектировании и поставке продуктов служб.

В Netflix нет подразделения по техническому обеспечению эксплуатации, все делается с помощью AWS API компании Amazon. В Netflix есть несколько групп инженерного обеспечения платформы, которые создают общие службы, используемые другими инженерными группами. Но каждая такая группа отвечает за проектирование, реализацию, поставку и поддержку написанного ей кода. Netflix называет такую идеологию «NoOps».

В Amazon, как и в Netflix, системы разбиты на микросервисы, которые проектируют и поставляют небольшие команды. Это относится и к службам Amazon AWS, которыми пользуются Netflix и другие клиенты AWS. И, как и в Netflix, команды Amazon несут ответственность за поставку, поддержку и эксплуатацию этой части системы. В Amazon этот подход называют «кто написал, тот и эксплуатирует».

Это крайние примеры того, как изменяются роли подразделений разработки и эксплуатации, по мере того как организация переходит на новые технологические архитектуры и новые способы ускорения и удешевления поставки систем.

Движения DevOps и NoOps набирают популярность, и, как нам кажется, со временем многие более традиционные команды выберут этот путь.

Например, в ходе работы над средствами аудита и протоколирования команда должна учитывать нужды и требования следующих команд:

Бизнес-аналитика

Хранение сведений о пользователях и их действиях, чтобы понять, какие функции представляют для них наибольшую ценность, какие им не нужны, как они пользуются системой и где и как проводят большую часть времени. Эта информация используется в процессе A/B-тестирования, по результатам которого принимаются решения о проектировании будущих продуктов, и анализируется в системах обработки больших данных для поиска тенденций и закономерностей.

Соответствие нормативным требованиям

Соблюдение требований регуляторов об аудите действий, о том, какая информация должна храниться, как долго и кто имеет право доступа к этой информации.

Информационная безопасность

Какая информация нужна для мониторинга атак и криминалистической экспертизы.

Эксплуатация

Мониторинг системы и эксплуатационные показатели, необходимые для управления и планирования на уровне служб.

Разработка

Собственные информационные потребности группы разработчиков – для поиска неисправностей и отладки.

Команды должны понимать и учитывать операционные требования к конфиденциальности, целостности и доступности, не важно, от кого они исходят: от инженерных команд, от группы эксплуатации или от уполномоченных по соответствию нормативным требованиям. Они должны понимать имеющиеся ограничения со стороны подразделений инфраструктуры и эксплуатации, особенно в корпоративных средах, где система работает как часть гораздо большего целого. Они должны уметь отвечать на целый ряд важных вопросов:

- Какой будет исполняющая среда: облачной, локальной или гибридной? Виртуальные машины, контейнеры или физические серверы?
- Какая ОС?
- Какая база данных или иное хранилище?
- Каков объем внешней и оперативной памяти, сколько и каких процессоров?

Как будет обеспечена безопасность инфраструктуры? Каким нормативным требованиям нужно будет соответствовать?

Упаковка и развертывание

Как будут собираться и упаковываться приложение и его зависимости? Как планируется управлять артефактами сборки и развертывания? Какие инструменты и процедуры будут применяться для развертывания системы? Какова протяженность временного окна для обновления – в какой момент можно остановить систему и на какой срок?

Мониторинг

Какую информацию (оповещения, сообщения об ошибках, показатели) должны собирать эксплуатационники для мониторинга и устранения неисправностей? Каковы требования к протоколированию (формат, синхронизация серверов, ротация журналов)? Как будет передаваться по инстанциям информация об ошибках и отказах?

Управление секретами

Какие ключи, пароли и прочие секретные данные нужны системе? Где они будут храниться? Кто должен иметь к ним доступ? С какой частотой они должны ротироваться, и как это планируется делать?

Архивирование данных

Какую информацию необходимо хранить и в течение какого времени, чтобы удовлетворить требования регуляторов, непрерывности бизнеса и прочие? Где должны храниться журналы и в течение какого времени?

Доступность

Как осуществляется кластеризация – на уровне сети, ОС, базы данных и приложения? Каковы директивный срок восстановления и целевая точка восстановления (RTO/RPO) в случае серьезных отказов? Как планируется противостоять DDOS-атакам?

Разделение обязанностей

Разрешен ли разработчикам доступ к производственной системе для поддержки? Разрешено ли тестирование на производственных машинах? Что разработчикам разрешено видеть и делать, а что запрещено? Какие изменения им разрешено вносить без явного разрешения (и разрешено ли вообще)? Какие средства аудита необходимы для подтверждения всего этого?

Протоколирование

Как мы уже видели, протоколирование нужно для самых разных целей: для технической поддержки и поиска неисправностей, для мониторинга, для криминалистических экспертиз, для аналитики. Что именно нужно сохранять для решения всех этих задач: дату и временную метку (с какой точностью?), идентификатор пользователя, IP-адрес источника, узел, идентификатор службы? Что еще? Какого типа информацию следует сохранять на каждом уровне (DEBUG, INFO, WARN, ERROR)?

Должны ли сообщения быть понятны человеку, или они будут разбираться инструментальными средствами? Как планируется защищаться от изменения и подделки записей журнала? Как обнаруживаются лакуны в протоколировании? Какую секретную информацию следует маскировать или не помещать в журналы? Какие системные события и события безопасности следует протоколировать? Стандарт PCI DSS[2] содержит полезные рекомендации по протоколированию и аудиту. Для соответствия PCI DSS необходимо сохранять следующую информацию:

- все попытки доступа к критическим и секретным данным;
- все попытки доступа от имени суперпользователя (администратора);
- все попытки доступа к контрольным журналам (аудит системы аудита);
- все нарушения контроля доступа;
- события аутентификации и изменения данных аутентификации (вход в систему, заведение нового пользователя, изменение пароля и т. д.);
- аудит и протоколирование системных событий: запуск, остановка, приостановка, перезапуск, ошибки.

Безопасность и CIA

Все операционные требования к безопасности отображаются на один или несколько элементов триады, обозначаемой акронимом CIA (C: конфиденциальность, I: целостность, A: доступность).

Конфиденциальность

Гарантирует, что информацию могут читать, потреблять или использовать только уполномоченные пользователи системы.

Целостность

Гарантирует, что информацию могут модифицировать только пользователи, имеющие на это право. Что изменение будет произведено надлежащим образом и корректно запротоколировано.

Доступность

Гарантирует, что информация будет доступна нуждающимся в ней лицам в тот момент, когда они в ней нуждаются.

[2] Стандарт безопасности данных индустрии платежных карт.

Сухой остаток

Ниже перечислены основные моменты, о которых следует помнить при включении задач безопасности в журнал пожеланий.

- Безопасность должна быть неотъемлемой частью мышления и проектирования историй, недостаточно вспоминать о ней только во время кодирования. Помните о ней с самого начала и учите других думать о безопасности.
- Обращайте внимание на пользовательские истории в момент написания и детализации. Не упускайте из виду риски и соображения безопасности.
- Подумайте о построении деревьев атак, которые помогут команде лучше понимать, как противник мог бы атаковать систему и какие меры защиты следует предпринять.
- Пишите истории противника для пользовательских историй с высокой степенью риска: они должны быть зеркальным отражением истории, написанной с точки зрения противника.
- Используйте стандарт OWASP ASVS и истории SAFECode, это поможет при написании историй и условий удовлетворенности для историй, касающихся безопасности.
- Если команда моделирует пользовательские персоны, то помогите ей в создании антиперсон, чтобы иметь представление о противнике, когда придет время позаботиться о выполнении требований и проверке условий.
- При формулировании требований, в т. ч. к безопасности, помните об эксплуатации и инфраструктуре, а не только о функциональности.

Глава 6

Гибкое управление уязвимостями

Новые уязвимости программного обеспечения обнаруживаются ежедневно. Многие организации не ведают об уязвимостях в своих системах, пока не оказывается слишком поздно. Хуже того, разработчики и их начальники зачастую игнорируют уязвимости, о которых знают. Это означает, что противник может эксплуатировать уязвимость месяцами и годами после первого сообщения о ней, применяя автоматизированные сканеры и комплекты эксплойтов.

Управление уязвимостями – одна из самых важных обязанностей группы безопасности. Его цель – создать условия, при которых ответственные лица в организации постоянно проверяют наличие известных уязвимостей, понимают и оценивают риски, которые эти уязвимости несут организации, и принимают необходимые меры для их устранения.

Группа безопасности обязана взаимодействовать с разработчиками, эксплуатационниками, уполномоченным по соответствию нормативным требованиям и руководством, чтобы всё это делалось, дабы управление уязвимостями всегда оставалось в фокусе внимания.

В этой главе мы рассмотрим вопрос об управлении уязвимостями и о том, как оно сочетается с гибкими методиками. Мы также поговорим о бумагах, необходимых для того, чтобы без особого труда и эффективно «прикрыть задницу» при проверках соответствия нормативным требованиям.

Сканирование на уязвимости и применение исправлений

Правильная настройка и задание расписания сканирования на уязвимости, конфигурирование и непреложное соблюдение политик скани-

рования, анализ результатов и их сортировка с учетом риска, подготовка исправлений и тестирование того, что они не вносят новых ошибок, развертывание обновлений по расписанию и учет всех этих действий, позволяющий доказать, что они действительно выполняются, – это составляющие эксплуатации, имеющие огромное значение для организации любого размера.

Чтобы сделать эту работу безопаснее и дешевле, пригодятся методы и средства, обсуждаемые в этой книге, в том числе программное управление автоматизацией настройки и автоматизация сборки, тестирования и поставки.

Сначала поймите, что сканировать

Первый шаг к пониманию причин уязвимостей и управлению ими – составить реестр всех подлежащих защите систем и приложений, размещенных как локально, так и в облаке. В большинстве организаций составить точный и актуальный список всего подлежащего сканированию, трудно, а в масштабе предприятия может оказаться даже невозможным, ведь на небольшой группе безопасности может лежать ответственность за тысячи приложений в нескольких ЦОДах.

Одно из многих преимуществ, которые дают автоматизированные средства управления конфигурацией типа Ansible, Chef и Puppet, – тот факт, что в их основе лежит центральный репозиторий, в котором описаны все серверы, их настройки и установленные на них пакеты.

UpGuard: непрерывная оценка уязвимостей

Система UpGuard (*https://www.upguard.com/*) автоматически отыскивает информацию о конфигурации серверов Linux и Windows, сетевых устройств и облачных служб, выявляет уязвимости и несогласованности и ведет учет изменений со временем.

Она постоянно оценивает риски, связанные с уязвимостями, автоматически сканирует системы самостоятельно или использует информацию от других сканеров и назначает каждой системе оценку соответствия.

UpGuard также создает тесты, призванные навязать политики конфигурации, и умеет генерировать готовый код применения обновлений и изменений конфигурации, пригодный для выполнения инструментами Ansible, Chef, Puppet, Microsoft Windows PowerShell DSC и Docker.

Этой информацией можно воспользоваться, чтобы лучше понять системы, подлежащие сканированию на уязвимости, и выявить среди них те, к которым необходимо применить исправления, – получив тревожное уведомление. После этого те же самые инструменты можно использовать, чтобы быстро и автоматически применить исправления.

Затем решите, как сканировать и с какой частотой

Большинство программ управления уязвимостями работает в реактивном цикле сканирования и исправления, который запускается по расписанию, как, например, в программе Microsoft «Patch Tuesday» (исправления по вторникам).

1. Настройте сканирование серверов и сетевых устройств с помощью таких инструментов, как Core Impact, Nessus, Nexpose или OpenVAS, или онлайновых служб типа Qualys. Эти сканеры ищут известные уязвимости в распространенных дистрибутивах ОС, сетевых устройствах, базах данных и других работающих программах, в частности устаревшие версии программных пакетов, подразумеваемые по умолчанию учетные данные и другие опасные ошибки конфигурации.
2. Проанализируйте найденные уязвимости, отбросьте дубликаты и ложноположительные тревоги, затем назначьте оставшимся находкам приоритеты, основываясь на риске.
3. Передайте результаты техническому подразделению для немедленного устранения уязвимостей. Для этого может понадобиться скачать и применить исправления, или исправить конфигурацию, или добавить сигнатуры в систему обнаружения и предотвращения вторжения (IDS/IPS) либо WAF/RASP[1] для перехвата и блокировки попыток выполнить эксплойт.
4. Проведите повторное сканирование и убедитесь, что проблемы действительно решены.
5. Зафиксируйте все свои действия для аудиторов.
6. Повторяйте каждый месяц или каждый квартал – как записано в нормативных требованиях.

[1] WAF – Web Application Firewall, брандмауэр на уровне веб-приложения. RASP – Runtime Application Self-Protection, самозащита приложения на этапе выполнения. – *Прим. перев.*

Как следует изменить подход к сканированию в гибких средах и средах DevOps

Если изменения вносятся каждую неделю или несколько раз на дню, то сканировать системы раз в месяц или раз в квартал недостаточно. Этак вы никогда не поспеете.

Необходимо автоматизировать и упростить сканирование, чтобы его можно было запускать гораздо чаще, по возможности ежедневно, в составе сборочного конвейера.

Чтобы создать эффективный цикл обратной связи, необходимо придумать, как автоматически устранять дубликаты и отфильтровывать ложноположительные результаты сканирования, а затем передавать результаты напрямую инструментам, которые команда использует для управления своей работой, будь то система учета ошибок типа Jira, или система управления журналом пожеланий типа Rally или VersionOne, или система канбан типа Trello.

Учет уязвимостей

Как мы увидим ниже, существуют и другие важные способы получить информацию об уязвимостях, помимо сканирования сетевой инфраструктуры, а именно:

- сканирование приложений в поисках типичных ошибок кодирования и уязвимостей времени выполнения с помощью средств автоматизированного тестирования безопасности (automated security testing – AST), которые мы будем рассматривать в главе 11;
- тестирование на проникновение и другие ручные и автоматизированные виды тестирования безопасности;
- программы вознаграждения за найденные уязвимости (см. главу 12);
- разведка угроз безопасности и оповещение производителей;
- инструменты анализа программных компонентов (software component analysis – SCA), которые проверяют наличие известных уязвимостей в компонентах с открытым исходным кодом (мы еще вернемся к этому вопросу ниже в данной главе);
- сканирование работающих контейнеров и образов контейнеров в поисках известных уязвимостей;
- сканирование облачных экземпляров в поисках типичных уязвимостей и небезопасных конфигураций с помощью таких служб, как AWS Inspector (*https://aws.amazon.com/ru/inspector/*);

- ручные инспекции или аудит кода приложения, а также инфраструктурных рецептов и шаблонов;
- извещения об ошибках, полученные от партнеров, пользователей и других заказчиков (конечно, это не самый лучший способ узнавать об уязвимостях в своей системе).

Для получения полной картины риска, которой подвергается безопасность системы, следует консолидировать эту информацию для всех систем, хранить ее и сводить в отчеты. Ее также необходимо оценить, рассортировать по приоритетам и отправить обратно эксплуатационникам и разработчикам в виде, который им понятен и хорошо ложится на схему их работы, чтобы уязвимости были быстро устранены.

Управление уязвимостями

Уязвимости в программе или инфраструктуре – это дефекты требований, проектирования, кодирования или внедрения. К ним следует относиться, как к любым другим дефектам системы: исправить немедленно, или добавить в журнал пожеланий команды и назначить приоритет наравне с другими работами, или игнорировать, потому что команда (включая ее руководство) решила, что проблема не стоит того, чтобы ее исправлять.

Но уязвимости привносят риски, которые технические команды, менеджеры продуктов и другие лица в организации понимают с трудом. Обычно пользователи и члены команды сразу понимают, что имеется функциональная ошибка или эксплуатационная проблема, которую нужно исправить, или что производительность системы никуда не годится. С такими проблемами технари умеют справляться. Уязвимости системы безопасности не столь очевидны.

В главе 7 мы увидим, что для решения вопроса о том, какие дефекты безопасности следует устранять и как быстро, нужно знать ответы на несколько вопросов.

- Каков общий профиль угроз в организации? С какого рода угрозами сталкивается организация, и какие цели преследует противник?
- Каков профиль риска системы (или систем), в которых обнаружена уязвимость?
- Насколько широко распространена уязвимость?
- Легко ли противнику обнаружить и эксплуатировать ее?
- Насколько эффективны существующие меры защиты для отражения атаки?

- Каковы потенциальные технические и коммерческие последствия успешной атаки для организации?
- Можно ли обнаружить идущую атаку?
- Как быстро можно остановить обнаруженную атаку, восстановиться после нее и предотвратить повторение атаки?
- Во что обойдется устранение уязвимости, и насколько вы уверены, что это будет сделано корректно и безопасно?
- Каковы ваши обязательства с точки зрения соответствия нормативным требованиям?

Чтобы понять и оценить эти факторы риска, необходимо смотреть на уязвимости отдельно от остальной работы, возложенной на технические команды. Учет и отчетность об уязвимостях – необходимая составная часть программ контроля, управления рисками и соответствия нормативным требованиям (governance, risk and compliance – GRC), принятых во многих организациях. Кроме того, во многих нормативно-правовых документах, в частности в стандарте PCI DSS, требуется способность продемонстрировать руководству и аудиторам ответственное отношение к идентификации рисков безопасности и управлению ими.

Управление уязвимостями включает большой объем рутинной, но важной учетной работы:

- фиксация всех уязвимостей: когда, где и как обнаружена;
- сканирование или проверка того, где еще может присутствовать уязвимость;
- назначение приоритета исправлению на основе оценки риска;
- назначение ответственного за решение проблемы;
- установление сроков исправления;
- проверка корректности исправления и учет потраченного на это времени;
- отчет обо всем этом, с четким указанием исключений, для аудиторов.

В крупных организациях это работа для сотрудника на полной ставке. Во многих мелких организациях, особенно в стартапах, ей вообще никто не занимается.

Инструменты управления уязвимостями, например bugBlast (*https://buguroo.com/products/bugblast-next-gen-appsec-platform*), Code Dx (*https://codedx.com/*) и ThreadFix (*https://threadfix.it/*) компании Denim Group, консолидируют уязвимости, найденные разными сканерами, а также проблемы, обнаруженные при тестировании проникновения и в ходе ручных инспекций всех имеющихся систем. Они помогают выявлять

риски внутри отдельной системы и на стыке систем, определяя, какие системы или компоненты имеют наибольшее число уязвимостей, какие типы уязвимостей наиболее распространены и как долго уязвимости остаются незакрытыми.

Можно оценивать, какие команды лучше всех справляются с устранением уязвимостей и соответствуют ли команды нормативным требованиям. Можно также оценивать эффективность инструментария, проанализировав, какие инструменты нашли наибольшее количество уязвимостей.

Эти инструменты упрощают работу по отысканию уязвимостей тем, что отфильтровывают дубликаты, найденные разными средствами или на разных прогонах, и дают возможность проанализировать, квалифицировать и назначить приоритеты найденным уязвимостям, до того как требовать от разработчиков их устранения. Большинство инструментов управления уязвимостями имеют интерфейс к популярным средствам разработки, так что открывается возможность автоматически обновлять журнал пожеланий команды.

CVE, CWE, CVSS... да как же все это запомнить!

Оценить, насколько плохо мы справляемся со всем, что связано с ошибками в системе безопасности, можно, взглянув на обширную инфраструктуру и бюрократию, выросшую на почве отчетности, классификации и назначения приоритетов проблемам безопасности. Вместо простого «Глянь-ка, тут ошибка, давай исправляй!» мы наплодили кучу акронимов – CVE, CWE, CVSS, CWSS – для учета всех этих ошибок: почему они произошли, как были обнаружены, насколько важны, как их искать, как узнать, исправлены они уже или нет.

CVE (Common Vulnerabilities and Exposures – типичные уязвимости и риски)

CVE (*https://cve.mitre.org/index.html*) – это перечень, включающий тысячи конкретных уязвимостей, найденных в программах. Его ведет корпорация MITRE в интересах US-CERT и Министерства внутренней безопасности США, всем остальным он также доступен бесплатно.

CWE (Common Weakness Enumeration – бюллетень типичных уязвимостей)

CWE (*https://cwe.mitre.org/*) – более короткий, но все равно длинный список, в котором сделана попытка описать и классифицировать ошибки разработчиков, ставшие причиной уязвимостей, включенных в базу данных CVE. При его создании CVE был проанализирован с целью понять глубинные причины каждой уязвимости.

В CWE включены особо опасные ошибки программирования или проектирования, которые заведомо ведут к допускающей эксплуатацию уязвимости. В Национальной базе данных уязвимостей (National

Vulnerability Database) перечислены различные типы CWE (*https://nvd.nist.gov/vuln/categories#cwes*).

Для каждой записи в CWE приведены объяснение типа ошибки, примеры, методика поиска ошибки при тестировании, инспекциях кода и использовании инструментов статического анализа, а также инструкция по исправлению.

Компании MITRE и SANS составили список 25 самых опасных ошибок, назвав его «25 самых опасных ошибок при разработке программного обеспечения из списка CWE/SANS» (*http://cwe.mitre.org/top25/*).

NVD (National Vulnerability Database – Национальная база данных уязвимостей)

База данных NVD (*https://nvd.nist.gov/home*) спонсируется теми же агентствами правительства США. В ней ведется учет CVE с указанием того, какие версии программ затронуты, кто и когда обнаружил уязвимость, и ссылок на другие ресурсы в помощь заинтересованным лицам.

Системы балльной оценки CVSS и CWSS

CVSS и CWSS – альтернативные системы балльной оценки для определения серьезности CVE или CWE. При оценке сопоставляются возможность эксплуатации (насколько противнику просто найти уязвимость и воспользоваться ей) и потенциальный ущерб, который может нанести противник.

Пример Heartbleed

Рассмотрим пример: уязвимость Heartbleed в комплекте программ OpenSSL. В Национальной базе данных уязвимостей Heartbleed присвоен идентификатор CVE-2014-0160. Хотя это была серьезная и весьма распространенная уязвимость, нанесшая значительный ущерб и получившая широкое освещение в прессе, первоначально по системе CVSS v2 она получила лишь оценку 5.0 по шкале от 1 до 10. В версии модели CVSS v3 оценка Heartbleed была повышена до 7.4, но все равно это бледнеет по сравнению со знаменитой уязвимостью ShellShock, заслужившей максимальные 10 баллов.

Чтобы понять, как возникла эта уязвимость, нужно ознакомиться с информацией о ней в CWE. В CWE Heartbleed находится в разделе CWE-119 «Ошибки работы с буфером: ненадлежащая реализация ограничений на выход за пределы буфера в памяти», а точнее в разделе CWE-126 «Чтение за границей буфера». Это ошибка при проверке выхода за границу, возникающая в программах на C/C++, когда забывают о надлежащем контроле входных данных.

Вся суета возникла из-за мелкой и типичной ошибки кодирования. В последующих главах мы еще будем возвращаться к ошибке Heartbleed, поскольку она хорошо известна, был проведен тщательный анализ того, как и почему она произошла и как можно предотвратить подобные ошибки в будущем – а ведь именно этому предмету и посвящена книга.

Что все это значит для разработчиков?

Вы часто будете встречать ссылки на извещения об уязвимостях из CVE. Именно так работают сканеры инфраструктуры безопасности типа Nessus или OpenVAS, и именно так системы обнаружения и предотвращения вторжений (IDS/IPS), а также брандмауэры уровня приложений типа ModSecurity обнаруживают и блокируют атаки по их сигнатурам. И именно так исследователи в области безопасности классифицируют и включают в отчеты свои находки.

Вам также будут встречаться ссылки на CWE в отчетах об ошибках, которые формируют инструменты статического анализа и другие средства сканирования. Эта информация полезна при сравнении результатов, полученных разными инструментами, и для оценки их эффективности, хотя, к сожалению, на практике все обстоит не так просто, как в теории, – ведь разные инструменты могут ссылаться на различные, хотя и родственные, CWE для одного и того же типа ошибки.

Знать об экосистеме CVE/CWE/CVSS/NVD полезно. Но если вы не занимаетесь созданием инструмента безопасности и не собираетесь выполнять какую-то исследовательскую работу в этой области, то можете ответственно создавать безопасные системы, не слишком заботясь о CVE, CWE и всем остальном – до тех пор, пока кто-нибудь не сообщит о CVE в написанной вами программе или в программе, которой вы пользуетесь.

Как относиться к критическим уязвимостям

Если найдена критическая уязвимость типа Heartbleed или ShellShock, которую нужно устранить быстро, то следует положиться на принятую в организации программу управления уязвимостями, которая должна обеспечить следующие условия:

- вы получите информацию об уязвимости по каналам разведки угроз, например в виде оповещения CERT (*https://www.us-cert.gov/ncas/alerts*), а иногда и из прессы. В случае Heartbleed это было нетрудно, поскольку, подобно вирусу Anna Kournikova или ShellShock, уязвимость получила широкое освещение в популярных СМИ благодаря выразительному названию.
- вы понимаете, в чем состоит ошибка и насколько она серьезна по шкале риска CVE;
- вы уверены, что сможете выявить все системы, которые необходимо проверить на данную уязвимость;

- с помощью сканирования и проверок конфигурационных файлов вы можете проверить, присутствует ли уязвимость, и если да, то где именно;
- вы можете быстро и безопасно применить исправление, или обновить уязвимый программный пакет, или выключить уязвимые функции, или добавить сигнатуру в систему IDS/IPS либо в правила брандмауэра, чтобы заблокировать атаку (в качестве паллиативного решения);
- вы можете удостовериться, что исправление или какие-то иные действия по смягчению риска произведены успешно.

Управление уязвимостями сводит воедино безопасность, соответствие нормативным требованиям, разработку и эксплуатацию во имя защиты систем и организации в целом.

Обеспечение безопасности цепочки поставок программного обеспечения

Важная часть управления уязвимостями состоит в том, чтобы хорошо понимать и обеспечить безопасность цепочки поставок тех компонентов ПО, из которых строятся современные системы. Команды, практикующие гибкие методики и DevOps, широко применяют библиотеки и каркасы с открытым исходным кодом, чтобы сократить время и стоимость разработки. Но эта медаль имеет и обратную сторону: возможность получить в наследство чужие ошибки и уязвимости.

Согласно данным компании Sonatype, которая управляет Центральным репозиторием (Central Repository) – крупнейшим в мире репозиторием ПО с открытым исходным кодом для Java-разработчиков, – от 80 до 90% кода современных приложений взято из библиотек и каркасов с открытым исходным кодом.

И в значительной части этого кода есть серьезные ошибки. В Центральном репозитории находится свыше 1,36 млн компонентов (по состоянию на сентябрь 2015 г.), и каждый день добавляется почти 1500 новых. Более 70 000 компонентов в этом репозитории содержат известные уязвимости. Ежедневно появляются сообщения в среднем о 50 новых критических уязвимостях в ПО с открытым исходным кодом.

В 2015 году Sonatype изучила 31 млрд запросов на скачивание от 106 000 организаций. Обнаружилось, что крупные организации, предоставляющие финансовые услуги, и другие корпоративные предприятия

ежегодно скачивают в среднем свыше 230 000 «программных деталей». Но имейте в виду, что это только компоненты, написанные на Java. А общее количество компонентов, включая пакеты RubyGem, NuGet, образы Docker-контейнеров и прочее добро, гораздо больше. В этих 230 000 скачанных компонентах 1 из 16 запросов приходился на ПО, содержащее хотя бы одну известную уязвимость.

Всего один пример – Sonatype проанализировала скачивание популярной криптографической библиотеки The Legion of the Bouncy Castle. В 2015 году она была скачана 17,4 млн раз. Но в одной трети случаев скачивались версии, содержащие известную уязвимость. Это значит, что почти 14 000 организаций по всему миру без всякой необходимости и, возможно, даже не подозревая, подвергли себя потенциально серьезным рискам (*https://www.sonatype.com/2016-state-of-the-software-supply-chain-report*), пытаясь сделать свои приложения более безопасными.

Еще не страшно? А зря.

Очевидно, команда должна обязательно знать, какие компоненты с открытым исходным кодом включены в ее приложения, следить за тем, чтобы заведомо надежные версии скачивались из заведомо надежных источников, и загружать обновленные версии компонентов после исправления вновь обнаруженных уязвимостей.

OWASP Dependency Check

OWASP Dependency Check – это бесплатный сканер, который каталогизирует все используемые в приложении компоненты с открытым исходным кодом и показывает имеющиеся в них уязвимости. Имеются версии для Java, .NET, Ruby (gemspec), PHP (composer), Node.js и Python, а также для некоторых проектов на C/C++. Dependency Check интегрируется с распространенными средствами сборки, в т. ч. Ant, Maven и Gradle и серверами непрерывной интеграции типа Jenkins.

Dependency Check сообщает обо всех компонентах с известными уязвимостями из Национальной базы данных уязвимостей (NVD) NIST и обновляется на основании данных из новостных каналов NVD.

Перечислим еще несколько популярных инструментов проверки зависимостей с открытым исходным кодом:

- Bundler Audit для Ruby;
- Retire.js для Javascript;
- SafeNuGet для пакетов NuGet.

По счастью, все это можно делать автоматически с помощью таких инструментов, как проект OWASP Dependency, или коммерческих программ типа Black Duck (https://www.blackducksoftware.com/solutions/application-security), JFrog Xray (https://jfrog.com/xray/), Snyk (https://snyk.io/), Nexus Lifecycle (https://www.sonatype.com/nexus-lifecycle) компании Sonatype или SourceClear (https://srcclr.com/).

Эти инструменты можно включить в сборочные конвейеры, чтобы автоматически составлять опись зависимостей с открытым исходным кодом, выявлять устаревшие версии библиотек и библиотеки, содержащие известные уязвимости, и прерывать сборку в случае обнаружения серьезных проблем. Имея для каждой системы актуальную опись зависимостей, вы будете подготовлены к появлению таких уязвимостей, как Heartbleed или DROWN, поскольку сможете быстро определить, находитесь ли под угрозой и что нужно сделать для устранения опасности.

Эти инструменты умеют также отправлять уведомление при обнаружении новых зависимостей, чтобы вы могли запланировать их инспекцию.

Уязвимости в контейнерах

Если пользуетесь в производственной среде (или хотя бы для разработки и тестирования) контейнерами типа Docker, то должны применять аналогичные средства контроля к образам контейнеров. Хотя Docker сканирует образы в официальных репозиториях в поисках пакетов с известными уязвимостями, существует шанс скачать «отравленный образ», содержащий устаревшее или вредоносное ПО или сконфигурированный небезопасным образом.

Вы должны сканировать образы самостоятельно с помощью инструментов с открытым исходным кодом типа OpenSCAP или Clair либо коммерческих служб сканирования от компаний Twistlock, Tenable или Black Duck Hub. А затем поместите эти образы в собственный безопасный репозиторий или частный реестр, откуда их смогут безопасно брать разработчики и эксплуатационники.

Лучше меньше, да лучше

Чрезмерное расширение цепочки поставщиков несет очевидные издержки на сопровождение и риски безопасности. Согласно модели бережливого производства компании Toyota, нашей стратегической целью должно быть «меньше поставщиков, но хороших» – ограниче-

ние списка библиотек, каркасов, шаблонов и образов теми, которые заведомо работают, решают важные для разработчиков задачи и тщательно проверены группой безопасности. В Netflix это называют строительством асфальтированной дороги, поскольку разработчики (а также безопасники и уполномоченные по соответствию нормативным требованиям) знают, что если будут пользоваться этим кодом, то дорога станет легче и безопаснее.

Оценка стоимости и рисков цепочки поставщиков

Компания Sonatype разработала бесплатный калькулятор, помогающий разработчикам и руководству оценить стоимость и риски, которые со временем накапливаются в результате использования слишком большого числа сторонних компонентов.

Но нужно понимать, что хотя в долгосрочной перспективе это имеет смысл, заставить различные инженерные команды пользоваться стандартизованным набором общих компонентов будет нелегко. Трудно просить разработчиков, поддерживающих унаследованные приложения, тратить силы на такого рода изменения. И не менее трудно в микросервисных средах, когда ожидается, что разработчики могут выбирать любые средства для выполнения работы, ориентируясь на технологии, отвечающие конкретным требованиям, и даже на личные пристрастия.

Разумно начать со стандартизации самых нижних уровней ПО: ядра, ОС и виртуальных машин, а также служебных функций общего назначения, например протоколирования и сбора показателей, которые нужны во всех приложениях и службах.

Как устранить уязвимости по-гибкому

Основная проблема, знакомая почти всем организациям, заключается в том, что, даже зная о существовании серьезной уязвимости в системе, они не могут исправить ее достаточно быстро, чтобы помешать эксплуатации ее противником. Чем дольше существует уязвимость, тем больше шансов, что система будет или уже была атакована.

Компания WhiteHat Security, предоставляющая услугу по сканированию сайтов на уязвимости, регулярно анализирует и публикует собранные данные об уязвимостях. По данным за 2013 и 2014 год 35% сайтов

финансовых и страховых компаний «уязвимы всегда», т. е. хотя бы одна серьезная уязвимость присутствовала на них каждый день в году. Статистика для других отраслей и государственных учреждений еще хуже. Лишь 25% сайтов финансовых и страховых компаний были уязвимы менее 30 дней в году.

В среднем серьезные уязвимости оставались открытыми в течение 739 дней, и только 27% серьезных уязвимостей вообще устранялись из-за затрат, рисков и издержек, сопряженных с выпуском исправлений[2].

Есть много причин, по которым устранение уязвимостей занимает слишком много времени, – не только занятость команды работой над новыми функциями:

- время, затрачиваемое на бюрократические процедуры и оформление бумаг при передаче работы между группой безопасности и техническими группами;
- технические группы не понимают, насколько серьезны упомянутые в отчетах уязвимости и как их исправить;
- команда боится, что латание приведет к ошибке и выходу системы из строя, поскольку не уверена в своем умении собрать, протестировать и развернуть обновленное ПО;
- управление изменениями – затратная и медленная процедура, принимая во внимание все шаги по сборке, инспекции, тестированию и развертыванию, а также обязательные процедуры передачи и утверждения.

Как мы уже видели в этой книге, сама скорость гибкой разработки создает новые риски и проблемы в части безопасности. Но те же скорость и эффективность могут стать щитом против атакующих, способом быстрее закрыть окно уязвимости.

Гибкие команды создаются, чтобы оперативно реагировать на новые приоритеты и пожелания, будь то новая функциональность или проблема в производственной системе, нуждающаяся в срочном исправлении. Оперативное назначение приоритетов, инкрементное проектирование и быстрая поставка, автоматизация сборки и тестирования, измерение и оптимизация цикла – всё это способствует удешевлению, ускорению и упрощению процесса внесения изменений.

---

[2] См. Отчет WhiteHat Security «2017 Application Security Statistics Report: The Case for DevSecOps» по адресу *https://www.whitehatsec.com/resources-category/premium-content/web-application-stats-report-2017/*.

Назначение приоритетов уязвимостям

При назначении приоритета исправлению уязвимостей следует учитывать несколько факторов.

Серьезность риска

На основе балльной оценки, например CVSS.

Возможность эксплуатации

Предложенная командой оценка вероятности эксплуатации уязвимости в конкретной среде, широты распространения уязвимости, а также наличия компенсирующих средств контроля.

Стоимость и риск исправления

Объем работы, необходимой для устранения и тестирования уязвимости, может варьироваться в широких пределах: от развертывания узконаправленного исправления от поставщика или небольших технических действий по изменению списков контроля доступа (ACL) или конфигурационного параметра по умолчанию до полного обновления платформы или переработки логики приложения.

Соответствие нормативным требованиям

Стоимость и риск нарушить нормативные требования.

Практика и инструментарий DevOps, в частности автоматизированное управление конфигурацией, непрерывная поставка и повторяемое автоматическое развертывание делают получение изменений и внесение их в производственную систему еще дешевле, быстрее и безопаснее. DevOps как раз и стоит на способности минимизировать среднее время восстановления после эксплуатационных инцидентов, поскольку есть уверенность, что на исправление много времени не понадобится.

Рассмотрим, как снизить риски безопасности, воспользовавшись гибкими практиками, инструментами и циклом обратной связи, оптимизированными для достижения максимальной скорости и эффективности.

Безопасность через тестирование

Один из способов убедиться в том, что уязвимости приложения устранены, – написать автоматизированный тест (автономный или приемочный), который доказывает существование уязвимости, и прогонять его вместе с остальными тестами в процессе сборки системы.

После устранения уязвимости тест не должен пройти. Это похоже на то, как программисты, практикующие разработку через тестирование, проверяют, что ошибка исправлена (см. главу 11).

Проверка наличия Heartbleed с помощью Gauntlt

В главе 11 мы покажем, как писать тесты безопасности с помощью каркаса Gauntlt. В комплект поставки Gauntlt входят примеры атак, в т. ч. тест, специально предназначенный для проверки уязвимости Heartbleed. Его можно взять в качестве образца при написании собственных проверок на другие уязвимости.

Разумеется, чтобы команда приняла такой подход, человек, пишущий тест, должен быть членом команды. Ему необходима поддержка со стороны владельца продукта, который отвечает за расстановку приоритетов и вряд ли будет рад, если команда начнет отвлекаться на исправление чего-то такого, важность чего он не понимает.

Этот человек должен также понимать и уважать и принятые командой соглашения о структурировании кода и комплекта тестов и обладать техническими навыками написания хороших тестов. Тест должен недвусмысленно показывать, что существует реальная проблема. Он должен быть согласован с принятыми всей командой подходами, чтобы команда согласилась принять владение этим тестом. Для всего этого, вероятно, потребуется помощь со стороны какого-нибудь члена команды, и было бы гораздо проще, если бы команда уже приложила значительные усилия к автоматизации тестов.

Написание подобных тестов дает уверенность, что уязвимость исправлена надлежащим образом. И страхует от ее повторного появления. Как мы видели, это большой шаг в правильном направлении, по сравнению с оставлением отчета об уязвимости на столе разработчика.

Нулевая терпимость к дефектам

Некоторые гибкие команды стремятся к идеалу «нулевой терпимости к дефектам». Они настаивают на том, чтобы любой найденный дефект исправлялся, прежде чем функцию можно будет считать готовой и переходить к следующей функции или истории. Если проблема достаточно серьезна, то команда может прекратить все остальные работы и бросить все силы на ее исправление.

Если вы сможете объяснить такой команде, что уязвимость – это дефект, самый настоящий дефект, который необходимо исправить, то она будет обязана устранить ее.

Чтобы команда восприняла ваши слова серьезно, нужно сделать несколько вещей.

1. Безжалостно отсеять ложноположительные результаты и сосредоточиться на уязвимостях, которые действительно важны для организации, – серьезных и допускающих эксплуатацию проблемах.
2. Поместить их в журнал пожеланий в форме, понятной команде.
3. Потратить время на объяснение команде, включая и владельца продукта, что это за дефекты и почему они важны.
4. Потратить дополнительное время, чтобы объяснить команде, как тестировать и исправлять каждый дефект.

Этот подход работоспособен, если вы можете приступить к работе с командой на ранних стадиях разработки, чтобы разбираться с уязвимостями по мере появления. Было бы нечестно по отношению к команде или организации спустя много месяцев или лет успешной эксплуатации выкатывать длинный список уязвимостей, найденных при последнем сканировании, и ожидать, что команда всё бросит и сразу же займется их устранением. Но можно поговорить с командой и составить план, в котором будет найден баланс между рисками безопасности и другой работой.

Коллективное владение кодом

Еще одна распространенная в гибких командах идея состоит в том, что код открыт для всех членов команды. Любой может проинспектировать код, написанный кем угодно, подвергнуть его рефакторингу, добавить тесты, внести исправления или изменения.

Это означает, что если инженер по безопасности нашел уязвимость, то у него должна быть возможность исправить ее, коль скоро он считается членом команды. Например, в Google большая часть кодовой базы открыта всем сотрудникам, а это означает, что инженеры по безопасности могут устранять уязвимости в любой части кода, при условии что будут следовать принятым в команде соглашениям и примут ответственность за проблемы, которые могут в результате возникнуть.

Для этого требуются глубокие технические знания (для инженеров в Google это, конечно, не проблема, но в вашей организации дело может обстоять не так радужно) и уверенность в своих возможностях. Но если вы знаете, что дефект серьезный, и знаете, в каком месте кода он нахо-

дится и как его исправить, то не будет ли правильно просто сделать это, а не пытаться убедить кого-то еще, что тот должен оставить все свои дела и исправить дефект вместо вас?

Даже если вам не хватает уверенности в собственном умении писать код, запрос на включение кода или просто пометка кода для инспекции может приблизить решение проблемы.

Спринты безопасности, спринты укрепления и хакатоны

Еще один способ способствовать устранению проблем с безопасностью, особенно если их много, как бывает, например, после тестирования на проникновение, аудиторской проверки или реакции на взлом, – организовать «спринт безопасности» или «спринт укрепления».

В гибких командах под «укреплением» (hardening) понимается всё необходимое для того, чтобы подготовить систему к передаче в эксплуатацию. На этой стадии команда перестает думать о поставке новых функций и посвящает большую часть времени упаковке, развертыванию, установке и настройке системы, дабы убедиться, что она готова к работе. Если команда практикует непрерывную поставку или непрерывное развертывание, то ко всему этому она готова при каждой фиксации изменения.

Но для многих других команд неприятным и дорогостоящим сюрпризом становится тот факт, что они должны взять функциональный прототип, прекрасно работающий в среде разработки, и превратить его в систему промышленного качества, готовую к работе в реальном мире, и, в частности, обеспечить надежность и безопасность.

В спринте укрепления команда прекращает работу над новыми функциями и расширением архитектуры, а все время посвящает приближению системы к выпуску.

Есть две принципиально различные категории людей. Одни понимают, что выделение времени на укрепление иногда необходимо, особенно в крупных проектах, где требуется прорабатывать вопросы интеграции. Другие непоколебимо уверены, что резервирование времени для укрепления – признак того, что что-то – или вообще всё – делается неправильно и что команда не справляется или уже потерпела крах. Особенно это относится к случаю, когда под «укреплением» команда понимает отдельный спринт (или несколько спринтов) для тестирования и исправления дефектов – вещь, которую нужно было делать в процессе

написания кода. Такой подход получил название «Agilefall» – объединение слов «agile» (гибкая модель) и «waterfall» (каскадная модель, или модель водопада), т. е. оксюморон.

Спринты укрепления встроены в SAFe (Scaled Agile Framework) – каркас для управления большими корпоративными гибкими проектами. В SAFe зарезервировано время для работ, которые можно выполнить только на этапе окончательного укрепления и упаковки, перед тем как выпустить большую систему. Сюда входят в том числе проверки на безопасность и соответствие нормативным требованиям. Еще один метод гибкой разработки корпоративных систем, DAD (Disciplined Agile Delivery, дисциплинированная гибкая поставка), был предложен Скоттом Эмблером (Scott Ambler) из IBM для масштабирования гибких методик на крупные проекты; он также включает этап укрепления перед каждым релизом.

Попробуйте организовать хакатон вместо спринта укрепления

Некоторые организации регулярно проводят хакатоны, на которых команды отвлекаются от плановой работы, чтобы изучить что-то новое, создать или усовершенствовать инструменты, опробовать новые идеи и вместе порешать задачки.

Некоторые команды успешно заменили спринты укрепления хакатонами с упором на безопасность. Хакатон, который нередко заканчивается поздно ночью, сопровождается помощью экспертов и направлен на поиск и устранение уязвимости определенного вида. Например, можно собрать всю команду и рассказать ей об уязвимостях из-за внедрения SQL: как их искать и правильно устранять. Затем все вместе, часто разбившись на пары, работают над устранением максимально большого числа таких уязвимостей в коде.

Такие хакатоны, очевидно, гораздо дешевле и проще организовать, чем отдельный спринт. К тому же они безопаснее: менее вероятно, что разработчики допустят ошибки, если всем сказано, что нужно делать, и все вместе работают над одной и той же задачей в течение короткого времени. Хакатоны проливают свет на риски безопасности, помогают обучить команду (потратив несколько часов на устранение конкретной уязвимости, в будущем любой сможет быстро найти и исправить подобные) и способствуют устранению важных дефектов, не слишком замедляя работу команды.

Запланировать спринт безопасности на этапе укрепления – действие, которое вам, возможно, придется сделать один, а то и несколько раз на

протяжении работы над проектом, особенно если вы работаете с унаследованной системой, в которой накопился большой технический долг, в т. ч. и в части безопасности. Но спринты укрепления дороги, и их необходимость трудно объяснить заказчику и руководству, которые, естественно, хотят знать, почему скорость команды упала до нуля и как она могла оказаться в таком плачевном положении.

Полагаться на то, что в ходе отдельного спринта укрепления удастся найти и исправить уязвимости и прочие ошибки прямо перед выпуском релиза, рискованно, а в длительной перспективе это означает ввязывание в заведомо проигранную битву. Попытка заставить все команды прекратить новые разработки и заниматься исключительно вопросами безопасности в течение нескольких недель или месяцев кряду была ранней и предпринятой от отчаяния частью инициативы Microsoft Trustworthy Computing Initiative. Но очень скоро Microsoft осознала, что это слишком накладно и при этом не приносит длительного улучшения надежности и безопасности ПО. Тогда компания пошла по другому пути – встраивать методы обеспечения безопасности непосредственно в жизненный цикл разработки.

Долг безопасности и его оплата

Гибкие команды научились распознавать технический долг и бороться с ним. Технический долг – это совокупность всего того, что команда или ее предшественники должны были бы сделать в процессе создания системы, но не сделали из-за нехватки времени или по незнанию. Сюда относится срезание углов, написанные тяп-ляп куски, тесты, которые надо было бы написать, да не написали, или написали, но они не прошли и их оставили в таком состоянии, ошибки, которые следовало бы исправить, код, не подвергнутый рефакторингу, несмотря на очевидную необходимость, заплаты, которые так и не наложили.

Всё это со временем накапливается, в результате чего система становится более хрупкой, менее надежной и безопасной, ее все труднее изменять. В конце концов, какую-то часть этого долга приходится оплачивать, да еще с процентами, и обычно взимается он с людей, которые не присутствовали в момент образования долга: как дети платят по закладным, взятым родителями.

Если команда или ее руководство отдает приоритет быстрой поставке новой функциональности, не заботясь о написании безопасного кода, не инвестирует в обучение, откладывает на потом инспекции и тестирование, не уделяет внимания проблемам безопасности и не вы-

деляет времени на их своевременное исправление, то накапливается долг безопасности. Этот долг повышает риск компрометации системы, а также стоимость устранения проблем, поскольку не исключено, что части системы придется перепроектировать и переписать.

Для некоторых организаций такой подход может оказаться правильным. Например, стартапы, исповедующие бережливое программирование, и другие команды, создающие минимально жизнеспособный продукт (MVP), должны урезать требования до абсолютного минимума и поставить систему как можно скорее, чтобы увидеть, как она работает, и получить отзывы. Было бы пустой тратой времени и денег писать на этой стадии солидный, безопасный код, выполнять все инспекции и тестирование, пытаясь гарантировать его правильность. Ведь очень может статься, что через несколько дней или недель код придется выбросить и начать все заново, или собрать вещи и заняться другим проектом, или искать другую работу, потому что система оказалась никуда не годной или кончились деньги.

Бывает и так, что люди, которые оплачивают или реально выполняют работу, вынуждены срезать углы, чтобы уложиться в жесткие сроки – когда сделать хоть что-то прямо сейчас важнее, чем сделать правильно, но позже.

Главное – чтобы все участники (те, кто создает систему, те, кто оплачивает эту работу, и те, кто пользуется системой для своих целей) ясно осознавали, что такие решения сопряжены с риском, и готовы были с этим риском смириться.

Именно так Microsoft, Facebook и Twitter завоевывали рынок. Такая стратегия высокого риска и высокого вознаграждения может окупиться и в приведенных примерах окупилась-таки, но в конечном итоге все эти организации были вынуждены столкнуться с последствиями ранее принятых решений и потратить массу времени, инвестировать огромные деньги и привлечь множество талантливых людей для оплаты своих долгов. Возможно, им так никогда и не удастся справиться с этим: Microsoft борется с серьезными проблемами безопасности с тех пор, как Билл Гейтс провозгласил «безопасные вычисления» наивысшим приоритетом компании еще в 2002 году.

А все потому, что долг безопасности, как и долг по кредиту, обрастает процентами. Небольшой долг, взятый на короткое время, оплатить легко. Но куча долгов, оставшихся с давних пор, могут привести систему – и организацию – к банкротству.

Следите за накапливающимся долгом безопасности и старайтесь делать выбор обдуманно и открыто. Долг безопасности и другие техничес-

кие долги, в т. ч. перечисленные ниже, должны быть понятны владельцам системы и одобрены ими:

- открытые уязвимости и не примененные исправления (т. е. величина и продолжительность окна уязвимости);
- результаты тестирования на проникновение и аудиторские проверки, по которым не приняты меры;
- высокорисковые участки кода с малым покрытием автоматизированными тестами;
- пробелы при сканировании и инспекциях;
- пробелы в обучении;
- показатели «время обнаружения дефекта» и «время исправления дефекта» (т. е. насколько оперативно команда выявляет аварийные ситуации и реагирует на них).

Пишите истории, которые объясняют, что делать для погашения долга, и помещайте их в журнал пожеланий, чтобы им можно было назначить приоритеты наряду с другими работами. И следите за тем, чтобы все участники осознавали риски и затраты, возникающие, когда что-то делается неправильно прямо сейчас, – исправление в будущем обойдется дороже.

Сухой остаток

Перечислим несколько фактов и рекомендаций, о которых следует помнить для эффективной борьбы с уязвимостями.

- Новые уязвимости ПО обнаруживаются ежедневно. Оценка и управление уязвимостями – непрерывная деятельность.
- Применяйте инструменты и API, чтобы перенести сведения об уязвимостях из отчетов в журнал пожеланий команды, чтобы работу по их устранению можно было запланировать, как любую другую.
- Делайте все, чтобы команда, а особенно владелец продукта и Scrum-мастер, понимали, что такое уязвимости, а также почему и как их следует устранять.
- Следите за уязвимостями в сторонних продуктах, в т. ч. библиотеках и каркасах с открытым исходным кодом, исполняющих средах и образах контейнеров. Сканируйте зависимости на этапе сборки и останавливайте сборку при обнаружении серьезных уязвимостей. Помещайте безопасные зависимости в локальные

репозитории или частные реестры образов, откуда их могут брать разработчики.

- Автоматизированное управление конфигурацией и конвейеры непрерывной поставки помогают быстро и уверенно реагировать на серьезные уязвимости. Уверенность в том, что вы сможете быстро написать и развернуть исправление программы, – важный шаг на пути борьбы с уязвимостями.
- Хакатоны могут стать эффективным средством привлечь внимание команды к уязвимостям и методам их устранения.

Глава 7

Риск для гибких команд

Для профессионалов в области безопасности управление рисками – хлеб насущный. Но разработчики, особенно гибкие команды, могут проводить свои дни в счастливом неведении о рисках.

Посмотрим, что нужно, чтобы свести эти два мира – или два взгляда на мир – воедино.

Безопасники говорят «нет»

Прежде чем приступить к разговору о том, как осуществляется управление рисками, давайте совершим небольшой экскурс в назначение управления рисками и безопасности вообще.

Во многих организациях безопасники снискали репутацию людей, всегда говорящих «нет». Работающая над проектом команда готова поставить новую функцию, но при этом использует подход или технологию, которую безопасники не понимают, а потому и не разрешают. Эксплуатационники хотят внести в конфигурацию брандмауэра изменение для поддержки новой системы, но брандмауэр – это вотчина безопасников, а они не успевают согласовать изменение вовремя, и вот – внедрение системы откладывается.

Все это делается во имя управления рисками. Смысл управления рисками в том, чтобы перечислить и количественно оценить неизвестное в стремлении контролировать риск. Самый простой способ контроля неизвестности и рисков – запретить любые изменения, тогда точно ничего не сломается. Но такой подход к делу упускает из виду главное, а если он практикуется в динамичной среде, то ведет ко множеству негативных побочных эффектов, влияющих в том числе и на безопасность в целом.

Группа безопасности должна делать так, чтобы организация достигала своих целей самым безопасным и защищенным способом. Это

значит, что эффективный процесс управления рисками не должен запрещать людям рисковать – но их следует информировать. Здесь *информировать* – ключевое слово: управление рисками – это не о запретах, а об осознанном понимании, снижении, общей ответственности и готовности к необходимому риску.

Вместо категоричного «нет» группа безопасности должна говорить «да, но», а еще лучше – «да, и» и предлагать свое руководство и помощь в выполнении действий самым безопасным из возможных способов.

А теперь, разобравшись с этим, познакомимся с типичными подходами к управлению рисками.

Осознание рисков и управление рисками

Управление рисками – центральная тема безопасности и соответствия нормативным требованиям, смысл его в том, чтобы защитить системы перед лицом атак.

В области безопасности за последние несколько десятилетий было предпринято немало усилий, чтобы определить и стандартизировать методы осмысления и контроля рисков. Такие стандарты, как PCI DSS и SOX, а также системы организации управления и контроля типа NIST SP 800-53 и COBIT подразумевают, что организация будет применять осознанный и проверенный подход к управлению рисками, например:

- ISO 27005;
- NIST SP 800-30/39;
- OCTAVE;
- FAIR;
- AS/NZS 4360.

Эти методологии управления рисками имеют дело с недостоверностью. Любой подход к управлению рисками начинается с того, чтобы идентифицировать, оценить и ранжировать риски, стремясь добиться некоторого контроля над недостоверностью.

При оценивании рисков принимают во внимание вероятность некоторого неблагоприятного события и потенциальный ущерб для организации, если это событие произойдет. Для этого каждому риску сопоставляют числовое значение (количественную оценку) или относительный приоритет по шкале высокий–средний–низкий (качественную оценку).

В международном стандарте оценки рисков ISO 31010 описан 31 метод оценивания – на выбор. Формальные методы оценки рисков помогают сделать трудный выбор между несколькими вариантами и ответить на

вопросы типа: «Сколько денег следует потратить на попытку защитить системы, данные, организацию и клиентов от возможных проблем?»

Но даже при использовании формальных методов оценить риски, свойственные программному обеспечению, может оказаться трудно. Каков ущерб от успешной эксплуатации уязвимости типа XSS (межсайтовый скриптинг) в одном из веб-приложений? И каковы шансы, что это случится? Какими критериями должен руководствоваться владелец продукта или руководитель программы, решая, каким рискам уделить внимание и в каком порядке?

Нормативные требования часто диктуют, как именно принимать такие решения. Например, в стандарте PCI DSS ясно указано, что все уязвимости с определенной оценкой риска должны устраняться не позднее определенного времени. Такие средства, как список 10 главных рисков, составленный OWASP, также могут оказать помощь, когда нужно оценить различные риски безопасности и решить, как к ним относиться.

Список 10 главных рисков OWASP

Сообщество OWASP составило список 10 главных рисков, включив в него самые распространенные и самые опасные риски, встречающиеся в веб-приложениях. К ним относятся внедрение, некорректная аутентификация и управление сеансами и межсайтовый скриптинг. Для каждого риска рассмотрены следующие элементы.

Злоумышленники

Пользователей какого типа следует опасаться.

Векторы и сценарии атак

Атак какого типа ожидать.

Пригодность для эксплуатации

Насколько просто противнику эксплуатировать уязвимость и как широко она распространена.

Простота обнаружения

Насколько просто противнику обнаружить уязвимость.

Последствия

Различные последствия: технические, для приложения, для бизнеса.

Эта информация – первая и важная остановка на пути к лучшему осмыслению рисков. Она поможет понять, на что в первую очередь обращать внимание при обучении, проектировании, кодировании, инспекциях и тестировании.

Управление рисками – не застывший свод правил: оно изменяется в ответ на уровень изощренности угроз, изменение нормативно-правовой базы и изменения, вносимые в систему и процедуры эксплуатации. Поэтому такие нормативные документы, как стандарт PCI DSS, настаивают на регулярной переоценке рисков.

Риски и угрозы

Понятия риска и угрозы были введены в этой книге ранее, а подробнее об угрозах мы будем говорить в другой главе. Здесь же мы хотим провести различие между угрозами и рисками и объяснить их взаимное влияние.

Угрозы

Люди и вещи, от которых вы должны защитить свою систему, данные и клиентов; всё, что может пойти не так, как ожидается, и ущерб, который может быть причинен системе или данным вследствие этого. Угрозы всегда конкретны.

Риски

Степень открытости системы для угроз (вероятность и материальные потери), что вы можете и должны сделать для уменьшения открытости и стоимость компромиссов. Риски абстрактны.

Угрозы и риски безопасности есть повсюду, надо только знать, где искать.

Если вашу отрасль активно атакуют, если в вашей организации хранится много ценных персональных или финансовых данных, или если сканирование показывает, что ваше приложение изъязвлено уязвимостями типа внедрения SQL или межсайтового скриптинга, то вы в большой беде и должны что-то предпринять.

Но если ваша организация может стать жертвой атаки, а вы не знаете, какие у вас есть уязвимости и насколько они серьезны, то вы в еще большей беде. И от того, что вы спрячете голову в песок, риски и угрозы никуда не денутся.

Сегодня каждый подвергается самым разным угрозам – независимо от отрасли промышленности, независимо от территориального расположения. Не стоит считать, что можно не заботиться о безопасности, в т. ч. данных или программ, потому что организация маленькая или вообще только что образовавшийся стартап или потому что вы работаете в унаследованной среде. Все это уже не повод для беспечности.

Для начала нужно понять, что угрожает вашей организации и системе и насколько эти угрозы серьезны. Это предмет разведки и оценки угроз – темы главы 8.

Затем нужно изучить, насколько хорошо ваша система и организация подготовлена к встрече с этими угрозами и что нужно сделать, чтобы уменьшить риски. Это и есть тема данной главы.

Регулирующим органам, вашему руководству, заказчикам и другим заинтересованным сторонам важно, чтобы у вас имелся серьезный и последовательный подход к выявлению и оценке рисков, способных повлиять на безопасность ваших систем, информированию о них и противодействию им.

Отношение к риску

Существуют различные стратегии борьбы с рисками.

Снижение

Внедрение контрмер, компенсирующих средств (аудит операций, система предотвращения вторжений или брандмауэры на уровне приложений), планов по управлению рисками, инструментария, обучения, тестирования и сканирования.

Избежание

Решение о том, чтобы не делать чего-то, отключение или упрощение некоторой функции или интерфейса, отказ от использования небезопасных или непроверенных технологий.

Готовность идти на риск

Ясное сознание того, что беда с большой вероятностью случится, и готовность встретить ее: мониторинг, реакция на инциденты и непрерывная поставка (доказанная способность быстро развертывать исправления).

Разделение или передача ответственности

Передача ответственности третьим сторонам, например операторам ЦОДов, поставщикам облачных служб или поставщикам услуг по управлению безопасностью (managed security service providers – MSSP), чтобы разделить с ними риск. Сюда же относится страхование.

Главное – помнить, что любая стратегия смягчения или контроля рисков обходится не бесплатно. Это могут быть финансовые затраты, влияние на производительность, задержка выхода на рынок или потеря

способности к быстрым изменениям либо удобства использования. Мы должны быть уверены, что выбранный подход не противоречит нуждам и приоритетам организации и что в результате принятых мер организация не потеряет ту самую ценность, которую мы пытаемся защитить.

Убедиться, что вам удалось в какой-то мере смягчить риски, позволит тестирование системы на уязвимости и другие предполагаемые события. Традиционно команды разработчиков ПО делают это в момент выпуска релиза, применяя средства тестирования на проникновение или еще какие-то инструменты проверки безопасности.

Но оставлять эти проверки на самый конец – значит заставлять ответственных за риск выбирать между неадекватным с точки зрения безопасности решением и срывом сроков. Именно поэтому слово «безопасность» стало ругательством в самых разных кругах, а мир изобилует непригодными и небезопасными системами.

Современные гибкие команды исповедуют другой подход – они интегрируют управление рисками прямо в процессы проектирования, разработки, тестирования и развертывания и применяют автоматизацию, итеративную разработку и циклы обратной связи для поддержания высокой степени уверенности в условиях быстрой поставки. Все это они должны делать так, чтобы не снижать скорости и не увеличивать без необходимости затрат.

Делать риски видимыми

Разумеется, невозможно смягчить все риски. Нужно отчетливо понимать, какие риски (из тех, о которых вам известно) еще не рассмотрены или рассмотрены лишь частично.

В традиционных программных проектах для этого служит реестр рисков, за который отвечает руководитель проекта. В нем отмечаются «остаточные риски», т. е. риски, оставшиеся в системе после развертывания.

Современные команды могут учитывать риски несколькими способами.

- Непрерывное сканирование инфраструктуры и коды, включая библиотеки и каркасы с открытым исходным кодом, на предмет устаревших версий пакетов и известных уязвимостей.
- Учет уязвимостей и показателей, характеризующих управление ими (см. главу 6).
- Измерение покрытия автоматизированными тестами и сложности кода в высокорисковых участках программы.

- Отслеживание еще не реализованных историй, касающихся безопасности, и историй противника в журнале пожеланий.
- Измерение сроков изменений и среднего времени восстановления для проблем с системами, запущенными в производство. Это показатель того, как быстро команда способна реагировать на серьезную уязвимость или взлом.
- Учет технического долга в журнале пожеланий в виде историй либо использование платформ автоматизированного анализа кода типа SonarQube (*https://www.sonarqube.org/*) или Code Climate (*https://codeclimate.com/*).

Некоторые гибкие команды ведут также реестр рисков в виде журнала рисков или стены рисков, на которую вывешиваются выявленные риски вместе с оценками и замечаниями о том, что команда собирается с ними делать. Этот журнал рисков управляется так же, как журнал пожеланий команды: каждому пункту назначаются приоритет и сроки, зависящие от приоритета и затрат.

Ход работ команды над журналом рисков можно измерять так же, как скорость поставки историй, а отчеты формировать, например, с помощью диаграмм сгорания рисков[1].

Принятие и передача рисков

Некоторыми рисками вы не можете управлять эффективно, по крайней мере в данный момент, а быть может, вы решили, что не стоит и пытаться их минимизировать, т. к. не верите, что они могут реализоваться, или потому что их последствия настолько незначительны, что не могут нанести организации серьезного материального ущерба.

В таких случаях есть два варианта действий.

1. Кто-то в организации, наделенный достаточными полномочиями, может взять на себя последствия риска, если он когда-нибудь реализуется. В гибких командах такое решение может принять владелец продукта или команда в целом при участии группы безопасности или руководства либо самостоятельно.
   Важно, чтобы лицо, принимающее такое решение, обладало достаточной информацией, чтобы подойти ответственно. И важно понимать, что принятие риска не означает, что он перестал существовать. Всякий раз, принимая риск, вы влезаете в некий долг – как в случае финансового или технического долга. Не забывайте, что на долг набегают проценты.

[1] См. статью Mike Cohn «Managing Risk on Agile Projects with the Risk Burndown Chart», 8 апреля 2010.

Вместе со своим руководством вы должны быть готовы, что когда-нибудь этот долг, возможно, придется оплатить. Один из способов подготовиться к этому – загодя создать хорошую систему реагирования на инциденты и испытать ее, чтобы иметь уверенность в своей способности справиться с проблемами, если они возникнут. Мы еще вернемся к этой теме в главе 13.

2. Можно также передать риск полностью или частично третьей стороне, отдав ответственность на аутсорсинг другой организации, которая лучше подготовлена к такому риску (например, передать управление своим ЦОДом поставщику облачных служб с хорошей репутацией), или застраховав часть риска.
   Но страхование – это не карточка «освобождение из тюрьмы» в игре «Монополия». Страховщики захотят убедиться, что вы ответственно подошли к снижению своего и их риска до приемлемого уровня. И на основе результатов своего анализа они будут определять размер страховой премии и франшизы, а также четко пропишут в договоре, что покрывается страховкой, а что – нет.

Изменение контекста рисков

Если вы решите принять риск, связанный с системой, то очень важно где-то зафиксировать этот факт и регулярно пересматривать свое решение. Особенно это относится к гибким командам.

Риски не остаются раз и навсегда неизменными. Это проявления угроз и уязвимостей системы. Это означает, что статус, значимость и важность рисков со временем изменяются, и, в частности, они изменяются вместе с изменением системы.

При традиционном управлении рисками это необязательно трудная проблема. Если система изменяется только после завершения крупных проектов (например, ежегодно), то к процессу управления рисками приходится возвращаться раз в год или около того.

Но гибкая команда постоянно изменяет систему в ответ на новую информацию, поэтому контекст, в котором риск был принят, может кардинально измениться за сравнительно короткое время.

Традиционные процессы управления очень трудно адаптировать к скорости работы гибкой команды, и делается это редко, а следовательно, решения о рисках часто принимаются в контексте отдельных изменений. Однако риски срастаются или смешиваются, и уловить это или построить адекватную модель затруднительно.

Аутсорсинг не устраняет риски

Передача ответственности за кодирование, тестирование или эксплуатацию на сторону не устраняет риски.

Передача части работы третьей стороне может иметь смысл с точки зрения бизнеса, а иногда помогает снизить эксплуатационные риски и риски безопасности. Удачным примером может служить поставщик корпоративных облачных служб типа AWS, у которого разместить приложение выгоднее, чем организовывать собственный безопасный ЦОД. Но все равно никто не снимет с вас ответственность за понимание и управление рисками приложения, а также новыми рисками, возникающими в связи с аутсорсингом. Регулирующие органы и страховщики будут настаивать на этом.

Вы должны быть уверены, что всякий, кто выполняет работу от вашего имени, ответственно подходит к защите ваших данных и данных ваших клиентов, что он надлежащим образом относится к управлению эксплуатационными рисками и безопасности.

Если ваш вес как клиента достаточно велик (крупная организация или большая сумма сделки), то вы можете настаивать на особых условиях контракта. Как крупные, так и мелкие фирмы должны регулярно анализировать работу основных поставщиков служб, включая это в программы управления эксплуатационными рисками.

В этом деле может помочь анкета для сбора структурированной информации (Structured Information Gathering – SIG), которую можно за небольшую плату скачать по адресу *https://sharedassessments.org*. Она покажет, как оценивать риски, сопряженные с поставщиком услуг. В ней организацию просят описать предпринимаемые ИТ-департаментом меры по обеспечению конфиденциальности, безопасности и управлению рисками. Ее можно использовать, когда вы подаете заявку на оказание услуги, в процессе регулярной инспекции или даже для оценки собственных мер контроля.

Анкета SIG – это стандартизованный полный набор вопросов, призванный обеспечить соблюдение нормативных требований и таких отраслевых стандартов, как PCI DSS, FFIEC, NIST и ISO. Ее можно адаптировать под конкретные нормативно-правовые требования или руководящие указания. SIG Lite – упрощенная анкета с меньшим числом вопросов, которую можно использовать для менее критичных поставщиков или на этапе начальной оценки рисков.

Например, один из авторов книги работал над системой единой точки входа в систему. В сценарии восстановления пароля была ошибка, из-за которой система в конце процедуры пускала пользователя, не требуя повторно ввести пароль.

Это создавало уязвимость, поскольку человек мог проследовать по ссылке в письме и, не введя старый пароль, перехватить управление учетной записью. Однако риск перехвата письма и эксплуатации этой уязвимости был признан низким, поэтому команда решила принять его и добавить в журнал рисков.

Прошло 12 месяцев, и та же проблема вновь была обнаружена в тесте безопасности. Но теперь проблема стала куда серьезнее, поскольку за эти месяцы система единой точки входа стала использоваться для входа в несколько других систем, содержащих секретные данные.

При добавлении новых систем в качестве дополнительной меры безопасности было решено использовать второй фактор, в данном случае код, полученный от генератора одноразовых паролей, который нужно было ввести вместе с паролем. Это требование было реализовано, и система считалась безопасной.

Однако ошибка в процедуре восстановления пароля теперь позволяла обойти двухфакторную аутентификацию, что являлось гораздо более существенной уязвимостью.

Команда смогла исправить ошибку и развернуть исправление в течение нескольких часов после выявления. Из этого инцидента был извлечен урок – хотя решение принять риск было обоснованным для начального контекста, оказалось нелегко осознать, что его следовало пересматривать после принятия других решений, изменяющих контекст, в частности добавления новых чувствительных систем или двухфакторной аутентификации.

Управление рисками в гибких методиках и DevOps

Гибкая разработка и применение практик DevOps приводят к проблемам при использовании традиционных методов управления рисками. Мы оказываемся на сравнительно новой территории, где отсутствуют четкие правила. Сообщество специалистов по управлению рисками еще не согласовало методологию управления рисками, совместимую с гибкими методиками, поэтому мы наблюдаем столкновение разных культур при попытке управлять рисками, так чтобы это было приемлемо как для всех сторон в организации, так и для аудиторов.

В частности, обычно гибкие практики воспринимаются как источник целого нового класса рисков.

Скорость поставки

Самый большой новый риск состоит в том, что гибкая команда стремится к высокой скорости поставки, а это приводит к такому темпу изменений, на который традиционные подходы к управлению рисками не рассчитаны.

Общепринятые методики контроля изменений, например определенные в ITIL или COBIT, создавались для проектов, управляемых по каскадной модели, когда большие порции изменений устанавливаются несколько раз в год, поэтому они не могут справиться с непрерывной поставкой или непрерывным развертыванием.

Ручное тестирование и контрольно-пропускные пункты, в т. ч. тестирование на проникновение и аудит соответствия нормативным требованиям, – это длительные процессы, в которых на оценку одной сборки может уйти несколько недель. Эти процессы необходимо фундаментально переосмыслить, чтобы адаптировать их к скорости поставки в гибких методиках.

А раз так, то многие гибкие команды отставляют в сторону, игнорируют или обходят практики управления риском. И тем самым создают новый риск: что ребенка выплеснут вместе с водой, что контроль рисков будет полностью отменен, а не заменен чем-то, обеспечивающим аналогичный, но более простой набор сдержек и противовесов за меньшую цену.

Однако при рассмотрении рисков безопасности скорость поставки может оказаться и преимуществом.

Немногие организации, работающие над проектами в традиционной среде, признают, что медленное, строго контролируемое развертывание само по себе является источником риска, поскольку в процесс изменения вовлечено очень много людей и количество «подвижных деталей» слишком велико. Поэтому если возникает потребность в срочных изменениях, то, чтобы уложиться в срок, приходится пропускать многие процессы контроля.

Гибкие команды производят изменения регулярно, поэтому риск, что одно какое-то изменение сможет нанести ущерб безопасности системы, значительно меньше, тогда как способность команды откатить изменение намного выше. Быстрые гибкие команды также могут оперативно реагировать на риски, уязвимости и прочие проблемы, как только о них становится известно.

Инкрементное проектирование и рефакторинг

Традиционные практики во многом опираются на такие артефакты процесса разработки, как зафиксированные в письменном виде спецификации проекта, модели, детальные технические требования и т. д.

Предполагается, что эти артефакты обновляются параллельно с внесением изменений в систему, так что в процессе управления рисками есть шанс проанализировать изменения проекта на предмет уязвимостей и рисков и предложить меры по их смягчению или модифицировать сам проект.

В гибких методиках упор делается на инкрементное и итеративное проектирование, а значит, не существует проекта системы, который можно было бы подвергнуть критическому анализу до начала работы. Да и артефактов, обновляемых при внесении изменений, тоже может не быть.

При таком подходе повышается риск, что команда не обратит внимания на ограничения, налагаемые нормативными требованиями, и не будет тратить времени на предвосхищение требований к безопасности при проектировании и планировании, а сосредоточится на поставке функциональности. К тому же из процесса устраняются контрольно-пропускные пункты и инспекции, которые в каскадной модели обязательны и на которые руководители и регуляторы привыкли полагаться.

Это также может означать отсутствие аудитопригодных точек принятия решений и артефактов, которые впоследствии могли бы изучить аудиторы и уполномоченные по соответствию нормативным требованиям. Вместо этого они должны обращаться к коду и искать там историю принятия проектных решений.

Ко всему прочему, добавить безопасность на поздних стадиях проектирования очень трудно, а в гибких методиках нет естественных стимулов думать о безопасности по умолчанию (если бы были, то и эта книга не понадобилась бы!).

Гибкие методики рассчитаны на способность вносить изменения при изменении окружающего контекста; поэтому если ландшафт безопасности меняется и возникают угрозы, то команда может отреагировать быстро – исправить проект и поведение системы.

Кроме того, адепты гибких методик говорят, что хотя они и не производят много документации, их опыт показывает, что документация редко отражает реальные свойства системы, понимая под этим, что *дрейф конфигурации* между прописанным в документации и фактическим поведением системы может быть огромным.

Заставляя безопасников, аудиторов и уполномоченных по соответствию нормативным требованиям изучить сам код, они приглашают смотреть на истинное устройство системы, а не на воображаемую реальность, описанную в проектной документации.

Самоорганизующиеся автономные команды

Гибкие команды обычно склонны к самоорганизации и автономной работе. Они сами контролируют рабочую нагрузку, журнал пожеланий продукта и способ поставки.

Это означает, что, в отличие от некоторых более традиционных компаний, разрабатывающих ПО, гибкая команда может противиться или вовсе отказаться от инспекционных комиссий, ответственных проектантов и прочих навязанных извне контрольных механизмов, если сочтет, что они только мешают быстрой поставке. Это представляет проблему для специалистов по безопасности, которые привыкли совместно с инспекционной комиссией по архитектуре и другими центральными органами вырабатывать руководящие принципы и правила обеспечения безопасности всех систем.

Кроме того, это обычно означает, что руководство меньше вникает и не так строго контролирует детали процессов разработки в каждой команде, а также «материальную ведомость» используемого ПО: библиотеки и каркасы. Это может вызвать возражения со стороны аудиторов и уполномоченных по соответствию, которые хотят точно знать, как была организована работа и какие версии библиотек использованы в каждом продукте.

Гибкие команды убеждают руководство верить тому, что они все сделают правильно, а также полагаться на личные качества и компетентность владельца продукта, который не допустит снижения качества.

Адепты гибких методик говорят, что контроль и принятие решений сверху вниз, практикуемые в традиционных инженерных методологиях, часто приводят к решениям, не эффективным для конкретной команды или продукта, а изменить эти решения оказывается сложно и долго из-за большого *радиуса поражающего действия* решения.

Например, решение перейти на последнюю версию библиотеки касается только команды, работающей над конкретным продуктом, но вынесение его на рассмотрение инспекционной комиссии по архитектуре означало бы, что нужно будет анализировать последствия этого решения для всех команд, работающих в организации.

Самоорганизующиеся и автономные команды снижают риск медленного принятия решения и создают брандмауэры между разными ко-

мандами, вследствие чего уязвимости, затрагивающие одну команду, необязательно повлияют на все остальные. Это также наделяет организацию способностью реагировать точечно и соответственно ситуации – силами каждой отдельной команды.

Автоматизация

Гибкие команды активно используют автоматизацию для обеспечения высокой скорости, повторяемости и согласованности, необходимых для движения вперед.

Однако самой автоматизации свойственны риски. Используемые инструменты могут стать мишенью и вектором атаки, мы будем говорить об этом в главе 13.

Автоматизированная система может способствовать распространению и умножению ошибок, дефектов и атак, так что ущерб будет намного выше, чем в ручной системе. В твиттер-аккаунте *@DevOpsBorat* есть такая шутка: «Делать ошибки – свойство человека. Автоматически распространить ошибку на весь сервер – это уже #devops»[2].

Кроме того, автоматические инструменты тоже небезупречны; в мире безопасности давно известно, как легко человек начинает доверять компьютеру и вообще забывает про здравый смысл. Так появляются на свет команды, считающие, что если тесты прошли, значит, система работает правильно, даже если все говорит о противоположном.

Автоматизированное управление конфигурацией, а также конвейеры сборки и развертывания могут поспособствовать разделению обязанностей, предоставив аутентифицируемые и аудируемые средства доступа к системе для внесения изменений. Для разработчиков это сводит к минимуму или вообще устраняет необходимость заходить напрямую в производственную систему. Но автоматизация может и мешать разделению обязанностей, поскольку не позволяет четко определить, у кого есть доступ к системе и кто может вносить в нее изменения. Этот вопрос мы подробно изучим в главе 14.

Гибкое смягчение риска

Гибкие методики способны до некоторой степени смягчить риски, свойственные традиционным практикам.

Известно, что каскадная модель, V-модель и прочие традиционные системы разработки ПО неизменно служат причинами выхода за пределы бюджета, сроков и рамок проекта. Для гибких проектов перерас-

[2] *https://twitter.com/DEVOPS_BORAT/status/4158716887079731*.

ход – тоже не диво, но их руководители обычно узнают об опасности перерасхода раньше, а принципы YAGNI (тебе это не понадобится), своевременного назначения приоритетов и непрерывной поставки работающего ПО с итеративным улучшением гарантируют, что даже если проект будет досрочно прекращен, все равно он принесет некоторую пользу организации.

Кроме того, благодаря частым демонстрациям, более коротким циклам обратной связи, проектированию минимального жизнеспособного продукта на каждой итерации и способности оперативно реагировать на изменения рисков для бизнеса в случае поставки ПО, не точно отвечающего потребностям пользователя, значительно снижаются.

Многолетний опыт доказывает, что изменения, внесенные в систему на поздних стадиях цикла разработки, обходятся дороже, чем внесение изменений на ранних стадиях. Но традиционный ответ на эту проблему состоял в том, чтобы продлить и сделать более консервативными ранние стадии в попытке выработать более полные и всесторонние требования и проектные спецификации, а не сосредоточиться на минимизации стоимости изменений.

Даже если проект успешно разработан, следуя каскадной модели или V-модели (а таких проектов много), все равно время разработки – это лишь краткий миг в жизни системы, а большую часть времени она проводит, выполняя свою задачу, что требует сопровождения и изменения. Гибкие методы могут помочь в снижении затрат и рисков, связанных с такими изменениями, на протяжении всей жизни системы.

Наконец, симптомом проблемы является тот факт, что во многие системы встраиваются очень развитые средства конфигурирования, а проект чрезмерно усложняется в попытке предвидеть или минимизировать стоимость будущих изменений. Типичные примеры – использование «машин обработки бизнес-правил» и «сервисные шины предприятия», призванные вынести наружу наиболее часто изменяемые компоненты системы и упростить их изменение.

Однако эти подходы только увеличивают сложность и издержки управления эксплуатацией системы. Кроме того, они сильно усложняют тестирование и репликацию среды, поскольку изменения необходимо протестировать в различных возможных конфигурациях, которым несть числа.

Программы, разработанные с применением гибких и бережливых методик, должны быть проще по построению, а если организация способна быстро писать и выпускать код изменений, то даже значительно проще. Чем проще система, тем меньше в ней компонентов, которые

могут «сломаться», и, следовательно, риск существенно ниже. Их проще понять, о них проще рассуждать, а это делает их безопаснее.

Наконец, чем чаще вносятся изменения, тем больше шансов, что что-то придется «сломать». Команды, практикующие гибкие методики, а особенно DevOps, сносят изменения гораздо чаще. В противоречие с интуицией исследования показывают, что в организациях, где частота изменений максимальна, частота отказов ниже, а надежность выше[3].

Как же они этого добиваются?

- Выпуск изменений небольшими порциями, непрерывное развертывание, использование «канареечных релизов» (для ограниченного числа пользователей) для контроля риска, связанного с развертыванием изменений.
- Оптимизация среднего времени восстановления за счет предвидения отказов и заблаговременной подготовки к ним.
- Автоматизация управления конфигурацией и развертывания с целью обеспечить согласованность и повторяемость.
- Встраивание циклов обратной связи в эксплуатационные процедуры и между эксплуатационниками и разработчиками, чтобы как можно скорее вылавливать и исправлять проблемы. Создание эффективного механизма реагирования на инциденты.

Команды, практикующие гибкие методики и DevOps, не имеют обыкновения много размышлять о таких видах риска, коль скоро придерживаются своих правил и правильно используют инструменты. Их успех во многом зависит от того, насколько хорошо владелец продукта понимает важность минимизации технических и эксплуатационных рисков и соглашается назначать высокие приоритеты необходимым работам, от готовности Scrum-мастера и всей команды работать ответственно и до доверия и готовности к сотрудничеству на всех уровнях с целью устранить взаимное непонимание и выставлять возникающие проблемы на всеобщее обозрение, чтобы их можно было решить.

В более крупных проектах применяются каркасы гибкого управления на уровне предприятия, например SAFe[4] (Scaled Agile Framework) и DAD[5] (Disciplined Agile Development); они поднимают управление рисками выше уровня команды. Эти каркасы явно вносят заблаговременное планирование и проектирование, отчетность и точки интеграции между разными командами, а также дополнительные роли для архи-

[3] См. отчет «2016 State of DevOps Report» компании Puppet Labs.

[4] *https://www.scaledagileframework.com/*.

[5] *http://www.disciplinedagiledelivery.com/*.

тектурного и проектного управления, позволяющие управлять рисками и зависимостями, связанными с наличием нескольких команд и частей проекта.

Многие опытные эксперты по практическому применению гибких методик скептически относятся к этим обещаниям, потому что уже доказано, как невероятно трудно примирить традиционные каскадные подходы к управлению проектами с гибкими принципами. Корпоративные каркасы управления могут помочь при переходе организации на гибкие методики, не отбрасывая сразу весь накопленный опыт управления, но мы рекомендуем ступать осторожно, следя за тем, чтобы, придерживаясь этих каркасов, не растерять все преимущества гибкости.

Отношение к рискам безопасности в гибких методиках и DevOps

Профессионалы в области безопасности часто позиционируют себя отдельно от профессионалов по управлению рисками. Они говорят, что деловые риски существенно отличаются от рисков безопасности, что безопасность – это особый подход к разработке ПО.

До некоторой степени это правда. Риски безопасности действительно куда менее понятны, чем традиционные деловые риски, их гораздо проще описать как «неизвестные неизвестные» и гораздо труднее измерить в терминах ущерба или последствий.

Однако безопасность – это просто еще одно свойство программного обеспечения, как производительность, качество, эффективность и удобство пользования.

Гибкие методики разработки иногда испытывали трудности с этими свойствами, которые обычно называют нефункциональными требованиями, потому что их очень сложно выразить в виде пользовательских историй.

Эти свойства, и в особенности безопасность, стали чем-то таким, о чем команда знает, чем владеет и для чего имеет внутренние механизмы управления, задействуемые при поставке каждой истории.

Рисками безопасности можно управлять, как любыми другими, путем встраивания механизмов управления рисками в подходы к проектированию, разработке, тестированию, развертыванию и эксплуатации системы.

В традиционном жизненном цикле разработки ПО оценка рисков основана на требованиях к системе, проектных спецификациях и моделях, создаваемых в самом начале. Аналитик пользуется этими докумен-

тами, чтобы выявить присутствующие в системе риски и составить план по их мониторингу и смягчению. Затем проводится аудит, призванный подтвердить, что созданная система отвечает документальным спецификациям и что план управления рисками по-прежнему актуален.

В случае итеративной и инкрементной гибкой разработки риски нужно постоянно переоценивать и непрерывно управлять ими. В гибких методиках, в частности Scrum, есть несколько мест, в которые можно встроить действия по управлению рисками.

Планирование спринта

Анализ и перечисление рисков.

Написание историй

Особое внимание историям, несущим риски безопасности и конфиденциальности. Противодействие рискам безопасности и соответствия нормативным требованиям путем написания историй, касающихся безопасности и соответствия, а также историй противника.

Написание тестов

Добавление автоматизированных тестов безопасности и проверок на соответствие нормативным требованиям.

Кодирование

Использование проверенных библиотек и паттернов.

Инспекции кода

Особое внимание рискам безопасности во время инспекций кода (особенно высокорискового), сканирование всего кода автоматизированными средствами статического анализа.

Рефакторинг

Уменьшение технической сложности кода и проекта путем применения дисциплинированного рефакторинга.

Проектирование

Моделирование угроз всякий раз, как вносятся высокорисковые изменения в части системы, лежащие на поверхности атаки.

Ретроспективный анализ

На совещаниях команды, посвященных изучению возможностей для улучшения, рассматривать риски безопасности и соответствия нормативным требованиям наряду с другими техническими и эксплуатационными рисками. Обсуждать управление рисками.

«Посмертные» инспекции

Использование сведений, полученных после отказа или инцидента, чтобы проанализировать стоящие за ними риски и выработать решения.

Управление рисками также основывается и, в свою очередь, является движущей силой для автоматизации и стандартизации тестирования и развертывания посредством процессов непрерывной интеграции и непрерывной поставки. И результаты управления рисками служат входными данными для управления конфигурацией, мониторинга и других эксплуатационных процессов.

По существу, все практики, инструменты и методы, описанные в этой книге, являются частью программы управления рисками.

Сухой остаток

Большинство разработчиков, особенно в гибких средах, явно не имеют дела с управлением рисками, поскольку им это не нужно. Гибкие методики и практики позволяют разработчикам естественно учитывать многие базовые риски: проектные, технические и деловые.

Управление рисками безопасности, конфиденциальности и соответствия нормативным требованиям должно стать частью мышления и действий разработчиков точно так же, как управление другими рисками. Для этого:

- помогите команде понять типичные риски безопасности, например перечень 10 главных рисков от OWASP, и общепринятые способы управления ими;
- сделайте так, чтобы риски безопасности и другие риски были хорошо видны команде (и руководству), например в виде реестра рисков, или историй, касающихся безопасности и соответствия нормативным требованиям, в журнале пожеланий;
- риски и управление ими должны явным образом рассматриваться в процессе ретроспективного анализа и стать частью циклов обратной связи;
- гибкие команды снижают эксплуатационные риски и риски безопасности, поскольку вносят изменения непрерывно, но небольшими порциями. Мелкие изменения проще понять и протестировать, их развертывание безопаснее, а если что-то пойдет не так, то будет проще исправить;

- управление рисками при гибкой разработке должно быть интерактивным. Решения о принятии или смягчении рисков должны пересматриваться по мере изменения проекта системы;
- пользуйтесь преимуществами гибких практик и DevOps и встроенными в них точками контроля, чтобы добавить меры управления рисками и представить доказательства предпринятых действий аудиторам. Мы подробнее рассмотрим этот вопрос в главе 14.

Глава 8

Оценка угроз и осмысление атак

Невозможно выстроить эффективную защиту от противника, которого не видишь и не понимаешь. Необходимо понимать, что угрожает организации, системам и данным клиентов, и быть готовым отразить эти угрозы.

Для этого нужно иметь актуальную и точную информацию об угрозах.

- Настройте свои системы мониторинга безопасности, так чтобы они знали, что искать, и окружите организацию и системы средствами защиты от атак на этапе выполнения.
- Назначьте приоритеты применению исправлений и другим мерам устранения уязвимостей.
- Оценивайте эксплуатационные риски, чтобы иметь представление о том, в какой мере организация подготовлена (или не подготовлена) к отражению атак.
- Помогайте писать истории, касающиеся безопасности, моделируя злоумышленников в виде антиперсон.
- Опишите тестовые сценарии атаки на собственные системы с помощью таких инструментов, как Gauntlt, чтобы найти уязвимости раньше противника.
- Руководите обучением и просвещением.
- Оценивайте безопасность проекта посредством моделирования угроз.

Осмысление угроз: паранойя и реальность

Если читать новости или слушать комментаторов, то мир покажется довольно страшненьким местом. Неиссякающий поток уязвимостей и

скомпрометированных крупных систем проливается в новостях каждую неделю: широко известные организации падают жертвами самых разных атак и утрачивают контроль над конфиденциальной информацией.

Слушая всё это, легко предположить, что эти организации и системы в чем-то были исключениями из правил. Что было что-то необычное в способах атаки на них, или что они были выбраны мишенью, потому что чем-то отличались, или что они были особенно плохо настроены или разработаны. Но на самом деле абсолютно все системы и сети, подключенные к Интернету, являются мишенями атак в любой момент времени.

В Интернете обитает целая экосистема потенциально враждебных индивидуумов, групп и автоматизированных систем, которые, ни на секунду не останавливаясь, ищут новые уязвимые системы и слабые места, которые можно эксплуатировать. Организуемые ими атаки могут иметь конкретную цель, а могут быть неизбирательными и оппортунистическими по природе – подбирают все, что плохо лежит.

Наблюдая столь активную враждебную деятельность, немудрено пасть духом. Но хотя игнорировать угрозы опасно, нельзя бояться всего, эдак вы только парализуете свою организацию.

Так каким же должно быть отношение к угрозам, позволяющее предпринимать действия и принимать правильные решения? Начнем с простых вопросов. Кто мог бы атаковать вашу систему? С какой целью? Как он мог бы добиться успеха? Какие угрозы наиболее важны прямо сейчас? Как их опознать и наиболее эффективно выстроить защиту?

Понимание природы злоумышленников

Специалисты по безопасности для описания людей, организаций и вообще сторон, которые могли бы представлять риск (или угрозу) вашей организации, системам и людям, часто употребляют слово «злоумышленник» (threat actor). Один из способов понять природу злоумышленников, их возможности и намерения состоит в том, чтобы построить профиль, представить, как они могли бы вести себя по отношению к нам, а затем подумать, как их идентифицировать и нейтрализовать.

Прежде чем подробно рассматривать типичные архетипы злоумышленников, следует понять их общие черты. Можно выделить пять элементов описания злоумышленника. Они коррелируют с вопросами, которые хотелось бы задать о человеке, группе или организации в процессе анализа.

Краткое описание

Кто этот человек, группа или организация, какова его история?

Мотивация

Почему они хотят атаковать именно нашу организацию, технологию или сотрудников?

Цель

Чего они хотят добиться, атаковав нас?

Ресурсы

Насколько они квалифицированы, какими финансовыми и прочими возможностями располагают? Есть ли у них доступ к инструментам, людям или фондам, необходимым для проведения атаки?

Характеристики

Где они находятся? Каково их типичное поведение в ходе атаки (если известно)? Пользуются ли широкой известностью?

Истории противника и моделирование угроз как требования

Не забывайте, что моделирование угроз и определение историй противника взаимосвязаны. Вернитесь к главе 5, чтобы освежить в памяти, как этот образ мышления включается в процесс сбора требований.

Структурировав свои представления о злоумышленниках, применим их к описанию типичных профилей и типов противников.

Архетипы злоумышленников

Есть много способов классифицировать типичных злоумышленников, угрожающих организации. Для простоты сгруппируем их по положению относительно нашей организации.

Инсайдеры

Инсайдер имеет доступ к системе изнутри и занимает должность, пользующуюся доверием. Вот несколько типов инсайдеров:

- озлобленные или в чем-то ущемленные сотрудники;
- бывшие сотрудники, у которых мог сохраниться привилегированный доступ (например, к внутренним системам, облачным приложениям или оборудованию);

- активисты внутри организации: лица, имеющие «зуб» на организацию или желающие ей вреда по какой-то личной причине;
- шпионы: охотники за промышленными секретами или агенты иностранной разведки (да, такое бывает не только в кино);
- марионетки: работники, которых удалось угрозами принудить или с помощью методов социальной инженерии убедить использовать свой привилегированный доступ в чужих интересах;
- сотрудники, не по злой воле допустившие дорогостоящую ошибку (особенно администраторы с привилегированным доступом к системе).

Посторонние лица

Посторонние лица тоже могут провести атаку, но занимая при этом позицию извне, не облеченную доверием. Это могут быть:

- пользователи, беспечные, небрежные или просто любопытные, случайно ставшие причиной неприятностей;
- мошенники;
- боты, которые непрерывно и автоматически сканируют все вокруг в поисках уязвимостей;
- хакеры и взломщики-дилентанты: случайные придурки, стремящиеся произвести впечатление на окружающих своими набранными с бору по сосенке умениями;
- исследователи, занимающиеся безопасностью, или «исследователи в кавычках»: квалифицированные профессионалы, желающие «повыделываться»;
- хактивисты: группы с определенными моральными или политическими убеждениями, которые хотят атаковать вашу организацию или отрасль по идеологическим причинам. Такие атаки могут принимать разные формы: от отказа в обслуживании и искажения внешнего вида сайта до кражи секретных (и потенциально компрометирующих) документов и данных;
- организованные преступники, стремящиеся украсть данные или IP-адрес, получить выкуп, провести DDOS-атаку или иным способом шантажировать организацию;
- поддерживаемые государством группы, стремящиеся украсть данные или IP-адрес, произвести разведку или устроить саботаж в качестве акта кибервойны (в большинстве случаев такие действия не укладываются в модель угроз, и вряд ли вы сможете сделать что-то для обнаружения или предотвращения атаки).

Злоумышленники и мотивы

Продолжая попытки классификации, мы можем детализировать мотивы и цели. Если оставить в стороне злоумышленников, которые причиняют вред случайно или по халатности, то можно выделить пять групп мотивов.

Материальный

Атакующий хочет повысить свое благосостояние путем кражи информации, денежных средств или иных активов (или путем манипулирования данными либо рынками).

Политический

Атакующие преследуют политические или религиозные цели.

Самореклама

Атакующие хотят заявить о себе или продемонстрировать свои возможности.

Личный

Атакующим движет личная обида или эмоциональная реакция.

Стремление к хаосу

Атакующие хотят причинить максимальный ущерб и внести смятение просто потому, что могут. Желание приколоться – единственное, что движет такими злоумышленниками.

Бывает, конечно, что у злоумышленников есть несколько из вышеперечисленных мотивов, но обычно лишь один мотив определяет целеустремленность, а вместе с ней располагаемые ресурсы, энтузиазм и время, которое противник готов потратить на подготовку и проведение атаки.

Несколько лет назад большинство организаций, не принадлежащих к финансовому сектору, еще могли сделать вид, что их системы не интересуют профессиональных преступников и все внимание нужно сосредоточить на защите от инсайдеров, хакеров-дилетантов и других оппортунистов. Но сегодня преступники, а также группы хактивистов и другие хорошо мотивированные противники получили гораздо более простой доступ к инструментам и информации, необходимой для проведения изощренных и разрушительных атак, поэтому способны распространить свою деятельность на все новые и новые системы.

Принимая во внимание современное положение с атаками в Интернете, оправдано предположение о том, что каждая система может

стать мишенью атаки. И хотя неизбирательные попытки компрометации «на авось» по-прежнему имеют место, важно понимать, что и более умелые и целеустремленные злоумышленники также могут посматривать на ваши системы. Пусть даже вы не считаете себя достаточно важной целью для их радаров, спектр мотивов настолько широк, что рано или поздно вы неизбежно станете более интересной мишенью, чем думаете.

Те из нас, кто работает в правительстве, на предприятиях критической инфраструктуры или в финансовом секторе, а также в стартапах, разрабатывающих передовые технологии, должны ориентироваться на более умелых злоумышленников. Хотя группы, поддерживаемые государством, не являются основной угрозой для большинства предприятий, в некоторых отраслях эта угроза весьма и весьма актуальна.

Идентифицировать злоумышленников, существенных для вашей организации, зачастую далеко не просто, но на это стоит потратить время, если вы хотите направить свои ограниченные ресурсы на защитные меры, которые действительно помогут службе безопасности, когда эти ребята постучатся в дверь. Понимание мотивов злоумышленников и целей, которых они хотят достичь, атакуя вашу организацию, поможет яснее представить, как именно вас могут атаковать.

Угрозы и цели атаки

Различные организации и системы сталкиваются с разными угрозами в зависимости от отрасли промышленности, размера, типа данных, топологии сети и других факторов. Начинать анализ возможных угроз следует с вопроса о том, что может быть целью атаки.

Ваша организация

Из-за отрасли промышленности, торговой марки, репутации или людей, которые в ней работают.

Ваши данные

Из-за их ценности и (или) информации, к которой они имеют отношение.

Ваша служба

Из-за предоставляемой ей функциональности.

Ваши ресурсы

Из-за доступных вам вычислительных мощностей, систем хранения или полосы пропускания.

Ваша экосистема

Из-за связей с другими системами (которые могут быть использованы как трамплин для атак на другие системы или организации, как в случае атаки на компанию Target через ее поставщика, HVAC).

Данные – наиболее типичная цель атаки: их пытаются украсть, скомпрометировать или уничтожить. Вот несколько вопросов, касающихся данных:

1. Какие данные хранятся и обрабатываются в вашей системе? К каким конфиденциальным или секретным данным она имеет доступ?
2. У кого могло бы возникнуть желание украсть данные?
3. Что произойдет, если данные будут украдены? Что с ними можно было бы сделать? Кому будет причинен ущерб? Насколько серьезен будет ущерб?
4. Сколько данных необходимо украсть, чтобы это было существенно для клиентов, конкурентов, регулирующих или правоприменительных органов?
5. Как узнать, что данные были украдены?
6. Что, если данные будут уничтожены? Каков будет ущерб или убыток? Атаки программ-вымогателей быстро изменяют отношение организации к такой угрозе.
7. Что, если данные будут несанкционированно изменены? Каков будет ущерб или убыток? Сможете ли вы понять, что это произошло?

Разведка угроз

Существуют различные источники информации об угрозах, из которых можно понять, кто злоумышленники и какие риски они несут организации? Хотя это та область индустрии безопасности, которую принято считать рекламной шумихой, не оправдавшей возлагавшихся на нее надежд (см. предупреждение «Грозненькие угрозы» ниже), все же ей может найтись место в вашей программе обеспечения безопасности. Ниже приведено несколько категорий разведки угроз, которые могут оказаться вам полезны.

Центры обмена информацией

Замкнутые сообщества, позволяющие государственным учреждениям и промышленным предприятиям обмениваться представляющими интерес разведывательными данными об угрозах и рисках, в т. ч. признаками компрометации. Играют также роль

форума, участники которого делятся передовыми практиками и добровольно сообщают об инцидентах. Примерами могут служить FS ISAC для финансового сектора США, Auto ISAC для автопроизводителей, Центр обмена информацией о безопасности Министерства обороны США (US Defense Security Information Exchange – DSIE), Центр обмена информацией о кибербезопасности Великобритании (UK Cybersecurity Information Sharing Partnership – CISP) и Канадский центр реагирования на инциденты кибербезопасности (Canadian Cyber Incident Response Centre – CCIRC).

Государственные информационные бюллетени

Информационные бюллетени об угрозах и рисках, выпускаемые государственными службами и правоохранительными учреждениями, такими как US-CERT и FBI Infragard.

Каналы оповещения об угрозах, принадлежащие конкретным производителям

Новая информация об уязвимостях, распространяемая производителями программного обеспечения и сетевого оборудования.

Белые и черные списки, публикуемые в Интернете

Обновляемые списки достойных доверия и опасных IP-адресов, которыми можно пользоваться для построения белых и черных списков для брандмауэров, систем обнаружения и предотвращения вторжений и систем мониторинга.

Консолидированные каналы информирования об угрозах

Открытые и коммерческие службы новостей об угрозах, в которых консолидирована информация из разных источников и иногда проводится дополнительный анализ рисков.

Аналитические отчеты и исследования

Углубленный анализ, в частности исследование M-Trends компании Mandiant и отчет Data Breach Investigation Report компании Verizon.

Дополнительные сведения см. на странице «A curated list of Awesome Threat Intelligence resources» по адресу https://github.com/hslatman/awesome-threat-intelligence.

Эти источники предоставляют информацию об угрозах вашей отрасли или сектору, клиентам и поставщикам, включая новые и модифицированные вредоносные программы, обнаруженные в сети, новые уязвимости, которые активно эксплуатируются, известные источники вредоносного трафика, методы мошенников, новые виды атак, уязвимости нулевого дня, признаки компрометации, новости о взломах, оповещения, публикуемые правительством и правоприменительными органами.

Ниже перечислены платформы для обнаружения, сбора, агрегирования и составления отчетов об угрозах:

- Open Threat Exchange (*https://www.alienvault.com/open-threat-exchange*);
- OpenTPX (*https://www.opentpx.org/*);
- PassiveTotal (*https://community.riskiq.com/*);
- Critical Stack (*https://intel.criticalstack.com/*);
- Facebook ThreatExchange (*https://www.facebook.com/threatexchange*).

Информация об угрозах поступает в разных форматах: отчеты, уведомление по электронной почте, динамические ленты структурированных данных. Два проекта с открытым исходным кодом, созданных сообществом, направлены на то, чтобы упростить обмен информацией об угрозах между различными организациями и инструментами. Для этого определены стандарты описания и отчетности об угрозах.

Грозненькие угрозы

С сожалением констатируем, что для многих работающих в области безопасности разведка угроз превратилась в подобие дежурной шутки с безосновательными претензиями на ценность, которая так никогда и не материализуется, в конкуренцию между поставщиками, борющимися за количество информации, а не ее качество и релевантность. Очень зрелищные, попикивающие карты, анимированные в лучших традициях Голливуда, имеют крайне сомнительную практическую ценность и в реальных ситуациях пока никому не помогли. Презрение к этой части отрасли настолько велико, что в насмешку даже созданы пародийные сервисы типы Threatbutt.

Как часто бывает в информационной безопасности, вы должны ясно понять и оценить истинную ценность различных аспектов разведки угроз для своей организации, а не бездумно следовать моде.

STIX (*https://stixproject.github.io/*) – проект сообщества, призванный стандартизировать формат информации об угрозах, так чтобы он был понятен людям и инструментам агрегирования – точно так же, как SCAP или CWE. Стандарт определяет структурированный язык описания угроз, включая информацию о признаках (что видят другие организации).

TAXII (*http://taxiiproject.github.io/*) – еще одна инициатива по стандартизации способов обмена данными об угрозах, в которой также рассматривается вопрос об идентификации и реакции на информацию об угрозе. На сайте HailaTaxii.com имеется репозиторий открытых каналов об угрозах в формате STIX.

Оценка угроз

Следующий шаг – определить, какие угрозы наиболее существенны для организации и системы: к каким угрозам нужно готовиться немедленно.

Разведка угроз для разработчиков

По большей части разведка угроз состоит из обновления сигнатур файлов (для идентификации вредоносного ПО), а также рассылки уведомлений о фишинге, DDOS-атаках, программах-вымогателях и попытках мошенничества. Эта информация представляет интерес для группы безопасности, эксплуатационников и ИТ-департамента, но необязательно для разработчиков.

Разработчики должны понимать, от каких угроз защищаться в процессе проектирования, разработки и внедрения. Какие типы атак наблюдаются на практике, и от кого они исходят? Какие уязвимости стороннего ПО эксплуатируются? За какими событиями необходимо следить, о чем уведомлять как об исключительных ситуациях, что пытаться заблокировать?

Наилучший источник информации зачастую находится прямо перед глазами разработчиков: их собственные системы, работающие в производственном режиме. В зависимости от того, какого типа решения разрабатываются, можно также попытаться понять, нельзя ли извлечь какую-то пользу из интеграции данных разведки услуг в само решение.

Угрозы, с которыми сталкиваются другие организации из вашего сектора промышленности, должны стоять на первом месте. Разумеется, самыми срочными являются угрозы, которые уже реализовались: атаки, которые проводятся против вашей организации и систем.

Короче говоря, мы ведем речь о *защите в ответ на атаку*, когда наблюдаемая информация об идущих атаках определяет приоритеты безопасности. Информация от систем мониторинга производственных систем дает реальные сведения об активных угрозах, их нужно осмыслить и предпринять какие-то действия. Это не теоретические домыслы, а то, что происходит прямо сейчас. Мы не хотим сказать, что все остальные, не столь неотвратимые угрозы можно или должно игнорировать. Но если бандиты стучатся в дверь, следует проверить, что она надежно заперта.

Защита в ответ на атаку порождает контуры положительной обратной связи между разработчиками и эксплуатационниками.

1. Используйте динамическую информацию об атаке, чтобы лучше понять угрозу и составить более правильное представление о том, какие угрозы имеют высший приоритет.
2. Используйте разведку угроз, чтобы выяснить, на что обращать внимание. Для этого добавьте в систему мониторинга проверки, позволяющие узнать, что атака началась.
3. Используйте всю эту информацию, чтобы назначить приоритеты тестированию, сканированию и инспекциям. Это позволит найти слабые места в системе и устранить уязвимости или хотя бы вести наблюдение за ними, до того как их найдет и попробует эксплуатировать противник.

Чтобы воспользоваться этим, необходимо иметь:

1) эффективные средства мониторинга, способные выявить признаки атаки и передать эту информацию эксплуатационникам и разработчикам для принятия мер;
2) навыки расследования и проведения криминалистической экспертизы: является ли событие изолированным случаем, который вы вовремя обнаружили, или это уже случалось прежде;
3) практически доказанную способность быстро и безопасно применять исправления в ответ на атаки, полагаясь на автоматизированный конвейер сборки и развертывания.

Поверхность атаки вашей системы

Чтобы оценить безопасность своей системы и ее уязвимость для атак, нужно прежде всего определить *поверхность атаки* – те части системы, которые интересны противнику.

В любой системе есть три разные поверхности атаки, о которых следует помнить.

Сеть

Все оконечные точки сети, включая устройства и службы, прослушивающие сеть, операционные системы, виртуальные машины и контейнеры. Чтобы оценить сетевую поверхность атаки, можно воспользоваться инструментами сканирования типа *nmap*.

Приложение

Любой способ, которым противник может ввести команды или данные, и любой способ получения им данных из системы, особенно анонимные оконечные точки, а также стоящий за всем этим код, а точнее уязвимости в этом коде, допускающие эксплуатацию противником.

Человек

Люди, принимающие участие в проектировании, разработке, эксплуатации и поддержке системы, а также пользователи системы. Все они могут стать жертвами атак методами социальной инженерии. Их можно обмануть или скомпрометировать, чтобы получить несанкционированный доступ к системе или важной информации о системе, либо превратить в злоумышленников.

Картирование поверхности атаки приложения

Поверхность атаки приложения включает, в частности:

- веб-формы, поля, заголовки и параметры HTTP, куки;
- API, особенно открытые и доступные из сети, а потому удобные для атаки, а также административные API, атака на которые оправдывает дополнительные усилия;
- клиенты: код клиента, например мобильного, представляет отдельную поверхность атаки;
- конфигурационные параметры;
- инструментальные средства, применяемые при эксплуатации системы;
- закачивание файлов на сервер;
- хранилища данных: поскольку хранимые данные можно украсть или повредить и поскольку мы читаем данные в программу из хранилища, данные могут быть использованы против системы с помощью атак типа «хранимый XSS», и не только;
- контрольные журналы: поскольку они содержат ценную информацию или могут быть использованы для отслеживания действий атакующего;

- вызовы внешних служб, например электронной почты и других систем;
- учетные данные пользователей или систем, которые можно украсть или скомпрометировать для получения несанкционированного доступа к системе (сюда входят и маркеры API);
- хранящиеся в системе секретные данные, которые можно украсть или повредить: персональные данные, закрытая медицинская информация, финансовые данные, материалы ДСП или с грифом секретности;
- средства контроля безопасности – ваши защитные бастионы: логика управления удостоверениями, аутентификацией и сеансами, списки контроля доступа, ведение контрольных журналов, шифрование, проверка входных данных и кодирование выходных данных. В общем, любые уязвимости и слабые места в этих средствах, которые противник может найти и эксплуатировать;
- и, как мы увидим в главе 11, автоматизированный конвейер сборки и поставки системы.

Помните, что сюда включается не только код, написанный вами, но и все сторонние библиотеки, а также библиотеки и каркасы с открытым исходным кодом, используемые в вашем приложении. Даже у сравнительно небольшого приложения, построенного на основе такого развитого каркаса, как Angular или Ruby on Rails, поверхность атаки может быть велика.

Управление поверхностью атаки приложения

Чем больше и сложнее система, тем больше у нее поверхность атаки. Ваша цель – сократить поверхность атаки до минимума. Это и всегда-то было непросто, а в современных архитектурах стало еще сложнее.

Создается впечатление, что у нас на руках слишком много всего, нуждающегося в наблюдении и оберегании. Но проблему – и риск – можно расчленить.

Всякий раз, внося изменение в систему, думайте, как оно отразится на поверхности атаки. Чем больше изменений, тем больше взятый на себя риск. Чем крупнее изменение, тем больше взятый на себя риск.

Изменения, вносимые командами, практикующими гибкие методики и DevOps, по большей части небольшие и инкрементные. Добавление нового поля в веб-форму технически увеличивает площадь поверхности, но инкрементный риск, скорее всего, понятен, и с ним легко справиться.

Применение микросервисов резко увеличивает поверхность атаки

Микросервисы – одна из новомодных идей в архитектуре приложений, приобретшая популярность, потому что наделяет команды разработчиков гораздо большей гибкостью и независимостью. Трудно оспаривать успех микросервисов в таких организациях, как Amazon и Netflix, но надо также понимать, что микросервисы привносят серьезные проблемы в части эксплуатации и безопасности.

Поверхность атаки отдельного микросервиса, скорее всего, невелика и легко поддается анализу. Но поверхность атаки системы в целом возрастает экспоненциально, стоит только начать использовать микросервисы хоть сколько-нибудь широко. Количество оконечных точек и потенциальных соединений между микросервисами может стать неуправляемым.

Не всегда просто понять зависимости, создаваемые цепочками вызовов: вы не можете контролировать, кто кого вызывает и что вызывающая сторона ожидает от вашего сервиса. Невозможно контролировать, что делают зависящие от вас сервисы, когда и как они изменяются.

В возглавляемых инженерами компаниях, пошедших по стопам Amazon и Netflix, разработчики вправе выбирать инструменты по своему усмотрению, а следовательно, микросервисы будут написаны на разных языках, с использованием разных каркасов. Это создает новые технологические риски и резко осложняет задачу инспектирования, сканирования и обеспечения безопасности среды.

Контейнеры типа Docker, которые зачастую являются частью инфраструктуры микросервисов, также оказывают колоссальное влияние на поверхность атаки системы. Хотя контейнеры можно сконфигурировать, так чтобы поверхность атаки уменьшилась, а изоляция между сервисами повысилась, они открывают новую поверхность для атаки, которую следует тщательно контролировать.

Однако бывают другие, куда более серьезные изменения.

Вы добавили новую административную роль или развернули новый API, видимый из сети? Изменили средство контроля безопасности или ввели новый тип секретных данных, которые надо защищать? Внесли фундаментальное изменение в архитектуру или отодвинули границу доверия? Все эти изменения должны стать причиной переоценки рисков (проекта, кода или того и другого вместе), а возможно, и проверки

соответствия нормативным требованиям. По счастью, подобные изменения встречаются гораздо реже.

Гибкое моделирование угроз

Под моделированием угроз понимается такое исследование рисков безопасности, когда на проект смотрят глазами противника и убеждаются, что все необходимые защитные меры присутствуют. Если истории противника и сценарии злонамеренного использования побуждают команду пересмотреть требования с точки зрения атакующего, то в случае моделирования угроз команда должна взглянуть другими глазами на проект и подумать, что может пойти не так на этом уровне и что с этим можно сделать.

Но не надо делать это таким тяжеловесным и затратным способом, как на этапе критического анализа в каскадной модели. Нужно лишь построить простую модель системы, рассмотреть доверительные отношения, угрозы и способы защиты от них.

Доверие и границы доверия

Границы доверия – это те места в системе, где разработчик меняет свое мнение о том, какие данные безопасно использовать и каким пользователям можно доверять. Границы – это места, в которых нужно применять средства контроля и проверять предположения о данных, личности пользователя и авторизации. Все, что находится внутри границы, должно быть безопасно. Все, что за границей, мы не контролируем и должны считать небезопасным. Все, что пересекает границу, подозрительно: виновен, пока не доказана невиновность.

Откуда вы знаете, что можете доверять удостоверению пользователя? Где производилась аутентификация? Как и где производилась авторизация и применялись правила контроля доступа? А откуда известно, что можно доверять данным? Где данные проверялись или кодировались? Не мог ли кто-то изменять их по дороге?

Эти вопросы можно адресовать любой среде. И необходима высокая уверенность в ответах на них, особенно если решаемая задача предъявляет строгие требования к безопасности.

Вопросы о доверии следует задавать на уровне приложения, а также на уровне условий эксплуатации и исполняющей среды. На какие защитные механизмы вы полагаетесь: брандмауэры и прокси-серверы, фильтрацию, сегментацию сети? Какие уровни изоляции и другие средства защиты предоставляет ОС, виртуальная машина или контейнер? Откуда уверенность?

Доверие – это просто. А так ли это?

Концепция доверия проста. Но это то место, где при проектировании часто допускают серьезные ошибки. Легко сделать неправильные или наивные предположения о доверии к идентификатору пользователя или к данным, передаваемым между системами, между клиентами (любыми клиентами, включая браузеры, рабочие станции, мобильные клиенты и устройства интернета вещей) и серверами, между различными уровнями системы и между службами.

Доверие трудно моделировать и гарантировать в корпоративных системах, соединенных со многими другими системами и службами. И, как мы увидим в главе 9, доверие пронизывает и опутывает современные архитектуры, основанные на микросервисах, контейнерах и облачных платформах.

Границы доверия становятся особенно расплывчатыми в мире микросервисов, где один вызов может распространяться по длинной цепочке сервисов. Владелец каждого сервиса должен понимать и поддерживать предположения о доверии к данным и личностям. Если не все будут придерживаться общих паттернов, то легко нарушить отношения доверия и другие контракты между микросервисами.

В облаке необходимо четко понимать модель общей ответственности, предлагаемую поставщиком облачных служб: какие службы вам предоставляются и как безопасно пользоваться этими службами. Если вы доверяете поставщику облачных служб управление удостоверениями и доступом, или аудит, или шифрование, то тщательно ознакомьтесь с соответствующими службами и убедитесь, что они сконфигурированы и используются правильно. Затем напишите тесты, подтверждающие правильность конфигурации и использования. Убедитесь, что понимаете риски и угрозы, которые несут соарендаторы, и умеете безопасно изолировать свои системы и данные. А если вы реализуете гибридную архитектуру, сочетающую публичные и частные облака, то внимательно проанализируйте все данные и удостоверения, передаваемые между частным и публичным облачным доменом.

Ошибки, касающиеся доверия, стоят на первом месте в списке 10 главных изъянов проектирования безопасности ПО по версии IEEE[1]. Там дается такой совет: «Можешь завоевать доверие, можешь оказать доверие, но никогда не доверяй с чужих слов».

[1] Центр безопасного проектирования IEEE «Avoiding the Top 10 Software Security Design Flaws», 2014 (*https://www.computer.org/cms/CYBSI/docs/Top-10-Flaws.pdf*).

Разработчикам трудно понять угрозы, и всем без исключения трудно оценить их количественно. Доверие – более простая концепция, поэтому с размышлений о доверии и стоит начать анализ проекта на предмет уязвимостей. В отличие от угроз, доверие можно верифицировать: внимательно просмотрите проект или код и напишите тесты для проверки своих предположений.

Доверенный и достойный доверия

Термины *доверенный* (trusted) и *достойный доверия* (trustworthy) часто используются как синонимы, но в области безопасности различия между ними весьма существенны, а широко распространенное неправильное употребление иллюстрирует более глубокие недоразумения, бытующие в этой сфере. Вполне возможно, что корни недоразумения уходят в знаменитую Оранжевую книгу (Критерии оценки безопасности доверенных компьютерных систем) (*https://en.wikipedia.org/wiki/Trusted_Computer_System_Evaluation_Criteria*), подготовленную Министерством обороны США в начале 1980-х годов.

Если нечто называют *доверенным*, то имеется в виду, что кто-то этому доверяет, – и ничего больше. Если же нечто называют *достойным доверия*, то утверждается, что субъект заслуживает доверия к себе. Это совершенно разные понятия, имеющие различные свойства в контексте безопасности.

Часто можно встретить рекомендации не открывать вложения в сообщения от *недоверенных* источников и не переходить по *недоверенным* ссылкам, хотя на самом деле имеются в виду источники и ссылки, *не достойные доверия*.

При обсуждении пользователей и систем в терминах моделей угроз важно понимать разницу между доверенными сущностями (им доверяют вне зависимости от того, достойны они доверия или нет) и достойными доверия (степень доверия к ним оценена, а еще лучше – доказана).

Все это может показаться грамматическим педантизмом, но различия между доверенными и достойными доверия субъектами могут оказать далекое от голой грамматики влияние на ваши подходы к безопасности.

Брайан Слеттен выложил коротенькое, но весьма поучительное видео на тему различий между этими словами в контексте безопасности в браузере. Оно находится по адресу *https://www.oreilly.com/learning/trusted-vs-trustworthy*, и мы рекомендуем уделить ему 10 минут.

Построение модели угроз

Для построения модели угроз следует начать с изображения картины всей системы или ее части, показав стрелками входные и выходные потоки данных и выделив места, где создаются, обновляются, передаются или хранятся конфиденциальные данные, включая учетные и персональные данные. Самый распространенный и естественный способ сделать это – нарисовать диаграмму потоков данных с аннотациями.

Описав основные компоненты и потоки, можно заняться определением границ доверия. Принято рисовать их штриховыми линиями, ограничивающими зоны доверия, как показано на рис. 8.1.

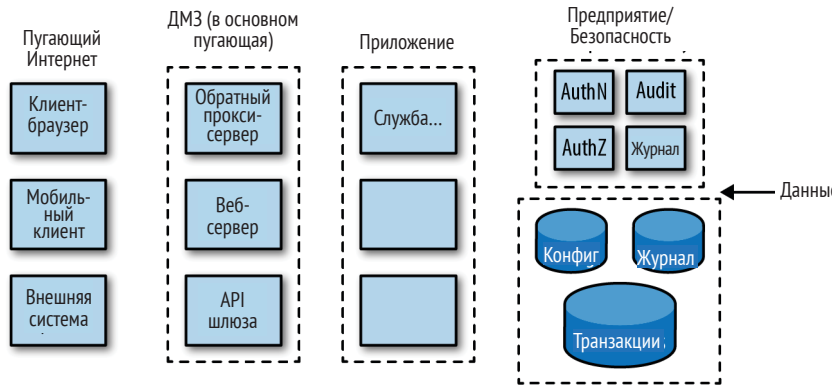

Рис. 8.1. Простая модель границ доверия/угроз

Зоны доверия можно устанавливать между организациями, между ЦОДами или сетями, между системами, между уровнями приложения, а также между службами или компонентами внутри одной службы – в зависимости от того, на каком уровне вы работаете.

Затем покажите средства контроля и проверки на границах:

- где производится аутентификация;
- где производится контроль доступа;
- проверка корректности данных;
- криптография – какая информация шифруется, где, когда;
- ограничение скорости;
- датчики и детекторы.

«Достаточно хорошо» – и достаточно

Не тратьте время на рисование красивых картинок. Модель может быть грубой, особенно в начале. Как всегда при гибком моделирова-

нии, нужно сделать ровно столько, чтобы все понимали, из чего состоит система и как ее части стыкуются; все сверх того – пустая трата времени. Для версии 0.1 сойдет нарисованная на доске диаграмма, сфотографированная на мобильник.

Бесплатный инструмент моделирования угроз от Microsoft

Microsoft предлагает бесплатную программу Microsoft Threat Modeling Tool 2016, которая позволяет создавать модели угроз и сохранять их, так чтобы модель можно было пересмотреть и обновить после внесения изменений в систему.

Как и у всех бесплатных инструментов, у этой программы есть достоинства и недостатки. Программа позволяет легко перетаскивать компоненты по диаграмме, не заставляя помнить, какой конкретно символ должен присутствовать на каждом этапе. Имеется также обширная палитра символов и примеров компонентов.

Но есть и проблемы. Сильно ориентированный на Microsoft язык и предлагаемый набор компонентов затрудняют решение о том, что должно быть на диаграмме. Часто существуют десятки компонентов с одинаковым символом, но разными названиями в меню.

Кроме того, в программу встроен генератор отчетов, который экспортирует оценку угроз на основе модели (и атрибутов, заданных для каждого компонента). На первый взгляд, замечательно, но на практике может вылиться в сотни угроз общего вида в формате HTML, которые еще нужно рассортировать и оценить, а только потом использовать. На сортировку этих отчетов часто уходит гораздо больше времени, чем на ручную оценку угроз, а без сортировки оценка может оказаться впечатляющей, но абсолютно беспредметной.

Мы рекомендуем использовать инструменты для моделирования систем, а оценку угроз производить вручную или вне инструмента, желательно таким способом, который тесно интегрируется с системой учета проблем и конвейером.

Нарисовав достаточно хорошую диаграмму, соберите всю команду, чтобы пройтись по диаграмме и устроить мозговой штурм на тему угроз и атак. Обращайте внимание на информацию, пересекающую границы доверия, особенно секретную. Реалистично относитесь к угрозам и сценариям атак. Пользуйтесь сведениями об угрозах, добытыми в ходе разведки, и концентрируйте внимание на тех способах проведения атак, которые наблюдали в производственной системе. На этом

этапе рекомендуется также выносить за пределы диаграммы вещи, не относящиеся к вашей модели атак, которые вы сознательно хотите вывести за скобки, поскольку это поможет сосредоточиться на том, что действительно имеет значение.

Моделирование угроз будет более эффективным, если удастся привлечь людей с разными точками зрения. Разработчиков, которые знают всю анализируемую систему или ее часть, архитекторов и разработчиков, которые знают другие части системы, тестировщиков, знающих, как система в действительности работает, эксплуатационников и сетевых инженеров, понимающих, в какой среде система исполняется, а также специалистов по безопасности – все они увидят различные проблемы и риски.

Старайтесь не увлекаться спорами о низкоуровневых деталях, если только вы не занимаетесь детальным анализом, когда необходимо проверить все такие предположения.

Еще о моделировании угроз

На сайте SAFECode имеется 45-минутный бесплатный онлайновый курс по моделированию угроз, содержащий неплохой обзор (*https://training.safecode.org/default.aspx*).

Microsoft популяризировала практику моделирования угроз приложению. Краткое введение в методику (или, по крайней мере, бывшую методику) моделирования угроз в Microsoft имеется в блоге Питера Торра «Guerrilla Threat Modelling (or “Threat Modeling” if you’re American)» (*https://blogs.msdn.microsoft.com/ptorr/2005/02/22/guerrilla-threat-modelling-or-threat-modeling-if-youre-american/*).

Если вы действительно хотите разобраться во всем, что касается моделирования безопасности, то прочитайте книгу Adam Shostack «Threat Modeling: Designing for Security» (издательство Wiley) (*https://threatmodelingbook.com/*).

Шостак был одним из лидеров практики обеспечения безопасности ПО в Microsoft и автором новаторского подхода к моделированию угроз, который нашел продолжение в Microsoft SDL. В его книге детально раскрыта тема моделирования угроз, включая различные методы, игры и инструменты для построения моделей угроз, а также приводится анализ различных типов угроз, которые следует иметь в виду.

Потратьте время на выявление рисков и слабых мест проекта, а только потом переходите к обсуждению решений. Затем подумайте, что

можно было бы изменить, чтобы защитить систему от выявленных угроз. Можно ли переместить ответственность извне границы доверия вовнутрь? Можно ли уменьшить количество связей между модулями или упростить проект либо рабочий процесс, чтобы уменьшить поверхность атаки? Нужно ли передавать информацию через границу доверия или достаточно передать безопасный ключ либо маркер?

Какие изменения можно или должно сделать прямо сейчас, а какие допустимо добавить в журнал пожеланий и запланировать на потом?

Думать как противник

В главе 5 мы предлагали разработчикам надеть черную шляпу и начать думать, как противник, а затем написать истории противника или негативные сценарии. Но как это сделать? Как заставить себя думать, как противник? Что это значит – смотреть на систему глазами противника?

Адам Шостак, ведущий специалист по моделированию угроз, говорит, что нельзя попросить человека «думать, как противник» и ожидать какого-то полезного результата, если этот человек никогда не занимался тестированием на проникновение или не был серьезным хакером. Это все равно, что попросить «думать, как профессиональный шеф-повар». Может быть, вы знаете, как приготовить обед для семьи и друзей, и, возможно, даже изрядно поднаторели в этом деле. Но это вовсе не значит, что вы понимаете, как выглядит работа в ресторане с мишленовскими звездами или труд повара в заведении быстрого питания в утренние часы пик[2].

Чтобы понять, как думает и работает атакующий, нужно иметь практический опыт и контекст. Кое-чему со временем можно научиться, наблюдая за работой хорошего тестировщика проникновения (или работая в составе команды красных или в программе вознаграждения за обнаруженные уязвимости) и пройдя курс обучения. Использование инструментов тестирования на проникновение типа OWASP ZAP в своих программах тестирования позволяет через щелочку заглянуть в мир атакующего. Участие в конкурсах «Захвати флаг» и практические учебные курсы по эксплуатации уязвимостей помогут познакомиться с тем, как хакеры думают о мире. Ну а в краткосрочной перспективе придется положиться на специалиста по безопасности (если он есть под рукой), который сыграет роль противника.

[2] Почитайте книгу Bill Buford «Heat: An Amateur's Adventures as Kitchen Slave, Line Cook, Pasta-Maker, and Apprentice to a Dante-Quoting Butcher in Tuscany» (издательство Vintage), чтобы почувствовать, каково это.

STRIDE – структурная модель для лучшего понимания противника

Даже если вы не знаете, как работают атакующие, будет разумно применить структурную модель, чтобы лучше запомнить, как защищаться от различных угроз и рисков безопасности. Один из самых известных подходов такого рода – модель *STRIDE* от Microsoft.

Акроним STRIDE расшифровывается в следующей таблице.

	Угроза	**Решение**
Подлог (Spoofing)	Можно ли доверять удостоверению пользователя?	Аутентификация и управление сеансами, цифровые сертификаты
Манипулирование (Tampering)	Можно ли доверять целостности данных? Мог ли кто-то изменить данные, сознательно или случайно?	Цифровые подписи, контрольные суммы, учет последовательности, контроль доступа, аудит
Отрицание (Repudiation)	Можно ли доказать, что некто выполнил определенное действие, и установить, когда это было сделано?	Сквозной аудит и протоколирование, учет последовательности, цифровые сертификаты и цифровые подписи, предотвращение мошенничества
Разглашение информации (Information Disclosure)	Каков риск кражи или утечки данных?	Контроль доступа, шифрование, преобразование данных в маркеры (data tokenization), обработка ошибок
Отказ от обслуживания (Denial of service)	Может ли кто-то помешать авторизованным пользователям получить законный доступ к системе – сознательно или случайно?	Ограничение темпа, защита границ, эластичное выделение ресурсов, службы доступности
Повышение привилегий (Elevation of Privilege)	Может ли непривилегированный пользователь получить привилегированный доступ к системе?	Авторизация, выделение наименьших необходимых привилегий, контроль типобезопасности и параметризация для предотвращения атак внедрением

STRIDE – простой способ проанализировать самые распространенные пути получения доступа к системам и данным, применяемые атакующими. Существуют и другие, не менее эффективные подходы, на-

пример деревья атак, которые были рассмотрены в главе 5. Важно лишь договориться о каком-нибудь простом, но эффективном способе рассуждать об угрозах и методах защиты от них на этапе проектирования.

Не забывайте о случайных ошибках

Думая о том, что может пойти наперекосяк, не забывайте о случайных ошибках. Ошибка, совершенная без злого умысла, может причинить огромный вред системе, особенно если ее допустил сотрудник службы эксплуатации или привилегированный пользователь приложения (см. обсуждение посмертного анализа без поиска виновных в главе 13).

Инкрементное моделирование угроз и оценка рисков

Как согласовать моделирование угроз с гибкими спринтами или непрерывным потоком в бережливых методиках? Когда следует остановиться и заняться моделированием угроз, чтобы заполнить пробелы в проекте (например, в ответ на отзывы пользователей), в условиях, когда не существует явного подписания проектных документов, поскольку проектирование «никогда не кончается» и может вообще не существовать никакого проектного документа, т. к. «код и есть проект»?

Оценка рисков в самом начале

Можно начать заранее, понимая, что даже если имеется только грубый набросок архитектуры, который еще много раз будет изменяться, все равно команда должна определиться с набором инструментов и стеком времени выполнения, иначе она просто не сможет приступить к работе. Даже имея лишь смутное и неполное представление о том, что предстоит сделать, мы можем начать думать о рисках и слабых местах проекта и о том, как с ними бороться.

Например, в PayPal каждая команда, приступая к работе над новым приложением или микросервисом, должна провести начальную оценку рисков, заполнив автоматизированную анкету. Одно из ключевых решений – будет ли команда использовать языки и каркасы, уже одобренные группой безопасности в каком-нибудь другом проекте. Или же она собирается применить в организации технологии, которые группа безопасности раньше не проверяла? С точки зрения риска, существует большая разница между «просто еще одно мобильное или веб-прило-

жение» на одобренной платформе и техническим экспериментом с новыми языками и инструментами.

Перечислим некоторые вопросы, которые нужно понимать и оценивать в случае анализа рисков в начале проекта.

1. Понимаете ли вы, как следует безопасно использовать язык(и) и каркасы? Какие меры защиты предлагает каркас? Что предстоит добавить, чтобы разработчикам было проще «поступать правильно» по умолчанию?
2. Существуют ли для этого языка и каркаса средства поддержки непрерывной интеграции и непрерывной поставки, включая инструменты статического анализа кода и средства управления зависимостями, позволяющие обнаруживать уязвимости в сторонних библиотеках и библиотеках с открытым исходным кодом?
3. Используются ли в системе секретные или конфиденциальные данные? Какие данные необходимы, как они обрабатываются и что следует подвергнуть аудиту? Требуется ли хранить данные, и если да, то как? Следует ли подумать о шифровании, маркерах и маскировании, контроле доступа и аудите?
4. Границы доверия между данным приложением (службой) и другими: где должны быть размещены средства аутентификации, контроля доступа и проверки качества данных? Какие предположения делаются на верхнем уровне проекта?

Пересмотр угроз при изменении проекта

Затем моделирование угроз следует проводить при каждом существенном изменении поверхности атаки системы, как уже было объяснено раньше. Моделирование угроз следует включить в критерий приемки истории в момент ее первоначального написания, или детализации в процессе планирования спринта, или когда члены команды работают над историей, или на этапе тестирования кода и выявления рисков.

В гибких средах и при использовании DevOps это означает, что угрозы приходится моделировать гораздо чаще, чем при следовании каскадной модели. Но зато это должно быть проще.

В начале проекта, когда только закладывается архитектурная основа системы и начинают наполняться содержанием проект и интерфейсы, поверхность атаки будет изменяться очень часто – это не удивительно. Вероятно, будет трудно понять, на какие риски обращать особое внимание, поскольку так много всего нужно учесть и поскольку проект пре-

терпевает постоянные изменения, по мере того как мы больше узнаем о предметной области и выбранной технологии.

Можно было бы отложить моделирование угроз до тех пор, пока детали проекта не прояснятся, понимая, что мы берем на себя технический долг и долг безопасности, которые впоследствии придется оплатить. А можно продолжать итеративное уточнение модели, чтобы не упускать контроль над рисками, понимая, что, по крайней мере, часть этой работы пойдет насмарку, когда в проект будут внесены изменения.

Позже, когда изменения станут целенаправленными и не столь массивными, моделировать угрозы будет проще и быстрее, поскольку можно сосредоточиться на контексте и последствиях каждого изменения. Станет легче выявлять риски и проблемы. И тем проще и быстрее пойдет процесс, чем чаще команда совершает эти действия и пересматривает модель. В какой-то момент моделирование угроз превратится в рутину и станет неотъемлемой частью мышления и работы команды.

Ключом к моделированию угроз в гибких методиках и DevOps является осознание того факта, что поскольку проектирование, кодирование и развертывание выполняются в одном итеративном цикле, в том же цикле следует производить и оценку технических рисков. Это означает, что мы можем – и должны – сделать моделирование угроз эффективной, простой, прагматичной и быстрой процедурой.

Получение выгоды от моделирования угроз

Команда должна решить, в каком объеме необходимо производить моделирование угроз и сколько на это тратить времени, принимая во внимание соотношение между риском и временем. Как и в случае других гибких инспекций, команда может принять решение об ограничении времени на обсуждения модели угроз, чтобы сделать работу более предсказуемой и сосредоточенной, даже если это означает, что инспекция не будет полной и всесторонней.

Заниматься моделированием угроз на скорую руку часто все же гораздо лучше, чем не заниматься им вовсе.

Как и инспекция кода, моделирование угроз необязательно должно быть вселяющей ужас, дорогой и тяжеловесной процедурой, да оно и не может быть таковой, если мы хотим, чтобы гибкая команда осуществляла поставку с высокой скоростью. Не важно, какой подход вы применяете: STRIDE, CAPEC (*https://capec.mitre.org/*), DESIST[3], деревья

[3] Вариант STRIDE; dispute (оспаривание), elevation of privilege (повышение привилегий), spoofing (подлог), information disclosure (раскрытие информации), service denial (отказ от обслуживания), tampering (манипулирование).

атак (см. главу 5), убийственную цепочку компании Lockheed Martin (*https://www.lockheedmartin.com/en-us/capabilities/cyber/cyber-kill-chain.html*) или просто тщательную проверку предположений о доверии в своем проекте. Важно отыскать такой подход, который команда понимает и которому готова следовать, тогда он станет реальным подспорьем в поиске – и разрешении – проблем безопасности при проектировании.

Итеративный подход к проектированию и разработке позволяет вернуться назад и пересмотреть проект и модель угроз, быть может, даже неоднократно, поэтому не нужно изо всех сил стараться сделать все правильно и окончательно при каждом пересмотре. Подходите к моделированию угроз так же, как к проектированию и написанию кода. Немножко попроектируйте, потом уменьшите поверхность атаки в ходе моделирования угроз, напишите код и тесты. Снова вернитесь к проекту и снова взгляните на модель угроз. Моделирование угроз не кончается никогда.

Минимальный жизнеспособный процесс (и его бумажное оформление)

Типичная ошибка, которую мы часто встречаем, работая с командами, которые впервые занимаются моделированием угроз, – это чрезмерное внимание к документации, бумажному оформлению и вообще к процессу, а не к результату.

Велико искушение заставить документировать каждое изменение по какому-то шаблону и заполнять «простенькую» форму оценки угроз для каждой новой функции. Но на самом деле даже облегченные процессы иногда отвлекают нас от истинного смысла деятельности. Если вы замечаете, что команда тратит больше сил на рисование диаграммы или подготовку документа, а не на осмысление результатов оценки, то пересмотрите свой подход.

Даже в условиях, когда требуется строгое соответствие нормативным требованиям, требования к процессу и документации оценки угроз должны быть ограничены минимумом, необходимым для аудита, а не становиться основной целью работы.

Помните, что в основном мы инженеры и созидатели. Пользуйтесь своими умениями и навыками, чтобы автоматизировать процесс и подготовку документации, а свое время тратить на оценку, поскольку только она и представляет ценность.

Всякий раз, возвращаясь к проекту и к внесенным в него изменениям, вы располагаете новой информацией и новым опытом, а это значит, что можете задать новые вопросы и обнаружить проблемы, мимо которых прошли раньше.

Акцентируйте внимание на высокорисковых угрозах, особенно активных. И не забывайте о широко распространенных атаках.

Типичные векторы атак

Знание основных видов атак, от которых следует защищаться, и технических деталей их проведения поможет наполнить абстрактные риски и угрозы конкретным смыслом, понятным разработчикам и эксплуатационникам. Это также поможет им более реалистично подходить к обдумыванию угроз и сосредоточиться на защите от тех сценариев и уязвимостей, которые противник может попробовать в первую очередь.

Существует несколько способов узнать о типичных атаках и о том, как они работают.

Сканирование

Инструменты типа OWASP ZAP или службы сканирования типа Qualys или WhiteHat Sentinel, которые автоматически проводят типичные атаки против вашего приложения.

Результаты тестирования на проникновение

Потратьте время на то, чтобы понять, что и почему проверяли тестировщики проникновения и что они нашли.

Вознаграждение за найденные уязвимости

В главе 12 мы увидим, что программы вознаграждения за обнаружение уязвимостей могут быть полезным способом расширить покрытие проблем безопасности тестами, пусть даже он не всегда дает эффект.

Игры «красные против синих»

В главе 13 описано, как обороняющаяся команда синих ведет активное наблюдение за реальными атаками и учится защищаться от них.

Платформы имитации атак

Платформы AttackIQ (*https://www.attackiq.com/*), SafeBreach (*https://safebreach.com/*) и Verodin (*https://verodin.com/*) автоматически проводят типичные атаки на сеть и на приложение и позволяют написать скрипт, имитирующий собственную атаку или реакцию на

нее. Это черные ящики, которые проверяют ваши защитные меры и наглядно показывают, как будет скомпрометирована ваша сеть при различных условиях.

Отраслевые аналитические отчеты

В таких отчетах, как ежегодный «Отчет об исследовании уязвимостей данных» компании Verizon (*http://www.verizonenterprise.com/verizon-insights-lab/dbir/*), приводится обзор наиболее распространенных и серьезных атак, наблюдавшихся на практике.

10 главных рисков OWASP (*https://www.owasp.org/index.php/Main_Page*)

В этом документе перечислены самые серьезные риски и наиболее распространенные атаки на мобильные и веб-приложения. Для каждого из 10 главных рисков OWASP рассматривает потенциальных злоумышленников (на какого рода пользователей обращать внимание), типичные векторы атак и примеры сценариев атак. Мы еще вернемся к этому документу в главе 14, поскольку приведенный в нем перечень многие регулирующие органы сделали эталонным требованием.

Журналы событий в ваших системах

Содержат признаки реальных атак и сведения о том, были ли они успешны.

Типичные атаки следует учитывать при проектировании, тестировании, проведении инспекций и организации мониторинга. Свое понимание атак можно отразить в историях, касающихся безопасности, при написании автоматизированных тестов безопасности, при проведении совещаний по моделированию угроз и для определения сигнатур атак в системах мониторинга и защиты.

Сухой остаток

Для эффективной защиты от противников и создаваемых ими угроз системе и организации следует предпринять следующие меры.

- Включить в циклы обратной связи гибких методик и DevOps разведку угроз.
- Один из лучших источников разведки угроз – наблюдение за тем, что происходит в производственных системах прямо сейчас. Используйте эту информацию в методологии защиты в ответ на атаку, чтобы найти и закрыть бреши раньше, чем их сумеет найти и эксплуатировать противник.

- Гибкие команды редко тратят много времени на начальное проектирование, но тем не менее, прежде чем команда окончательно выберет технологическую платформу и среду исполнения, обсудите с ней риски безопасности, имеющиеся в предполагаемом подходе, и способы их минимизации.
- В гибкой среде, а особенно в среде DevOps, поверхность атаки системы постоянно изменяется. Понимая поверхность атаки и влияние на нее вносимых изменений, вы сможете выявить риски и потенциальные слабые места, на которые необходимо обратить внимание.
- Поскольку поверхность атаки постоянно изменяется, необходимо столь же постоянно заниматься моделированием угроз. Делать это следует инкрементно, итеративно и необременительно.
- Просветите команду о типичных атаках, перечисленных в списке 10 главных рисков OWASP, и объясните, как учитывать их при проектировании, тестировании и мониторинге.

Глава 9

Построение безопасных и удобных для пользования систем

Что значит построить безопасную систему?

Это значит, что при проектировании и реализации системы следует учитывать потенциальные риски и оставлять только приемлемые.

Применимый уровень *безопасности* зависит от организации, отрасли промышленности, клиентов и типа системы. Если разрабатывается онлайновая площадка для торговли потребительскими товарами или мобильная игра, то в расчет нужно принимать совсем не такие соображения безопасности, как при разработке шифровального устройства, которое будет использоваться морскими пехотинцами, находящимися в авангарде.

Однако есть паттерны, общие для большинства ситуаций, связанных с безопасностью.

Проектируйте с защитой от компрометации

При построении безопасной системы следует защищаться от компрометации, будь то атаки внедрением SQL, атаки по энергопотреблению или путем анализа перехваченного спектра электромагнитного излучения. Важно, чтобы вы понимали операционную среду и умели встраивать защиту от компрометации.

Для этого следует изменить предположения, лежащие в основе проекта.

Годы криптологических исследований показали: следует предполагать, что любые данные, введенные пользователем, могут быть ском-

прометированы и что противник может надежно и повторяемым образом внедрить в систему любые данные, какие захочет. Следует также предполагать, что противник может надежно читать все, что появляется на выходе системы.

Можно даже зайти еще дальше и предполагать, что противник знает все о вашей системе и о том, как она работает[1]. Так что же мы в результате имеем?

Защита от компрометации – это про тщательную проверку всех входных данных и вывод лишь такого объема информации, без которого нельзя обойтись. Это про предвидение отказов и их безопасную обработку, про обнаружение и протоколирование ошибок и атак. Про то, как максимально затруднить жизнь противнику, не принося в жертву понятность системы и удобство пользования ей.

Почти все мы можем предполагать, что ресурсы и мотивация противника ограничены. То есть если мы будем сопротивляться достаточно долго, то у противника кончатся время и деньги, ему надоест, и он обратит свои взоры на кого-то другого. (Если вы не из этого лагеря, то вам нужна помощь специалиста, а эту книгу, пожалуй, читать не следует!) Если атаковать вас вознамерилось государство, то, скорее всего, своей цели оно добьется, но все равно вы должны затруднить им работу и постараться застать на месте преступления.

Безопасность и удобство пользования

Часто можно услышать мнение, что безопасность и удобство пользования – противоположные стороны одной медали. Что забота о безопасности заставляет нас создавать неудобные системы, а специалисты по удобству пользования хотели бы, что мы убрали все защитные механизмы.

Но постепенно приходит понимание, что это неверно. Это узкое восприятие безопасности как технической, а не всеобъемлющей задачи.

Системы, которыми неудобно пользоваться из-за перегруженных инженерными излишествами средств безопасности, заставляют пользователей искать обходные пути, ставящие под угрозу безопасность: записывать пароли на бумажках, заводить общие для всей команды учетные записи, так что администратор не может понять, кто виноват, когда такую запись взламывают.

Мы, специалисты по безопасности, должны воспринимать создаваемую систему во всей целостности и делать все возможное для того,

[1] Этой идее уже больше 100 лет, ее обычно формулируют в виде афоризма Клода Шеннона «враг знает систему».

чтобы пользователю было удобно работать с системой максимально безопасным образом. Принимайте любые меры, какие считаете нужными для защиты системы, но при этом не мешайте использовать ее для того, для чего она задумана, и так, как ее предполагалось использовать.

Технические средства контроля

Говоря о средствах обезопасить систему, многие представляют себе именно технические средства. Это решения, мешающие противнику (или пользователю) получить неавторизованный доступ к функциональности, которая предназначена не для него, или к механизмам управления системой.

Технические средства контроля часто «прикручены» к проекту системы для решения проблем безопасности. Им свойственна двоякая проблема: с одной стороны, это место будто медом намазано для различных поставщиков решений, а с другой – сфера их применения сильно ограничена.

Поставщики сплошь и рядом соловьем заливаются, расписывая прелести своего черного ящика, который защитит вашу систему от всех напастей. Но спросите себя: неужели вы думаете, что все те организации, что пострадали от взлома за последний год, не использовали брандмауэров нового поколения, систем обнаружения вторжений, передовой защиты оконечных точек и прочих волшебных черных ящиков?

При выборе технических средств контроля толика здорового скептицизма не повредит. Хотя это важная часть архитектуры безопасности, в большинстве случаев они защищают только лежащее на поверхности (типичные уязвимости) или предназначены для весьма специфических пограничных случаев. Точно так же, как запертая дверь не защитит ваш дом от всех воров, технические средства контроля не защитят систему от всех угроз. Их следует включать в состав более широкого, целостного решения.

Мы выделяем несколько категорий средств контроля безопасности. Как во всех системах классификации, наши категории частично перекрываются и далеки от совершенства, но и этого достаточно, чтобы начать разговор на эту тему.

Сдерживающие средства контроля

Сдерживающие средства контроля ясно дают понять, что произойдет при попытке атаки на систему. Это технический аналог таблички на воротах «Осторожно, злая собака» или «Это здание находится под охраной».

Физически это могут быть и другие хорошо заметные средства, поэтому системы охранного видеонаблюдения или ограждения, находящиеся под током или сделанные из колючей проволоки, являются не только высокоэффективными средствами противодействия, но и играют роль сдерживающих.

В цифровой системе можно говорить о включении в заголовки служб предупреждений, а также о предупредительных сообщениях и специализированной обработке ошибок. Видимое присутствие такого рода вещей ясно показывает, что вы знаете, что делаете, и дает противнику знать, что он пересекает красную линию.

Средства противодействия

Средства противодействия предназначены для того, чтобы замедлить противника, но не остановить его. К ним можно отнести ограничение количества попыток входа с одного адреса или ограничение темпа работы с системой. Умный противник сможет обойти эти средства, но они замедлят его продвижение, повысив шансы быть пойманным, и, возможно, заставят выбрать другую жертву.

Поскольку цель этих средств – замедлить и обескуражить противника, мы можем включить в эту категорию также запутывание кода, неконкретные сообщения об ошибках и адаптивное управление сеансами.

Запутывание кода как средство противодействия вызывало оживленные дебаты, поэтому рассмотрим, во что обходятся такие меры. После пропускания кода через программу запутывания его становится трудно читать, инспектировать и понимать. Хотя это действительно способно замедлить не слишком квалифицированного противника, не забывайте, что у всего есть цена. Запутанный код часто трудно отлаживать, и он может загнать в ступор группу поддержки.

Средства противодействия полезны, но применять их нужно с осторожностью. Замедление и обескураживание противника редко бывает приемлемым, если одновременно обескураживает и мешает работать законным пользователям системы.

Защитные средства контроля

Защитные средства фактически предотвращают атаку. К этому виду относится большинство технических средств безопасности, в т. ч. брандмауэры, списки контроля доступа (ACL), ограничения на IP-адреса и т. п.

Некоторые защитные средства, например открытие доступа к системе только в определенное время или разрешение доступа только с опре-

деленных рабочих мест, могут резко отрицательно сказаться на работе с ней. И хотя бывает, что для таких мер у предприятия есть обоснование, всегда следует учитывать сопутствующие им неудобства.

Например, запрет входа в систему не в рабочее время или не из офиса может помешать сотрудникам проверять свою почту, когда они находятся на конференции. Люди (включая и противника) относятся к защитным мерам, как вода, встречающая препятствие на своем пути. Они найдут способ обойти их и придумают прагматичное решение, позволяющее двигаться дальше.

Детекторные средства контроля

Некоторые средства контроля предназначены не для того, чтобы остановить или хотя бы замедлить противника, а просто для обнаружения факта вторжения. Это может быть простой контрольный журнал, или средство регистрации событий безопасности, или более продвинутые механизмы: медовые ловушки, графы потока трафика и даже мониторинг аномальной загруженности процессора.

Детекторные средства контроля широко распространены и знакомы эксплуатационникам и отделам ИТ-инфраструктуры. Основные проблемы состоят в том, чтобы настроить их на обнаружение вещей, важных для конкретной среды, и назначить лицо, которое будет просматривать журналы и реагировать, если что-то будет обнаружено.

Компенсационные средства контроля

Наконец, бывает, что нужного вам средства контроля нет. Так, если хочется запретить доступ к системе из-за границы, но большая часть работников постоянно находится в разъездах, то мы оказываемся в тупике.

В таком случае часто применяются другие средства контроля, компенсирующие отсутствие того, что нужно. Например, в описанной ситуации можно использовать физический предмет, играющий роль второго фактора аутентификации, и следить за различием между фактическим местонахождением работника и указанным в его командировочном удостоверении, чтобы обнаружить попытки злонамеренного использования.

Другие примеры компенсационных средств контроля будут рассмотрены в главе 13 при обсуждении таких механизмов, как WAF, RASP и защиты исполняющей среды.

Архитектура безопасности

Ну и что дальше? Как же нам все-таки построить безопасную систему?

Мы можем использовать средства контроля в сочетании с определенными принципами, чтобы построить систему, которая противодействует противнику и активно защищается от атаки.

С самого начала мы должны знать, что система делает и кто может ее атаковать. Хочется надеяться, что, последовав рекомендациям из главы 7, касающимся оценки угроз, вы поняли, кто является вероятным противником, за чем он охотится и к каким способам атаки, скорее всего, прибегнет.

Затем мы должны подумать о самой системе. Мы часто говорим о системе как о едином монолитном черном ящике. Это наследие традиционного мышления системного инженера. Когда-то *давным-давно* для построения системы нужно было покупать достаточно много компьютеров, поэтому на физическое оборудование приходилась значительная часть общей стоимости.

В наши дни, когда контейнеризация, микросервисы, виртуальные машины и облачные вычисления обходятся дешево, а будут обходиться еще дешевле, можно начать проводить разделительную черту между компонентами системы.

С точки зрения безопасности, это выглядит гораздо лучше. О каждом компоненте становится проще рассуждать, и его нормальное функционирование можно обезопасить. Компоненты зачастую меньше и «заточены» под одну функцию, поэтому развертывание, изменение и управление ими можно осуществлять независимо.

Однако при этом возникают новые риски и вызовы безопасности.

Традиционная модель безопасности побуждает нас рассуждать о системе как об аналоге конфет M&M или Smartie (в зависимости от того, где вы росли – в Америке или в Британии) или, если речь идет о Новой Зеландии, об аналоге *броненосцев* (наверное, потому что жизнь новозеландцев ближе к дикой природе, а не к конфетам, или потому что они считают, что дикая природа и есть конфетка). У всего вышеназванного есть твердая внешняя оболочка и мягкое липкое содержимое.

Безопасность без периметра

Исторически система с архитектурой типа M&M (или броненосца) имела всего несколько периметров, или границ доверия. Все находящееся вне границы доверия (например, сторонние системы или пользователи) считалось недоверенным, а все системы и сущности внутри гра-

ницы доверия или укрепленного периметра, за ДМЗ и брандмауэрами, рассматривались как безопасные.

В предыдущей главе, посвященной угрозам и атакам, мы видели, что понятия доверия и границ доверия в монолитной системе, целиком расположенной на территории предприятия, довольно просты. В этой модели внутри периметра всего несколько сущностей или компонентов, и нет никаких причин считать их риском.

Но слишком многое зависит от небольшого числа периметров. Стоит противнику проникнуть внутрь периметра, как ему становится открыто всё.

По мере того как архитектура распадается на слабо связанные части, мы должны изменить способ мышления. Поскольку каждая сущность управляется отдельно, мы больше не можем верить, что прочие компоненты системы не скомпрометированы. Это также означает, что мы больше не вправе предоставить администраторам внутренних сетей неограниченный доступ ко всей системе (конец «режиму Бога» и черным ходам для поддержки).

Вместо этого системы следует строить, не предполагая, что какие-либо пункты физической или виртуальной сети достойны доверия. Иными словами, уровень доверия к сети низкий, или, как теперь говорят, это *сеть с нулевым доверием*.

Чтобы больше узнать, как проектировать и эксплуатировать сети с нулевым доверием, почитайте об инициативе Google BeyondCorp (*http://static.googleusercontent.com/media/research.google.com/en//pubs/archive/43231.pdf*).

А также книгу Evan Gilman, Doug Barth «Zero Trust Networks: Building Trusted Systems in Untrusted Networks» (издательство O'Reilly).

В этой среде все, что находится в сети, следует защищать от злонамеренных инсайдеров или противников, преодолевших периметр или другие средства защиты.

- Переоценивайте и подвергайте удостоверение аудиту в каждой точке. Вы всегда должны знать, с кем имеете дело. Для доказательства подлинности удостоверения можно использовать маркеры с ограниченным сроком действия, выпущенные сервером аутентификации, криптографически безопасные ключи и другие подобные методы.

- Рассмотрите использование протокола TLS для организации защищенной связи между службами, по крайней мере с граничными службами, службами аутентификации и другими особо ответственными службами.
- Повторно проверяйте входные данные, даже от основных служб системы и платформенных служб. Это значит, что нужно проверять все заголовки и все поля каждого запроса.
- В каждой точке применяйте правила контроля доступа к данным и функциям API. Сделайте эти правила настолько простыми, насколько возможно, но применяйте их последовательно и следите за соблюдением принципа минимальных привилегий.
- Рассматривайте все закрытые данные (любые данные, которые кому-то может захотеться украсть) как токсичные[2]. Вы должны всегда знать, где они находятся и кто ими владеет, обращаться с ними безопасно, осторожно подходить к вопросу о том, с кем вы ими обмениваетесь и как храните (если хранить их обязательно), и при любой возможности шифровать.
- Протоколируйте трафик в каждой точке – не только в брандмауэре периметра – чтобы можно было определить, где и когда произошла компрометация. Помещаемые в журнал записи следует направлять безопасной центральной службе протоколирования, чтобы можно было проследить запросы, которыми обмениваются службы, и защитить журналы на случай компрометации службы или узлы.
- Укрепляйте все исполняющие среды (ОС, виртуальные машины, контейнеры, базы данных), как если бы эти машины находились в демилитаризованной зоне.
- Применяйте *прерыватели* и *переборки*, чтобы ограничить распространение отказов исполняющей среды и минимизировать «радиус поражающего действия» уязвимости. Эти паттерны стабильности взяты из книги Michael Nygards «Release It!» (издательство Pragmatic Bookshelf)[3], в которой объясняется, как проектировать, реализовывать и эксплуатировать отказоустойчивые онлайновые системы.
- Переборками можно окружать соединения, пулы потоков, процессы и данные. Прерыватели защищают вызывающую сторону

[2] См. статью в блоге Брюса Шнайера «Data Is a Toxic Asset» (*https://www.schneier.com/blog/archives/2016/03/data_is_a_toxic.html*) от 4 марта 2016.

[3] Майкл Нейгард «Release it! Проектирование и дизайн ПО для тех, кому не все равно», Питер 2016.

от сбоев вызываемой стороны, поскольку автоматически обнаруживают тайм-ауты и зависания и восстанавливают работоспособность. Система Netflix Hystrix (*https://github.com/Netflix/Hystrix*) – отличный пример реализации прерывателя.

- Пользуйтесь контейнерами для управления и защиты служб. Хотя контейнер не обеспечивает такого же уровня изоляции на этапе выполнения, как виртуальная машина, он и «весит» значительно меньше, поскольку контейнер можно – и должно – упаковать и настроить, включив лишь минимальный набор зависимостей, необходимых для конкретной службы, что сокращает общую поверхность атаки сети.
- Будьте предельно осторожны при обращении с закрытыми ключами и другими секретами. Подумайте об использовании безопасной службы управления ключами типа AWS KMS (*https://aws.amazon.com/ru/kms/*) или универсального диспетчера секретной информации типа Vault компании Hashicorp (*https://www.vaultproject.io/intro/*). См. главу 13.

Предполагайте, что система скомпрометирована

Одно из важных новых веяний – предположение о том, что все доступные из сети службы скомпрометированы. Иными словами, в многоуровневой архитектуре следует предполагать, что следующий за вами уровень уже скомпрометирован.

Можно считать, что и службы предыдущего уровня тоже скомпрометированы, но в большинстве случаев, если нижний уровень скомпрометирован, игра проиграна. Невероятно трудно защититься от атаки, идущей с нижних уровней стека вызовов. Но вы можете и должны попытаться защитить свою службу (и службы, расположенные выше нее) от ошибок и сбоев исполняющей среды на нижних уровнях.

Службы или пользователи, которые могут вызывать вашу службу, находятся на более высоком уровне и потому представляют опасность. Поэтому не имеет значения, пришел ли запрос из Интернета, с рабочей станции персонала, от другой службы или от администратора системы: все они должны рассматриваться как потенциально скомпрометированные.

Как поступать, если мы считаем, что следующий уровень скомпрометирован? Запросить удостоверение вызывающей стороны. Тщательно проверять входные данные, стараясь отсечь все, что пытается проникнуть на наш уровень извне. Возвращать только необходимую информа-

цию и ничего лишнего. И аудировать все происходящее в каждой точке: что получили, когда получили, что сделали с этим.

Старайтесь проявлять паранойю, оставаясь практичным. Защитное проектирование и кодирование, разумное отношение к данным, которыми обмениваетесь с другими службами, продумывание того, как ограничить воздействие сбоев исполняющей среды и уязвимостей, – все это делает систему более безопасной и отказоустойчивой.

Сложность и безопасность

Сложность – враг безопасности. С ростом и усложнением системы становится труднее понять ее и обеспечить ее безопасность.

> Невозможно сделать безопасным то, чего не понимаешь.
> – Брюс Шнайер «*Мольба о простоте*»

Гибкие и бережливые методики разработки помогают уменьшить сложность, стремясь сделать проект максимально простым.

Инкрементное проектирование начинается с простейшей работоспособной модели. Бережливые методики настаивают на скорейшей поставке минимального жизнеспособного продукта (MVP). Принцип YAGNI (тебе это не понадобится)[4] напоминает команде, что в ходе реализации нужно сосредоточиться только на том, что необходимо сейчас, и не пытаться предвидеть будущие потребности. Всё это – средства против сложности и проектирования всей системы с самого начала.

Если набор функций сведен к минимуму, а каждая функция максимально проста, то риски безопасности уменьшаются за счет сокращения поверхности атаки приложения.

Со временем сложность все-таки стремится прокрасться в систему. Но благодаря итеративному и непрерывному рефакторингу проект удается вычищать и сохранять простым.

Однако есть разница между не поддающимся дальнейшему упрощению и опасно наивным.

Чистая архитектура с четко определенными интерфейсами и минимальным набором функций – не то же самое, что простецкий и неполный проект, рассчитанный только на то, чтобы побыстрее реализовать функциональность, не обращая внимания на безопасность и конфиденциальность данных или обеспечение защиты от отказов исполняющей среды и атак.

Существует также важное различие между эссенциальной (существенной) и акцидентальной (случайной) сложностью.

[4] *https://martinfowler.com/bliki/Yagni.html*.

Некоторые проблемы проектирования, особенно в области безопасности, действительно трудно решить надлежащим образом: примерами могут служить криптография и распределенное управление удостоверениями. Это эссенциальная сложность, справиться с которой помогает передача работы и рисков на сторону, использование проверенных, достойных доверия библиотек и служб вместо попытки сделать все собственными силами.

Но, как мы увидим в главе 10, есть немало случаев, когда ненужная сложность только вносит ненужные риски. Код, который трудно понять, и код, который невозможно полностью протестировать, – это код, безопасности которому нельзя доверять. Система, которую невозможно повторяемым образом собрать и уверенно развернуть, не может считаться безопасной и защищенной.

Повторим, что многие рассмотренные в этой книге гибкие практики позволяют устранить ненужную сложность и снизить риски:

- разработка через тестирование и проектирование на основе поведения;
- совместное владение кодом и следование наставлениям по кодированию;
- автоматизированное сканирование кода с целью выловить ненадлежащие практики кодирования и запахи кода;
- парное программирование и инспекции кода;
- дисциплинированный рефакторинг;
- непрерывная интеграция.

Все вышеперечисленное делает код проще и безопаснее. Автоматизация сборочного конвейера и развертывания, а также непрерывная интеграция уменьшают сложность и риск поставки и внедрения изменений за счет стандартизации и обеспечения тестопригодности шагов, а также потому, что развертывание небольших инкрементных улучшений и исправлений безопаснее и дешевле, чем капитальная модернизация.

Разбиение крупной системы на меньшие части с четким разделением обязанностей помогает уменьшить сложность, по крайней мере на первых порах. Небольшие службы с одной-единственной функцией понятны с первого взгляда, их легко тестировать отдельно от всего остального и безопасно развертывать автономно. Но если и дальше создавать только небольшие службы, то наступает момент, когда общая сложность системы значительно возрастает, а вместе с ней и риск безопасности[5].

[5] Чтобы разобраться в этих вопросах и понять, как их решать, посмотрите презентацию Лауры «Practical Microservice Security» на конференции NDC 2016 в Сиднее.

На вопрос, как бороться с такого рода сложностью, нет простых ответов. Необходимо, чтобы команды следовали единым паттернам и стратегиям проектирования, применяли общие средства контроля. Необходимо глубокое проникновение во внутреннее устройство и механизмы работы системы, следует регулярно производить переоценку архитектуры на предмет упущений и слабых мест. И, как мы уже объясняли, следует проектировать каждый компонент в предположении, что он будет работать во враждебном, не заслуживающем доверия и непредсказуемом окружении.

Сухой остаток

Для построения безопасной системы необходимо следующим образом изменить проектные предположения.

- Проектируйте систему с защитой от компрометации. Предполагайте, что противник знает все о системе и о том, как она работает. Встраивайте защиту от сбоев, ошибок и атак.
- К архитектуре системы можно «прикрутить» технические средства контроля, которые могут сдержать или оказать противодействие противнику, обнаруживать атаки и защищаться от них или компенсировать слабые места системы. Можно также включить в свой код средства контроля безопасности, пронизывающие всю архитектуру и проект системы.
- Всегда предполагайте, что система скомпрометирована и что на защиту периметра или черные ящики нельзя полагаться.
- Обеспечение безопасности увеличивает сложность проекта по основательным причинам, но излишняя сложность – враг безопасности. Применяйте гибкие принципы и практики для уменьшения сложности проекта и кода. Тогда систему будет проще изменять и безопаснее эксплуатировать.

Глава 10

Инспекция кода в интересах безопасности

Мы обсудили, как учитывать безопасность в процессе планирования, сбора требований и проектирования. Пришло время поговорить о безопасности на уровне кода.

По крайней мере, половина уязвимостей – это результат простых ошибок программирования, небрежности со стороны разработчиков, игнорирования требований, непонимания или неправильного применения языка, библиотек и каркасов.

Есть два основных подхода к обнаружению в коде ошибок, в т. ч. связанных с безопасностью.

Тестирование

: Автоматизированное или ручное, включая сканирование на предмет уязвимости, когда система рассматривается как черный ящик.

Инспекция кода

: Включая парное программирование и дружественную оценку (peer review), аудит кода и автоматизированное сканирование кода.

В следующих двух главах мы рассмотрим сильные и слабые стороны обоих подходов. Но начнем с места инспекции кода в процессе гибкой разработки и обсуждения того, как ее можно использовать для выявления существенных проблем.

Зачем нужна инспекция кода?

Инспекции кода проводят по разным причинам.

Контроль

Дружественная оценка играет важную роль в управлении изменениями, поскольку дает уверенность, что, по крайней мере, еще один человек знает об изменении кода и явно или неявно его одобрил.

Прозрачность

Во время инспекции кода члены команды получают информацию обо всем происходящем в проекте. Это повышает их информированность о том, как работает и как изменяется система. Проливая свет на каждое изменение, инспекция также уменьшает вероятность того, что злонамеренный инсайдер подложил логическую бомбу, оставил черный ход или попытался совершить мошенничество.

Соответствие нормативным требованиям

Инспекция кода может являться частью нормативно-правовых требований, например стандарта PCI DSS.

Единообразие

Инспекция помогает внедрить в команду общие соглашения, стиль и паттерны кодирования.

Обучение

Инспекция кода дает возможность менее опытным членам команды обучаться профессиональным хитростям и тонкостям ремесла. Авторы и инспекторы могут учиться друг у друга.

Обобществление кода

Инспекция кода коллег создает ощущение общего владения кодом. Если инспекции проводятся регулярно, то у разработчиков постепенно исчезает собственническое отношение к своему коду, они начинают более терпимо относиться к изменениям и отзывам.

Ответственность

Зная, что кто-то другой будет пристально и критически оценивать его работу, разработчик начинает относиться к ней более вдумчиво и внимательно, перестает срезать углы, а значит, код становится лучше.

Но для наших целей важнее всего тот факт, что правильно организованные *инспекции кода – эффективный способ отыскания ошибок, в т. ч. уязвимостей*.

Типы инспекций кода

Существуют разные способы инспектировать код:

- формальные инспекции;
- метод утенка (самоинспекция);
- парное программирование;
- дружественная оценка;
- внешний аудит кода;
- автоматизированная инспекция.

Рассмотрим плюсы, минусы и стоимость каждого подхода.

Формальные инспекции

Формальная инспекция кода осуществляется в ходе совещания инспекционной группы (в ее состав входят автор кода, читатель кода, который просматривает код, один или несколько инспекторов и модератор или тренер), которая сидит вокруг стола и внимательно изучает распечатки или код, проецируемый на экран. Такие инспекции все еще практикуются некоторыми командами, особенно при разработке высокорисковых систем, ошибки в которых могут угрожать жизни людей. Но это дорогой и неэффективный способ выявления проблем в коде.

Недавние исследования показывают, что организация и проведение формальных инспекций кода значительно увеличивают время и стоимость разработки, не давая существенного эффекта. На планирование, подготовку бумаг и последующие действия могут уходить часы, совещание включается в график загодя, за несколько недель, но при этом на самом совещании обнаруживается менее 5% дефектов. Все остальное инспекторы находят сами, просматривая код перед совещанием[1].

Метод утенка, или Проверка за столом

В основе «метода утенка» (rubber ducking) лежит идея о том, что если нет никого, кто проинспектировал бы ваш код, то просмотреть его и попытаться объяснить воображаемому собеседнику (или резиновой уточке) все же лучше, чем не инспектировать вовсе.

Самоинспекция не отвечает требованиям контроля, соответствия нормативным требованиям, прозрачности и обобществления кода. Но при условии дисциплинированного проведения она все же может стать эффективным способом нахождения ошибок, включая и уязвимости.

---

[1] См. статью Lawrence G. Votta, Jr. «Does every inspection need a meeting?», 1993.

В исследовании инспекций кода, проведенном в Cisco Systems, разработчики, перепроверяющие свою работу, находили половину дефектов, обнаруженных другими инспекторами, без посторонней помощи.

Перед тем как инспектировать собственный код, сделайте перерыв

Если вы можете себе позволить промежуток в несколько дней между написанием и инспектированием кода, то будет больше шансов увидеть собственные ошибки.

Парное программирование (и программирование толпой)

Парное программирование, когда два разработчика пишут код вместе – один стучит по клавиатуре, а другой направляет и помогает (как водитель и штурман в поездке), – фундаментальный метод экстремального программирования (XP). В случае парного программирования инспекция кода осуществляется сразу и непрерывно: разработчики тесно общаются, делятся мыслями, вместе решают проблемы и помогают друг другу писать более качественный код. Такие организации, как Pivotal Labs, прославились успехами в парном программировании.

Парное программирование подразумевает совместный поиск решения водителем и штурманом. Это замечательный способ ввести в курс дела нового члена команды или отладить заковыристую ошибку. Результатом парного программирования является более чистый и лаконичный код, более четкие абстракции и меньшее количество логических и функциональных ошибок. Но если в паре не участвует специалист по безопасности или технически подкованный разработчик, то код необязательно получится более безопасным.

Но и с парным программированием сопряжены определенные затраты. Хотя две головы, конечно, лучше одной, но занимать двух разработчиков для написания одного и того же кода, очевидно, дорого. Кроме того, парное программирование – это интенсивная деятельность, требующая высокой дисциплины, и многие считают, что она служит причиной социального напряжения и умственного истощения, так что долго работать в таком режиме трудно. Немногие разработчики выдерживают работу в паре более нескольких часов в день или нескольких дней в неделю.

Еще более экстремальный вариант парного программирования – *программирование толпой* (mob programming – http://mobprogramming.org/),

когда вся команда работает над одним куском кода за одним компьютером. Этот подход поощряет сотрудничество, командное решение задач и взаимное обучение. Лишь немногим организациям удавалось таким путем добиваться успехов, но у него все же есть достоинства – полная прозрачность и еще более высокое качество кода.

Дружественная проверка

Облегченная неформальная дружественная проверка кода применяется во многих командах, практикующих гибкие методики и DevOps. Такие организации, как Google, Microsoft, Facebook и Etsy, поощряют и даже настаивают на дружественных проверках до выпуска производственной версии кода. Инспекции кода являются также ключевой частью технологического процесса внесения изменений в большинстве крупных проектов с открытым исходным кодом, таких как ядро Linux, Apache, Mozilla и Chromium.

Инспекция кода производится либо людьми, сидящими рядом (инспекция «через плечо»), либо по электронной почте, либо путем извлечения кода из репозитория Git, либо с применением инструментов коллективной инспекции типа Phabricator, Gerrit, Review Board, Code Collaborator или Crucible. С помощью этих инструментов инспекторы – даже в распределенных командах – могут обмениваться мнениями с автором и друг с другом, комментировать и аннотировать подлежащий изменению код и открывать темы для обсуждения. При этом автоматически создается архив, который можно использовать для контроля и проверки соответствия нормативным требованиям.

Если дружественная проверка кода уже является частью технической культуры и практики команды, то этим можно воспользоваться в программе обеспечения безопасности, обучив команду проверять следование рекомендациям по безопасному кодированию и контролировать риски безопасности и уязвимости.

Аудит кода

Если парное программирование и дружественная проверка обычно являются составными частями повседневной работы по написанию кода, то аудит производится отдельно и вне рамок разработки. В этом случае внешний по отношению к команде специалист по безопасности (или небольшая группа таких специалистов) инспектирует на предмет наличия уязвимостей всю кодовую базу или, по крайней мере, настолько большую ее часть, насколько позволяет отведенное время. Такие ин-

спекции часто являются нормативным требованием или проводятся для управления особо важными рисками.

Аудит кода обычно отнимает несколько дней у инспекторов, а также у команды – чтобы помочь инспекторам понять проект и контекст системы и структуру кода. Затем команда должна потратить дополнительное время, чтобы понять, что нашли (или решили, что нашли) инспекторы, и совместно составить план работы над устранением недостатков.

Аудиторы кода привносят специализированные знания о безопасности или другой опыт, которого нет у разработчиков. Но поскольку эта работа изматывает, а времени у инспекторов мало и с кодом они плохо знакомы, многие важные моменты могут остаться незамеченными. Успех аудита зависит от опыта инспекторов, понимания ими языка и технологической платформы, способности быстро уяснить, что и как делает система, и их психологической выносливости.

Мы еще вернемся к аудиту кода в главе 12.

Автоматизированная инспекция кода

Инструменты сканирования кода можно применить для автоматической инспекции кода на предмет плохих приемов кодирования, а также типичных ошибок и уязвимостей. Как минимум автоматизированная инспекция с помощью инструментов сканирования может служить подспорьем для ручной инспекции, поскольку способна выловить тонкие ошибки по небрежности, которые трудно увидеть, просматривая код глазами.

Ниже в этой главе мы рассмотрим сильные и слабые стороны автоматизированной инспекции кода.

Какой подход к инспекции оптимален для вашей команды?

Чтобы оказывать полезный и продолжительный эффект в быстро изменяющейся среде, процедура инспекции кода должна соответствовать гибким методикам: быть упрощенной, практичной, недорогой и быстрой.

Формальные инспекции обладают строгостью, но дороги и слишком медленны, поэтому большинство гибких команд считает их непрактичными.

Как мы видели, если доверить разработчикам тщательную проверку собственной работы, то можно обнаружить многие дефекты на ранней стадии. Но самоинспекция не отвечает нормативным требованиям и

требованию контроля, она не обеспечивает достаточной прозрачности изменений и не способствует обмену информацией и идеями внутри команды, так что этот подход не годится, хотя он все же лучше, чем полный отказ от инспекций. Если, конечно, команда не состоит из одного человека.

Аудит кода вынесен за рамки цикла разработки: это точечная оценка риска и соответствия нормативным требованиям. На него нельзя рассчитывать как на часть повседневной деятельности.

Парное программирование подходит не всем. Если оно нравится людям, то нравится на все сто. А уж если не нравится, то не нравится ни единая черточка. Хотя парное программирование – замечательный способ обмениваться идеями и вместе решать трудные задачи, брать шефство над новыми членами команды и подвергать код рефакторингу на лету, вовсе не обязательно, что оно окажется эффективным в деле поиска и предотвращения проблем, касающихся безопасности.

Остаются дружественные проверки. Если поставить дело правильно, то обращение к членам команды с просьбой проинспектировать код коллег – пожалуй, самый эффективный подход в гибкой среде. Такие инспекции можно проводить часто, понемногу, без существенных задержек и затрат. В этой главе мы сосредоточимся на том, как сделать дружественные проверки частью цикла разработки и как включить в них проверку на безопасность.

Когда следует инспектировать код?

В процессе разработки есть несколько точек, где можно и должно осуществлять инспекцию кода: до записи кода в главную ветвь репозитория, перед тем как выпускать релиз с изменениями и после обнаружения проблем.

До фиксации изменений

Самый естественный и самый полезный момент для инспекции кода – перед записью изменений в главную ветвь. Именно так сегодня работают многие гибкие команды и разработчики продуктов с открытым исходным кодом, применяющие инструменты коллективной инспекции кода типа Gerrit.

Современные системы управления исходным кодом типа Git легко позволяют это делать. В Git программист создает запрос на включение кода, когда хочет записать изменения в репозиторий. Такой запрос служит для остальных членов команды уведомлением об изменении и дает

им возможность проинспектировать и обсудить изменение, перед тем как объединять его с основным кодом. Администратор репозитория может задать правила, требующие одобрения изменений, и установить другие инструкции по подаче кода.

Во многих средах единственный способ гарантировать проведение инспекций – проводить их до записи в репозиторий: очень трудно убедить разработчика внести в код изменения, после того как он записал код и перешел к новой работе.

Это место также хорошо подходит для выполнения автоматизированных проверок. Например, компания ThoughtWorks разработала для команд, работающих с Git, применяемый перед фиксацией инструмент Talisman (*https://github.com/thoughtworks/talisman*), который ищет в коде подозрительные вещи, например секреты, и предотвращает запись такого кода в репозиторий. Вы можете расширить этот инструмент (или написать свой собственный), реализовав другие проверки, важные в вашей среде.

Контрольно-пропускные проверки перед релизом

Группы безопасности и соответствия нормативным требованиям могут настаивать на проведении инспекций кода хотя бы для некоторых историй или высокорисковых исправлений, иначе работа не может считаться выполненной. Это означает, что команда не может двигаться дальше и развертывать изменения, пока инспекция не завершится и все обнаруженные дефекты не будут устранены.

Ваша задача – работать совместно с командой, направляя ее по безопасному пути. Вместо того чтобы откладывать проверки на конец, перед выпуском релиза, постарайтесь встроить эти шаги в технологический процесс, чтобы группа безопасности немедленно получала уведомления о необходимости инспекций и проводила их как можно быстрее. Останавливайте работу команды, только когда это необходимо, чтобы держать риски под контролем.

Посмертное расследование

Еще один важный момент для проведения инспекций – посмертное расследование, имеющее место после взлома или остановки работы либо если в процессе внешнего аудита или тестирования на проникновение обнаружились неприятные сюрпризы. О том, как правильно проводить посмертные расследования, мы расскажем в главе 13.

Посмертная инспекция кода обычно проводится старшими членами команды, в зависимости от серьезности проблемы к ней могут привле-

каться внешние эксперты. Цель инспекции – убедиться, что вы понимаете, что произошло, и знаете, как исправить дефект, а также помочь в анализе глубинных причин – разобраться, как могла возникнуть проблема, и понять, как предотвратить такие проблемы в будущем.

Понадобится также оформить какие-то бумаги, чтобы доказать – высшему руководству, аудитору, регуляторам, – что инспекция кода была проведена надлежащим образом, и задокументировать предпринятые по ее итогам действия. Это может оказаться серьезной и дорогой работой. Позаботьтесь о том, чтобы люди извлекли максимум пользы из представившейся «возможности».

Как проводить инспекцию кода

Команда, включая уполномоченных по безопасности и соответствию нормативным требованиям, владельца продукта и руководство, должна согласовать порядок проведения инспекций кода.

Сюда включаются правила поведения, описывающие, как следует конструктивно высказывать и принимать критику, чтобы не обидеть людей и не вызвать разлад в команде.

Разработчикам нелегко высказывать критические замечания. Но принимать критику еще труднее. Даже если вы не говорите коллеге прямо, что он дурак или что сделал глупость, все равно он может воспринять ваши слова именно так. Воздерживайтесь от ругани: критикуйте, но вежливо. Пусть лучше инспекторы задают вопросы, а не высказываются в утвердительном ключе. Помните – вы инспектируете код, а не человека. Вместо «тебе не надо было делать X» говорите «программа должна была бы сделать Y, а не X».

Еще одно важное правило поведения состоит в том, что инспекторы должны отзываться на запрос об участии в инспекции в разумное время. Это важно для планирования и позволит команде не терять темп.

Применяйте наставление по кодированию

Важность наставления по кодированию в гибких командах объясняется тем, что они поддерживают принцип *коллективного владения кодом*, согласно которому любой член команды должен быть готов принять любое поручение по написанию кода или исправить любую ошибку (в рамках своей компетенции), и каждый должен смочь выполнить рефакторинг чужого кода. Для эффективного решения этой задачи код должен быть написан единообразно, следуя общему стилю и соглашениям.

В наставлении по кодированию должны быть прописаны как тривиальные соглашения: об именовании элементов, величине отступов и форматировании, – так и более важные вопросы: порядок использования каркасов и библиотек, принципы аудита и протоколирования, запрещенные практики и функции.

В сети есть несколько свободно доступных наставлений по кодированию:

- наставление по кодированию Google (*https://github.com/google/styleguide*) для различных языков;
- стандарты кодирования CERT (*https://wiki.sei.cmu.edu/confluence/display/seccode/SEI+CERT+Coding+Standards*) для C, C++, Java, Android, Java и Perl;
- наставление Microsoft по безопасному кодированию на платформе .NET (*https://docs.microsoft.com/en-us/previous-versions/visualstudio/visual-studio-2008/d55zzx87(v=vs.90)*);
- наставление Oracle по кодированию на платформе Java SE (*http://www.oracle.com/technetwork/java/seccodeguide-139067.html*);
- практика безопасного кодирования OWASP (*https://www.owasp.org/index.php/OWASP_Secure_Coding_Practices_-_Quick_Reference_Guide*);
- наставление Mozilla по безопасному кодированию веб-приложений (*https://wiki.mozilla.org/WebAppSec/Secure_Coding_Guidelines*).

Даже если ваша команда и практика кодирования совершенно не соответствуют этим наставлениям, все равно имеет смысл взять какое-нибудь из них в качестве идеала и составить план постепенного приближения к идеалу. Это будет проще, чем пытаться создать свое наставление с нуля.

Следуя одному из этих наставлений, вы сможете писать более чистый и единообразный код, который проще понять и инспектировать и который будет более безопасным по умолчанию.

Контрольные списки для инспекции кода

Контрольные списки – важный инструмент инспекции. Но они должны быть короткими и целенаправленными.

Ни к чему в контрольном списке пункт, напоминающий инспектору проверить, является ли код понятным и делает ли он то, что должен делать. Базовым правилам программирования и правилам, относящимся к конкретному языку, тоже не место в контрольном списке. Все это должны отлавливать автоматизированные средства статического анализа, в т. ч. и встроенные в IDE разработчика.

Контрольные списки, написанные для пилотов самолетов или хирургов и медсестер в палате интенсивной терапии, напоминают о мелких, но важных вещах, о которых легко забыть, когда все внимание обращено на главную задачу.

Дополнительные сведения о контрольных списках можно найти в замечательной книге доктора Atul Gawande «The Checklist Manifesto» (издательство Metropolitan Books) (*http://atulgawande.com/book/the-checklist-manifesto/*).

Контрольные списки должны напоминать о том, что не очевидно. Напоминать инспекторам надо не о ясности и корректности кода, а о важных вещах, которые легко упустить из виду, а также о вещах, которые не умеют находить инструменты. Как мы увидим ниже в этой главе, сюда относится корректная обработка ошибок, наличие секретных данных в коде и конфигурационных файлах, прослеживание использования конфиденциальных или закрытых данных, а также выявление отладочного кода и кода, оставленного случайно. Составьте собственные контрольные списки, ориентируясь на проблемы, встречавшиеся в производственной системе или выявленные в ходе тестирования на проникновение, чтобы не повторять прошлых ошибок.

Не делайте этих ошибок

Существует ряд типичных ошибок и антипаттернов, которых следует избегать при проведении инспекций кода.

1. Мнение о том, что код старших по должности инспектировать не нужно. Все члены команды должны придерживаться одинаковых правил, независимо от должности. Часто старшие члены берут на себя написание более трудного кода и решение сложных задач. Такой код инспектировать нужно в первую очередь.
2. Отказ от инспекции унаследованного кода. Исследования эффекта медового месяца в разработке ПО доказывают, что существует период – медовый месяц – после развертывания новых функций и изменений, в течение которого противники еще не понимают, как их можно использовать и атаковать. Большинство успешных атак направлено против кода, который существует уже достаточно долго, чтобы «плохие парни» нашли уязвимости и способы их эксплуатации. Особенно это относится к популярным сторонним библиотекам, в т. ч. с открытым исходным кодом, и к платформенному коду. После того как противник нашел уязвимость в

этом коде и научился опознавать ее по характерным признакам, риск компрометации резко возрастает[2].

Таким образом, изменения следует инспектировать по мере внесения, поскольку находить и исправлять ошибки сразу проще и дешевле, но не менее важно просматривать старый код, особенно если он написан до того, как команда прошла обучение безопасной разработке.

3. Полагаться только на автоматизированные инспекции кода. Инструменты автоматизированного сканирования могут помочь в нахождении проблем и выявлении кода, который надо бы почистить, но, как мы покажем ниже, они не могут служить полноценной заменой ручной инспекции. Это случай, когда важно И, а не ИЛИ.

Избегая этих принципиальных ошибок, вы повысите безопасность и качество своего кода.

Инспектируйте код небольшими порциями

Еще одна типичная ошибка – понуждать членов команды инспектировать большие куски кода. Опыт, подтвержденный специальными исследованиями, показывает, что между эффективностью инспекций и объемом инспектируемого кода имеется отрицательная корреляция. Чем больше файлов и строк кода должен просмотреть инспектор, тем сильнее он устает и тем меньше проблем в состоянии найти.

Инспектор, вынужденный просматривать 1000 строк измененного кода, может отметить, что код трудно понять и что какие-то вещи можно было бы сделать проще. Но он не сможет найти все ошибки. А если инспектору нужно просмотреть всего 50 или 100 строк кода, то найти ошибки, в т. ч. тонкие, ему будет гораздо легче. И справится он гораздо быстрее.

На самом деле исследования показывают, что эффективность инспекций падает после примерно 200 строк кода и что количество найденных дефектов стремительно снижается, если инспекция продолжается дольше часа (см. главу «Modern Code Review» в книге «Making Software» издательства O'Reilly).

Эта проблема неизбежна в случае аудита кода, когда аудиторы вынуждены просматривать тысячи строк каждый день, часто в течение нескольких дней подряд. Но ее не должно быть в гибких командах, осо-

[2] См. отчет Rebecca Gelles «The Unpredictable Attacker: Does the 'Honeymoon Effect' Hold for Exploits?», 6.02.2012 (*http://dreuarchive.cra.org/2011/Gelles/rdreureport.pdf*).

бенно практикующих непрерывную интеграцию, поскольку изменения вносятся итеративно, небольшими порциями, так что их можно инспектировать при каждой записи в репозиторий.

Какой код следует инспектировать?

Команда должна решить, какой код надлежит инспектировать и кто должен принимать участие в инспекциях.

В идеальном мире все изменения следовало бы инспектировать на предмет удобства сопровождения, правильности и безопасности. Это требует высокого уровня технической дисциплины и одобрения руководства.

Если вы пока к этому не готовы или не можете убедить руководство или заказчика выделить время для инспекции всех изменений, то все равно можно достичь многого, приняв прагматичный подход, основанный на оценке рисков. При использовании непрерывной интеграции изменения, по большей части, небольшие и инкрементные, и связанный с ними риск безопасности ограничен, особенно если код быстро выбрасывается в корзину, когда команда экспериментирует с альтернативными проектами.

Для многих таких изменений можно положиться на автоматизированные средства сканирования кода, встроенные в IDE разработчика и в конвейер непрерывной интеграции, – они обнаружат типичные ошибки кодирования и нерекомендуемые способы написания кода. Как мы увидим ниже, возможности этих инструментов ограничены, но все же их может хватить, чтобы ограничить риски и помочь в соблюдении нормативных требований (например, стандарт PCI DSS требует инспектировать все изменения, но допускает автоматизированные инспекции).

Но иногда необходимо просматривать код более пристально и тщательно, поскольку риск из-за пропуска ошибки слишком велик. Вот некоторые из таких случаев:

- функции, связанные с безопасностью, в т. ч. аутентификация и контроль доступа, криптографические функции и код, в котором они используются;
- код, относящийся к обработке полей, содержащих денежные суммы, конфиденциальную и частную информацию;
- API работы с другими системами, имеющими дело с обработкой денежных сумм, конфиденциальной и частной информации;
- пользовательский интерфейс мобильных и веб-приложений (например, массивные изменения или новые рабочие процессы);

- API, доступные из открытых сетей, импорт файлов из внешних источников (т. е. код, который часто становится мишенью атак);
- код, являющийся частью каркасов, и другой связующий код;
- неизведанный код, когда начинается освоение нового каркаса или подхода к проектированию и команда еще не поняла, как правильно работать в этих условиях;
- код, написанный новыми членами команды (по крайней мере, если они не работали в паре с более опытным коллегой), – необходимо проверить соблюдение принятых в команде паттернов и наставлений.

Изменения каркасов, средств обеспечения безопасности и открытых API следует также инспектировать в процессе проектирования, как составную часть моделирования угроз (см. главу 8).

Высокорисковый код и правило 80:20

Правило 80:20 напоминает, что большая часть ошибок обычно находится в небольшой части кода:

80% ошибок сосредоточено в 20% программы.

В ряде исследований обнаружено, что в половине кода может вообще не быть дефектов, а большинство дефектов сосредоточено в 10–20% кода, как правило, в той части, которая чаще всего изменяется (см. отчет Forrest Shull и др. «What We Have Learned About Fighting Defects»).

Учет ошибок поможет понять, на что обращать особое внимание в процессе инспекций и тестирования. Код с плохой историей ошибок следует также сделать первым кандидатом на рефакторинг или переписывание.

Хотя команде обычно очевидно, какой код несет в себе риски, перечислим несколько шагов, которые помогут выявить и поставить на особый учет код, нуждающийся в тщательной инспекции.

- Пометить пользовательские истории, имеющие отношение к безопасности или рабочим процессам, связанным с обработкой денежных сумм или секретных данных.
- «Прогрепить» исходный код в поисках обращений к опасным функциям, в т. ч. криптографическим.
- Просканировать комментарии к инспекциям кода (если вы пользуетесь каким-то инструментом коллективной инспекции типа Gerrit).

- Учитывать все операции записи в репозиторий, чтобы выявить часто изменяемый код: код, который часто переделывается, обычно содержит больше дефектов.
- Анализировать отчеты о дефектах и результаты статического анализа для выявления проблемных участков кода: искать следует код с плохой историей ошибок, код повышенной сложности и код с низким уровнем покрытия автоматизированными тестами.
- Выявлять код, который недавно был подвергнут крупномасштабному рефакторингу – «депульпированию зуба». Хотя каждодневный рефакторинг по ходу разработки способен многое сделать для упрощения кода, повышения его понятности и безопасности изменения, в процессе серьезного рефакторинга или перепроектирования возможно случайное изменение модели доверия приложения и возрождение старых, уже исправленных ошибок.

В Netflix практикуется интересный способ выявления высокорискового кода – программа Penguin Shortbread, которая строит карту обращений к микросервисам. Сервисы, которые вызываются из большого числа других сервисов, или те, которые обращаются ко многим сервисам, автоматически помечаются как высокорисковые зависимости, нуждающиеся в инспекции.

В компании Etsy, как только в процессе инспекции или сканирования выявлен высокорисковый код, разработчики вычисляют его хеш-значение и создают автономный тест, который автоматически уведомляет группу безопасности при каждом изменении этого хеш-значения в результате записи в репозиторий.

Наконец, можно поручить команде внимательно следить за рисками в коде – разработчик может запросить инспекцию кода, если полагает, что нуждается в помощи.

Кто должен инспектировать код?

Договорившись о том, какой код следует инспектировать и когда проводить инспекцию, следует решить, кто должен принимать участие в инспекциях и сколько инспекторов привлекать.

Может ли любой член команды выступать в роли инспектора, или это нужно поручать только тем, кто уже работал над инспектируемым кодом раньше, или это должен быть профильный специалист? Когда к инспекции следует привлекать более одного человека?

Сколько должно быть инспекторов?

В основе инспекций (и парного программирования) лежит разумное соображение: два человека найдут больше проблем, чем один. Но если две головы лучше одной, то почему бы не увеличить количество инспекторов, чтобы голов стало еще больше?

В Google и Microsoft, где имеется большой опыт успешных инспекций кода, экспериментально выяснили, что два инспектора – оптимальное число. Большинство команд требуют двух инспекций, хотя бывает, что автор запрашивает больше, особенно если инспекторы не согласны друг с другом.

Некоторые команды в Microsoft специально запрашивают инспекции двух разных видов, чтобы получить максимальную отдачу от каждого инспектора.

1. Инспекция до записи кода в репозиторий, ориентированная в основном на оценку удобочитаемости, ясности и удобства сопровождения.
2. Вторая инспекция (до или после записи в репозиторий) нужна для поиска рисков и ошибок.

Исследования показали, что второй инспектор находит вдвое меньше новых проблем, чем первый. Увеличение количества инспекторов – пустая трата времени и денег. В одном исследовании показано, что команды из трех, четырех и пяти человек находят одинаковое количество проблем, а в другом – что два инспектора справляются с работой лучше, чем четыре[3].

Отчасти это объясняется пересечением и избыточностью. Чем больше инспекторов, тем больше народу ищут и находят одни и те же проблемы (и тем больше находок оказывается ложноположительными, и автору приходится тратить время на их отсеивание). Встречается также проблема «социальной лени». Увеличение количества инспекторов вселяет ощущение самоуспокоенности и ложной безопасности: поскольку каждый инспектор знает, что код просматривает кто-то еще, личная ответственность за поиск проблем снижается.

Но есть задача и поважнее привлечения правильного числа инспекторов – выбрать подходящих для инспекции кода людей.

---

[3] Chris Sauer, D. Ross Jeffery, Lesley Land, Philip Yetton «The Effectiveness of Software Development Technical Reviews: A Behaviorally Motivated Program of Research», IEEE Transactions on Software Engineering, Vol 26, Issue 1, January 2000: 1–14.

Каким опытом должны обладать инспекторы?

Новый член команды может многое почерпнуть из анализа своего кода, проведенного опытным коллегой, в частности лучше понять, как работает система и команда. Инспектируя код, написанный другими членами команды, новый разработчик расширяет свои представления о кодовой базе и узнает что-то новое о системе. Но такой способ обучения неэффективен. И теряется главная цель инспекции кода – как можно раньше найти и исправить максимальное количество ошибок.

Исследования подтверждают то, что и так очевидно: эффективность инспекции кода всецело зависит от опыта инспектора, его знакомства с предметной областью и платформой и способности понять код. Проведенное в Microsoft исследование показало, что инспекторы, не входящие в состав команды или недавно принятые в команду и незнакомые с кодом или предметной областью, были способны только на поверхностный анализ: они находили лишь проблемы с форматированием и простые логические ошибки. Как и в других областях разработки ПО, диапазон индивидуальных показателей очень широк: лучшие инспекторы оказываются в 10 раз эффективнее в поиске проблем и высказывании ценных замечаний.

Это означает, что ваши лучшие, самые опытные разработчики будут – и должны – тратить уйму времени на инспекцию кода. Вам нужны инспекторы, которые хорошо умеют читать и отлаживать код, знают язык, каркасы и предметную область. Они гораздо лучше справятся с поиском проблем и смогут дать гораздо более полезные отзывы, в т. ч. предложения о том, как решить задачу проще и эффективнее или как правильнее пользоваться языком и каркасами. И все это они смогут сделать гораздо быстрее.

В нормативных требованиях может быть оговорен опыт инспекторов. Например, стандарт PCI DSS 6.3.2 требует, чтобы инспекторы «хорошо разбирались в технике инспекции кода и практических методах безопасного кодирования».

Если вы хотите, чтобы новый разработчик больше узнал о коде, соглашениях о кодировании и архитектуре, то будет гораздо полезнее посадить его в пару с опытным членом команды за программирование или отладку, а не просить проинспектировать чужой код. Если вы все-таки вынуждены привлекать неопытных разработчиков к инспекции кода, опустите планку ожиданий. Вы должны понимать, что в этом случае для

поиска реальных ошибок в коде придется больше полагаться на автоматизированные средства статического анализа и тестирование.

Автоматизированная инспекция кода

Автоматизированные средства статического анализа должны быть частью программы инспекции кода в силу следующих причин:

- автоматизированное сканирование кода – единственный практически применимый способ обеспечить покрытие большой унаследованной кодовой базы и добиться сколько-нибудь приемлемого понимания безопасности и качества этого кода;
- статическое сканирование можно производить непрерывно для всего кода. Программа никогда не устанет искать одни и те же проблемы и будет делать это гораздо более стабильно, чем инспектор-человек;
- в отличие от людей, хорошие инструменты статического анализа не смутят неудачно выбранное имя элемента, не тот отступ и другие косметические детали;
- хотя, как мы увидим, автоматизированные сканеры кода могут пропускать многие важные уязвимости, они хорошо находят некоторые ошибки, важные с точки зрения удобочитаемости и безопасности. Это особенно ценно, если в компании нет специалистов по безопасности, способных эффективно инспектировать код;
- средства автоматизированной инспекции кода считаются приемлемой альтернативой ручной инспекции в некоторых нормативных требованиях, в частности в стандарте PCI DSS. Для многих команд это практичный и рентабельный способ удовлетворить нормативные требования.

Некоторые инструменты статического анализа сканируют байт-код или двоичный код. Их проще всего настроить и запустить, поскольку достаточно указать сканеру местоположение развернутого пакета или загрузить двоичные файлы в службу сканирования типа Veracode – все остальное будет сделано без вашего участия.

Другие инструменты сканируют сам исходный код, а значит, не нужно ждать окончания компиляции и можно сканировать отдельные части кода или наборы изменений. Но чтобы просканировать систему целиком, нужно понимать, как устроены библиотеки приложения и все зависимости.

В отчете, сформированном автоматизированными инструментами, обычно указывается номер строки кода, в которой найдена проблема (а часто и путь, по которому программа достигает этой строки). Некоторые инструменты применяют просто сопоставление с образцом или проверяют соблюдение стандартов оформления кода, тогда как другие строят абстрактную модель приложения, создают карту потоков данных и анализируют пути выполнения кода в поисках таких уязвимостей, как внедрение SQL или межсайтовый скриптинг. Поскольку для большой кодовой базы это может занять много времени, некоторые движки анализа сохраняют абстрактную модель, сканируют только изменившийся код и инкрементно обновляют модель.

Инструменты анализа кода могут обнаружить типичные, но важные ошибки кодирования и помочь во внедрении хорошего стиля кодирования, чтобы инспекторы могли сосредоточиться на других важных проблемах.

Обычно они хорошо справляются со следующими дефектами:

- небрежное кодирование, в т. ч. плохо структурированный, неиспользуемый или недостижимый код, фрагменты, полученные копированием и вставкой, нарушения общепринятых правил хорошего кодирования;
- тонкие ошибки, которые должен был бы найти компилятор, но не нашел. Такие ошибки трудно обнаружить в процессе тестирования или ручной инспекции, например: ошибки в условной логике, переполнение буфера и затирание памяти в коде на C/C++, нулевые указатели в Java;
- уязвимости из-за пропущенной проверки данных и возможного внедрения, когда программист не проверяет длину входной строки или передает неочищенные данные интерпретатору, например базе данных SQL или браузеру;
- типичные ошибки в функциях, относящихся к безопасности, например криптографических (слабые алгоритмы шифрования и хеширования, слабые ключи, слабые генераторы случайных чисел);
- часто встречающиеся ошибки конфигурирования, например небезопасные HTTP-заголовки или куки без флага Secure.

Эти инструменты должны быть частью программы инспекции кода, но не единственной. Чтобы понять, почему, рассмотрим различные типы инструментов и назначение каждого из них.

Разные инструменты находят разные проблемы

Для большинства сред существует целый спектр инструментов статического анализа, предназначенных для поиска различных проблем.

Предупреждения компилятора

Статический анализ кода должен начинаться с проверки предупреждений компилятора. Авторы компиляторов включают такие предупреждения не без причины. Не нужно покупать специальную программу, которая скажет ровно то, что и так уже сообщает компилятор. Задайте максимальный уровень предупреждений, внимательно изучите их и наведите порядок.

Стиль кодирования и запахи кода

Инструменты, проверяющие код на единообразие, удобство сопровождения и ясность (PMD и Checkstyle для Java, Ruby-lint для Ruby), помогают писать код, который будет проще понять, проще и безопаснее изменять и проще инспектировать. Они помогут сделать весь код единообразным. Они укажут на плохо написанный код: несоблюдение соглашений о безопасном кодировании, типичные ошибки из-за копирования и вставки и потенциально серьезные ошибки объединения.

Но если вы с самого начала не придерживались общепризнанного стиля оформления кода, то придется подправить стилистические правила, так чтобы они соответствовали принятым в команде соглашениям.

Типичные дефекты

Инструменты, которые ищут типичные ошибки кодирования и паттерны ошибок (например, FindBugs и RuboCop), вылавливают тонкие логические ошибки, которые могли бы привести к отказам во время выполнения или уязвимостям.

Уязвимости (SAST)

Инструменты, способные выявлять уязвимости посредством анализа потока управления и потока данных, применения эвристик, анализа паттернов и других методов (Find Security Bugs, Brakeman, Fortify), умеют находить типичные проблемы, касающиеся безопасности, в т. ч. ошибки при использовании криптографических функций, ошибки в конфигурации и потенциальные уязвимости перед атаками внедрением.

Иногда эти инструменты собирательно называют SAST – «Static Analysis Security Testing» (статический анализ для проверки на уязвимости).

Эта категория введена компанией Gartner, чтобы отличить от динамического анализа работающего приложения – DAST (к этому классу относятся инструменты типа ZAP и Burp). Сканеры класса DAST мы будем рассматривать в следующей главе.

Специализированные команды grep и детекторы

Простые самодельные программы, которые, подобно grep, ищут в коде зашитые учетные данные, вызовы небезопасных или запрещенных функций (скажем, gets, strcpy, memcpy в C/C++ или eval в PHP и Javascript), обращения к криптографическим библиотекам и прочие вещи, на которые команда разработчиков или группа безопасности должны обращать особое внимание.

Вы можете написать собственные проверки, пользуясь возможностью расширения других инструментов, например реализовать собственный детектор дефектов в FindBugs или свое правило кодирования для PMD. Правда, делают это лишь немногие команды.

Обнаружение ошибок по ходу кодирования

Некоторые ошибки автоматически обнаруживает IDE с помощью модулей, подключаемых к Eclipse, Visual Studio или IntelliJ, либо встроенных инструментов проверки кода и автодополнения, которые входят в состав большинства современных комплектов средств разработки.

Эти инструменты не выполняют глубокого анализа потока данных или потока управления, но могут подсветить типичные ошибки и сомнительный код прямо во время ввода, т. е. играют роль своеобразного средства проверки правописания, только для безопасности.

Отметим несколько бесплатных подключаемых модулей для разных IDE:

- расширения Eclipse для FindBugs (*https://marketplace.eclipse.org/content/ findbugs-eclipse-plugin*) и Find Security Bugs (*http://find-sec-bugs.github.io/*) для Java;
- Puma Scan (*https://www.pumascan.com/*), расширение Visual Studio для C#.

Другие модули, подключаемые к IDE, например HPE Fortify, получают исходные данные в виде результатов проведенного ранее пакетного сканирования и представляют их в IDE в момент, когда программист работает над проблемными участками кода. Это дает возможность лучше видеть существующие проблемы, но не позволяет сразу же отлавливать новые ошибки.

Уязвимые зависимости

Инструменты типа OWASP Dependency-Check, Bundler-Audit для проектов на Ruby или Retire.JS для JavaScript ведут реестр зависимостей сборки и проверяют, что они не содержат известных уязвимостей.

Можно указать, что сборка должна завершаться ошибкой, если при проверке найдена серьезная уязвимость или иная проблема. Мы уже рассматривали эти инструменты в главе 6.

Анализ сложности кода и показатели технического долга

Другие инструменты могут формировать отчеты о таких показателях, как сложность кода или иные результаты измерения технического долга, выявлять проблемные участки кода («горячие точки» или кластеры) и тенденции. Например, сопоставление сложности кода с покрытием автоматизированными тестами – способ выявления потенциальных рисков в кодовой базе.

SonarQube (*https://www.sonarqube.org/*), популярная платформа для анализа качества и безопасности кода, включает калькулятор стоимости технического долга, а также другие средства измерения. Технический долг вычисляется путем назначения весов различным результатам статического анализа (отход от рекомендованных практик кодирования, наличие «мертвого» кода, сложность кода, наличие дефектов и уязвимостей) и пробелам в покрытии тестами. Стоимость устранения найденных проблем выражается в долларах. Даже если вы не согласны с моделью расчета, принятой в SonarQube, информационная панель полезна для отслеживания изменения технического долга со временем.

Какие инструменты для чего подходят

Убедитесь, что все – вы сами и члены команды – понимают, что дает инструмент статического анализа и в какой мере на результаты можно положиться.

Одни инструменты лучше находят определенные проблемы, чем другие, и это во многом зависит от того, как вы используете язык, каркасы и паттерны проектирования.

На сегодня существует несколько хороших инструментов статического анализа – коммерческих и с открытым исходным кодом – для основных языков (Java, C/C++ и C#), ориентированных на такие широко распространенные каркасы, как Struts, Spring и .NET. Есть также инструменты для других популярных сред разработки, например Ruby on Rails.

Но трудно найти хорошую инструментальную поддержку для таких новых, вызывающих острый интерес языков, как Go, Swift, F# или Jolie, а еще труднее отыскать инструменты, которые находят реальные проблемы в динамически типизированных скриптовых языках типа Javascript, PHP и Python, хотя именно тут проверка нужна больше всего. Большинство анализаторов кода для этих языков (по крайней мере, с открытым исходным кодом) ограничено соблюдением стилистических правил (linting) и простейшими проверками следования рекомендованным практикам. Это помогает писать более качественный код, но не гарантирует ни безопасность, ни хотя бы что программа не «грохнется» во время выполнения.

IAST или RASP: альтернативы статическому анализу кода

IAST (Interactive или Instrumented Application Security Testing – интерактивное или инструментальное тестирование безопасности приложения) и RASP (Runtime Application Self-Protection – самозащита приложения во время выполнения) – новые технологии, предлагающие альтернативу статическому анализу кода. Они оснащают исполняющую среду (например, Java JVM) измерительными средствами и строят модель приложения в процессе выполнения, для чего анализируют стек вызовов и переменные на предмет выявления уязвимостей в работающей программе.

Однако, как и в случае инструментов статического анализа, поддержка сильно зависит от языка и платформы, как, впрочем, и эффективность правил. Качество покрытия кода этими инструментами также зависит от того, насколько полно выполняется код в тестах.

Инструмент статического анализа может сообщить, когда программа вызывает небезопасные библиотечные функции, но не может сказать, что программист забыл обратиться к какой-то функции, хотя должен был. Инструмент может сообщить, что сделана простая ошибка при вызове криптографической функции, но ничего не скажет о том, что вы забыли зашифровать или представить в виде маркеров секретные данные или свериться со списком контроля доступа либо случайно раскрыли секреты в процессе обработки исключения. Это может сделать только опытный инспектор.

Проводятся важные исследования, позволяющие понять эффективность и пределы автоматизированных средств статического анализа.

Авторы одного исследования применили два коммерческих инструмента статического анализа к большому приложению, содержащему 15 известных уязвимостей (ранее обнаруженных в ходе структурированного ручного аудита экспертами по безопасности). Оба инструмента вместе нашли менее половины известных ошибок – и только простейшие проблемы, которые не требовали глубокого понимания кода или проекта[4].

Инструменты сообщили также о тысячах прочих проблем, которые необходимо было проанализировать и квалифицировать либо отбросить как ложноположительные. В том числе были найдены проблемы с корректностью на этапе выполнения – нулевые указатели, утечки ресурсов – которые, вероятно, следовало бы исправить, а также проблемы качества кода («мертвый» код, неиспользуемые переменные), но никаких других уязвимостей.

NIST регулярно проводит серию эталонных тестов для оценки эффективности инструментов статического анализа под названием SAMATE (*https://samate.nist.gov/Main_Page.html*). В последний раз (2014 год) NIST протестировала 14 таких инструментов для программ на C/C++ и Java, содержащих известные уязвимости. И вот что было обнаружено.

1. Свыше половины уязвимостей не обнаруживались ни одним инструментом.
2. По мере увеличения сложности кода способность инструментов находить проблемы резко падала. Многие инструменты просто отказывались работать. В частности, это было продемонстрировано на примере ошибки Heartbleed в OpenSSL, которую не смог найти ни один из доступных инструментов, отчасти из-за чрезмерной сложности кода.
3. NIST также обнаружила значительное и обескураживающее отсутствие пересечения между найденными результатами: менее 1% уязвимостей было найдено всеми инструментами[5].

Позже в составе проекта OWASP Benchmark Project (*https://www.owasp.org/index.php/Benchmark*) был создан комплект тестов с известными уязвимостями, предназначенный для оценки эффективности различных инструментов статического анализа и сканеров приложений. В нем

[4] James A. Kupsch, Barton P. Miller «Manual vs. Automated Vulnerability Assessment: A Case Study» (2009) (*http://pages.cs.wisc.edu/~kupsch/va/ManVsAutoVulnAssessment.pdf*).

[5] Aurelien Delaitre, Delaitre, Aurelien, Bertrand Stivalet, Elizabeth Fong, Vadim Okun «Evaluating Bug Finders – Test and Measurement of Static Code Analyzers», 2015 IEEE/ACM 1st International Workshop on Complex Faults and Failures in Large Software Systems (COUFLESS) (2015): 14–20 (*https://samate.nist.gov/docs/Evaluating_Bug_Finders_COUFLESS_2015.pdf*).

инструменты оцениваются путем вычитания ложноположительных результатов из истинно положительных. Средняя оценка коммерческих инструментов типа SAST составила всего 26%.

Хотя инструменты все время совершенствуются – становятся точнее, быстрее, проще для понимания и лучше поддерживают языки и каркасы, – они не могут заменить инспекции силами грамотных специалистов. Но могут выступать в роли неплохого подспорья ручным инспекциям путем вылавливания типичных ошибок. Кроме того, навязывая единообразие и соблюдение рекомендованных правил кодирования, они способны упростить ручные инспекции и сделать их более эффективными.

Приучение разработчиков к автоматизированным инспекциям кода

Мы хотим прийти к положению, когда команды разработчиков рассматривают результаты статического анализа как результаты прогона автономных тестов: если при попытке записать изменение в репозиторий инструмент сообщает о какой-то проблеме, то она исправляется немедленно, поскольку разработчики научились доверять инструменту и знают, что могут положиться на него в деле нахождения важных ошибок.

Внедрить статический анализ уязвимостей проще в команды, которые уже имеют положительный опыт использования инструментов статического анализа для оценки качества кода и доверяют им. Для начала постарайтесь убедить команду пользоваться каким-нибудь хорошим инструментом поиска дефектов, а когда команда примет его и сделает частью повседневной работы, переходите к проверке безопасности.

Бросив на стол разработчика отчет, содержащий тысячи предупреждений инструмента статического анализа, вы не завоюете популярности у команды. Попытайтесь применить постепенный подход, не приводящий к трениям.

Потратьте время, чтобы понять, как инструмент работает и как правильно настроить правила и средства проверки. Многие команды оставляют параметры по умолчанию, а это почти никогда не приносит должного результата. В некоторых инструментах по умолчанию выставляются консервативные параметры, т. е. не применяются правила и средства проверки, важные конкретно для вашего приложения. Другие инструменты хотят проверять все, что возможно, и выполняют не относящиеся к делу проверки, забрасывая разработчиков ложноположительными предупреждениями.

Установите инструмент и выполните серию сканирований с различными правилами и параметрами. Измерьте, сколько времени заняло сканирование и сколько было затрачено памяти и ресурсов процессора. Проанализируйте результаты и найдите оптимальное соотношение между максимизацией истинно положительных сигналов и минимизацией ложноположительных. Выберите правила и средства проверки, которые дают максимальную уверенность. Остальные правила отключите, по крайней мере на начальном этапе.

Сохранение правил в репозитории

Не забывайте сохранять выставленные правила и параметры в репозитории, чтобы можно было узнать, какие правила действовали в любой момент времени. Это может понадобиться, чтобы доказать аудитору, что вы не просто играете в безопасность, пытаясь обойтись проверками, не доставляющими хлопот разработчикам, но пропускающими слишком много реальных проблем.

Определите точку отсчета. Выполните полное сканирование, покажите результаты всей команде или, быть может, для начала двум самым опытным разработчикам, объясните, что умеет делать инструмент, и продемонстрируйте, как он работает. Затем пометьте всё найденное, чтобы инструмент по умолчанию не показывал эти результаты при последующем сканировании. Это долг безопасности, который нужно будет оплатить позже.

Убедите команду согласиться с нулевой терпимостью к дефектам в будущем: начиная с этого момента, команда обязуется анализировать все найденные инструментом серьезные проблемы и устранять те, что действительно присутствуют.

Вместо того чтобы просеивать многие страницы результатов, члены команды будут видеть только несколько предупреждений каждый день или при каждой записи в репозиторий – в зависимости от того, как часто запускается сканер. Это не добавит им много работы и в конечном итоге станет еще одной частью повседневного процесса.

После того как команда станет доверять инструменту и научится эффективно обрабатывать результаты, она может запланировать инспекцию и исправление проблем, найденных в момент создания точки отсчета.

Ополоснуть и повторить

Поскольку находки разных инструментов почти не пересекаются, для получения эффективного покрытия придется использовать несколько инструментов статического анализа. А это значит, что все описанные шаги придется повторить несколько раз. Конечно, с каждым разом процесс будет проходить все легче, но все равно будьте готовы к издержкам.

Разумеется, приняв этот подход, вы оставите некоторые дефекты и уязвимости открытыми для атаки, по крайней мере на время. Но теперь вы хотя бы знаете, какие есть проблемы и где они находятся, и можете поработать с владельцем продукта, командой и руководством, чтобы спланировать устранение рисков.

Сканирование в режиме самообслуживания

На многих крупных предприятиях сканирование кода производится группой безопасности в рамках модели, которую д-р Гэри Макгроу (Gary McGraw) и компания Cigital называют «фабрикой сканирования». Группа безопасности планирует и запускает сканирование различных проектов, анализирует и рассортировывает результаты, после чего сообщает о них разработчикам.

При таком подходе образуются ненужные задержки в доведении результатов до команды. На сканирование, анализ результатов и подготовку отчетов может уйти несколько дней. К тому моменту, как разработчики узнают о проблемах, они уже могут перейти к другой работе, а значит, им придется оставить текущие дела, вернуться к прежней задаче, восстановить контекст, найти проблему, исправить ее, протестировать, собрать систему, а затем снова переключиться. Это пустая трата времени, т. е. как раз то, чего гибкие и бережливые методики стараются избегать.

Еще один недостаток этой модели заключается в том, что сканирование становится «чьей-то проблемой», а у разработчиков исчезает ощущение владения.

Более эффективное, масштабируемое и гибкое решение – поручить сканирование кода самим разработчикам, и пусть они выбирают самый естественный для себя способ.

Вместо того чтобы выбирать стандартизированные инструменты, удобные группе безопасности для использования в нескольких проек-

тах, позвольте команде выбрать те инструменты, которые лучше отвечают ее нуждам и хорошо ложатся на технологические процессы.

Первое, что хочется сделать, – включить сканирование в режиме самообслуживания в IDE каждого разработчика и воспользоваться подключаемыми модулями или встроенными в IDE правилами для поиска проблем в процессе кодирования, или при сохранении изменений, или на этапе компиляции. Очень важно, чтобы правила имели высокую способность обнаружения, т. е. находили реальные проблемы, и чтобы им можно было доверять. В противном случае разработчики быстро привыкнут игнорировать все предупреждения и подсветку.

Но имейте в виду, что в этот момент остальные члены команды не знают, какие проблемы были найдены, и нет никакого способа доказать, что разработчик сразу исправляет найденные ошибки. А потому все равно придется включать инкрементный анализ и быстрое сканирование кода при каждой сборке, чтобы отловить оставшиеся проблемы.

У большинства инструментов сканирования имеется API, который позволяет передать результаты прямо в IDE каждого разработчика или автоматически создать отчет о дефектах в программах типа Jira или VersionOne. Можно также воспользоваться каким-нибудь из средств управления уязвимостями приложения, описанных в главе 6.

Обратная связь должна быть точной и быстрой

Если вы хотите, чтобы разработчики серьезно воспринимали результаты сканирования, то должны предоставить обратную связь, высвечивающую реальные проблемы, которые необходимо исправить. Время, потраченное на исследование ложноположительных предупреждений, не имеющих отношения к проекту, – это время, потраченное зря. Если так происходит часто, то и сам разработчик, и его начальник потеряют желание сотрудничать с вами.

Будьте безжалостны, выбирая между полнотой и скоростью и точностью результатов. В процессах с непрерывной интеграцией или непрерывной поставкой короткий и высококачественный цикл обратной связи, вообще говоря, важнее полноты.

Сделайте так, чтобы разработчикам было легко сообщить о встретившихся ложноположительных сигналах или шуме. В компании Twitter есть кнопка «Хрень собачья», которая позволяет разработчикам подавить ложноположительные предупреждения и сообщить об этом.

Для сканирования в режиме самообслуживания с быстрой обратной связью нужны инструменты, которые работают быстро и предоставляют понятный контекст для каждой находки, чтобы разработчику не приходилось звать специалиста по безопасности, который объяснил бы, в чем проблема, насколько она серьезна, где и как обнаружена и как ее исправить.

При наличии ресурсов можно выполнить и глубокое полное сканирование в свободное время, затем проанализировать и классифицировать результаты и поместить их в журнал пожеланий для последующего исправления. Для большой кодовой базы на это может уйти несколько часов или дней, но количество обнаруженных важных проблем должно быть невелико – это те ошибки, которые остались незамеченными в процессе более быстрой и достаточно надежной проверки, встроенной в цикл разработки. Поэтому затраты на анализ результатов и замедление работы команды, вынужденной возвращаться назад, скорее всего, будут минимальны, тогда как риск пропустить важную проблему все же снижается.

Инспекция инфраструктурного кода

Современные системы создаются с применением автоматизированных программируемых средств управления конфигурацией типа Ansible, Chef, Docker и Puppet. К рецептам Chef, манифестам Puppet, сценариям Ansible, Docker-файлам и шаблонам AWS CloudFormation должен применяться такой же жизненный цикл, как к остальному коду: запись в репозиторий, ручная инспекция, автоматизированные проверки во время сборки, проверки правильности оформления и другие виды статического анализа, а также автоматизированное автономное, интеграционное и приемочное тестирование.

Инспекция рецептов и сценариев производится по тем же причинам, что для кода приложения:

- гарантировать, что все изменения конфигурационных параметров системы проверены хотя бы одним человеком (контроль);
- проверить удобство сопровождения и соответствие соглашениям о кодировании;
- проверить правильность;
- убедиться, что к производственной системе случайно не применена тестовая конфигурация;
- применить политики эксплуатации и безопасности.

Инспекторы могут применять те же методы инспекции кода, рабочие процессы и средства инспектирования, что и разработчики, например Gerrit или Review Board. Они должны обращать внимание на стилистические и структурные проблемы, правильное разделение кода и данных, повторное использование и удобочитаемость. Кроме того, инспектор должен следить за тем, чтобы для каждого модуля были хорошие тесты. Но самое главное – обращать внимание на ошибки и упущения конфигурации, которые могут сделать систему уязвимой для атаки.

Не следует излишне полагаться на средства статического анализа при инспекции подобного кода. Такие инструменты, как Foodcritic для Chef или Puppet-lint для Puppet, выполняют простые проверки и ищут типичные ошибки, что существенно для этих скриптовых языков, когда желательно найти проблемы до запуска. Но по умолчанию они не находят серьезных ошибок, связанных с безопасностью. Для этого вам придется самостоятельно написать правила[6].

Мы вернемся к вопросу о безопасности, когда будем рассматривать работу с этими инструментами в главе 13.

Недостатки и ограничения инспекции кода

Хороший инспектор при наличии опыта и времени может найти много дефектов в коде. Он обнаружит логические ошибки и упущения, часто пропускаемые автоматизированными средствами сканирования, например:

- несогласованность (автор изменил *a*, *b* и *d*, но забыл изменить *c*);
- типичные опечатки, например использование < вместо <=, а иногда даже вместо > в сравнениях;
- ошибки смещения на единицу;
- использование не тех переменных в вычислениях или сравнениях (buyer вместо seller).

При чтении кода инспектор может также найти ошибки или слабые места в проекте – если он хорошо знаком с проектом системы. Но, как мы говорили в главе 8, на уровне проектирования следует проводить отдельные инспекции, в ходе которых рассматриваются предположения о доверии, угрозы и защита от угроз.

Инспектор легко найдет тестовый или отладочный код, оставленный по невниманию, и, скорее всего, заметит избыточный код и лишние проверки – признак копирования и вставки или объединения кода.

[6] Примеры проверок безопасности для puppet-lint см. на странице *https://github.com/floek/puppet-lint-security-plugins*.

Инспекция кода – пожалуй, лучший способ поиска проблем с конкурентностью и синхронизацией (многопоточность, блокировка, момент проверки, момент использования) и ошибок использования криптографических функций. Ниже мы увидим, что это, наверное, единственный способ найти черные ходы и бомбы замедленного действия.

Но практика и исследования показывают, что вместо поиска дефектов инспекторы тратят большую часть времени на выявление проблем, из-за которых код трудно понять и сопровождать. Они цепляются к таким вещам, как неудачный выбор имени, сбивающие с толку и отсутствующие комментарии, плохо структурированный, а также «мертвый» и неиспользуемый код[7].

Есть несколько причин, из-за которых во время инспекции кода выявляется не так много дефектов, как должно бы:

- для надлежащего проведения инспекции нужно время;
- разобраться в чужом коде нелегко (особенно если речь идет о технически сложном или разностороннем проекте, а также о большой кодовой базе);
- искать уязвимости в чужом коде еще труднее.

Рассмотрим каждую из этих причин более подробно.

Для инспекции нужно время

Во-первых, требуется время для тщательной инспекции кода. Команда, в т. ч. владелец продукта, и руководство должны понимать, что на инспекцию кода будет уходить значительная часть рабочего времени разработчиков и что ожидание инспекции кода замедлит поставку.

В Microsoft разработчики тратят в среднем от двух до шести часов в неделю на инспекцию изменений кода. В среднем разработчик ждет отзыва на свой код один день. Но в некоторых случаях инспекторам требуется несколько дней, а то и недель на вынесение вердикта. Такие изменения нельзя записать в основную ветвь, нельзя протестировать и нельзя учитывать в расписании поставки.

Поэтому так важно, чтобы команда соблюдала дисциплину и выработала культуру коллективной работы и поддержки друг друга, цель которой – сделать так, чтобы каждый был готов заступить на вахту и помочь товарищам в проведении инспекций. А еще важнее, чтобы все члены команды подходили к инспекции со всей серьезностью и были готовы

[7] Mika V. Mantyla, Casper Lassenius «What Types of Defects Are Really Discovered in Code Reviews?», IEEE Transactions on Software Engineering, Volume 35, Issue 3, May 2009: 430–448 (*https://dl.acm.org/citation.cfm?id=1592371*).

приложить все силы, чтобы работа была сделана на совесть. Поскольку, как мы скоро увидим, провести инспекцию как следует нелегко.

Разобраться в чужом коде трудно

Одна из основных причин, почему инспекторы находят не так много дефектов, как хотелось бы, состоит в том, что для нахождения проблемы нужно понимать, что код делает и почему он был изменен.

На то, чтобы разобраться в этом, уходит большая часть времени инспектора, и именно поэтому замечания чаще всего касаются удобочитаемости (выбор имен, комментарии, форматирование) и того, как сделать код проще или чище, – всего, чего угодно, только не фундаментальных проблем.

Именно в этом отношении могут помочь инструменты анализа кода, которые понуждают придерживаться выбранного стиля кодирования и соглашений, что делает код чище и единообразнее.

Если инспекторы незнакомы с кодом, то для понимания контекста им понадобится прочитать больше кода, а это значит, что выделенное на инспекцию время закончится раньше, чем будет найдено что-то действительно существенное.

И это еще одна причина, по которой инспекцию лучше поручать опытным членам команды. Человек, который раньше работал с кодом – изменял его сам или уже инспектировал, – имеет очевидное преимущество перед тем, кто в глаза не видел кода. Поэтому он будет работать намного быстрее, и его отзывы дадут больший эффект[8].

Искать уязвимости еще труднее

Как мы уже видели, искать функциональные или логические дефекты в чужом коде трудно. Инспектор должен понимать цель изменения и его контекст, а также достаточно хорошо понимать код, чтобы найти ошибки в логике или реализации.

Искать уязвимости еще труднее. К уже упомянутым трудностям – необходимости понимать назначение и контекст, структуру и стиль коди-

[8] Amiangshu Bosu, Michaela Greiler, Christian Bird «Characteristics of Useful Code Reviews: An Empirical Study at Microsoft», Proceedings of the International Conference on Mining Software Repositories, May 1, 2015 (*https://www.microsoft.com/en-us/research/publication/characteristics-of-useful-code-reviews-an-empirical-study-at-microsoft/*).

рования и т. д. – прибавляется владение методами безопасного кодирования и знания о том, что нужно искать.

Познакомимся с некоторыми исследованиями, чтобы понять, насколько трудно инспектировать код на предмет выявления уязвимостей.

В исследовании 2013 года 30 программистов на PHP (включая специалистов по безопасности) пригласили, чтобы вручную проинспектировать безопасность небольшого веб-приложения, в котором было 6 известных уязвимостей. На работу было отведено 12 часов, использовать инструменты статического анализа не разрешалось.

Ни один инспектор не смог найти все известные дефекты (хотя несколько человек нашли новую уязвимость типа XSS, о которой исследователи не знали), а 20% инспекторов не нашли ни одной уязвимости. Большой опыт кодирования не означает, что будет найдено больше ошибок, связанных с безопасностью, поскольку даже опытные разработчики не понимают, что искать в процессе инспекции на безопасность[9].

Насколько трудны и важны инспекции кода в области безопасности, можно оценить, взглянув на одну из самых опасных уязвимостей, найденных за последние несколько лет: ошибку Heartbleed в коде OpenSSL. Эта критическая брешь в механизме согласования SSL была вызвана переполнением буфера в коде на C – простой ошибкой низкоуровневого кодирования.

Удивительно, что, несмотря на то что исходное изменение инспектировалось кем-то, работавшим над кодом OpenSSL, прежде, и на то, что код был открыт для любого, желающего его прочитать, понадобилось больше двух лет, чтобы группа безопасности в Google нашла ошибку в процессе аудита кода[10].

Автор изменения, первый инспектор, несколько инструментов статического анализа и почти все члена сообщества, скачавшие и использовавшие этот код, пропустили ошибку просто потому, что код было слишком трудно понять, а значит, слишком трудно изменить безопасно. Так не должно быть для критического в отношении безопасности кода.

Понять код, сделать его яснее и чище, скрупулезно соблюдать наставление по кодированию, постоянно подвергать рефакторингу – все это позволит упростить изменение кода программисту, следующему за вами. А значит, сделает код безопаснее.

---

[9] Anne Edmundson, et al. «An Empirical Study on the Effectiveness of Security Code Review», ESSoS (2013) (*https://www.cs.princeton.edu/~annee/pdf/coderev-essos13.pdf*).

[10] David A. Wheeler «How to prevent the next Heartbleed», 29.01.2017 (*http://www.dwheeler.com/essays/heartbleed.html*).

Внедрение инспекций кода на безопасность

Чего ожидать от инспекции кода на безопасность? Как сделать ее эффективной?

Внедряйте инспекции кода на безопасность в работу команды постепенно, без нажима. Работайте в русле гибких методик. Начните с малого и подкрепляйте удачные способы работы. Постоянно учитесь и совершенствуйтесь. Чтобы изменить привычные способы мышления и труда, нужно время – но в этом и состоит суть гибкой и бережливой разработки.

Опирайтесь на то, что команда делает или должна делать

Если команда уже практикует парное программирование или дружественную оценку, помогите ей понять, как можно включить в этот процесс безопасность, следуя приведенным в этой книге рекомендациям. Работайте в тесном контакте с руководством, владельцем продукта и Scrum-мастером, чтобы те выделили команде достаточно времени для учебы на рабочем месте.

Позаботьтесь о том, чтобы у разработчиков была подготовка в области основ безопасного кодирования, чтобы они могли писать более безопасный код и знали, на что обращать внимание в процессе инспекции кода. Воспользуйтесь бесплатными учебными курсами SAFECode и открытой информацией, публикуемой OWASP. Когда дойдет до обучения, проявляйте прагматизм. Ваша цель – не обучить каждого члена команды низкоуровневым деталям правильного шифрования. Но каждый должен понимать, когда и как правильно вызывать криптографические функции.

Не полагайтесь на инспекции, проведенные неопытными членами команды. Настаивайте на привлечении опытных, квалифицированных инспекторов, особенно если речь идет о высокорисковом коде.

Если команда не практикует парное программирование или дружественную оценку, то:

- найдите людей, которые пользуются уважением и доверием команды, старших разработчиков, которые думают о высоком качестве кода и любят принимать вызовы;
- сделайте все возможное, чтобы убедить этих людей (и их начальство), что инспекция кода – важное и нужное приложение их знаний и умений, и начните с высокорискового каркасного кода и средств, имеющих отношение к безопасности;

- позаботьтесь о том, чтобы у них было достаточно информации и знаний о коде, чтобы понять, что искать;
- если вы работаете в регулируемой отрасли, то апеллируйте к необходимости соблюдать нормативные требования – если больше ничего не помогает.

Следите за тем, чтобы инспекторы изучали также тесты, особенно для высокорискового кода. Код без тестов опасен тем, что может перестать работать. Если вам небезразличны надежность и безопасность кода, то не следует упускать из виду качество и покрытие автоматизированных тестов, а также забывать о негативных тестах. Мы еще поговорим об этом в главе 11.

А если вы печетесь о самодостаточном развитии системы в долгосрочной перспективе, то должны инспектировать также тесты, дабы иметь уверенность, что они хорошо написаны, понятны и удобны для сопровождения.

Делайте инспекцию кода максимально легким делом

Внедряйте в команду статический анализ, частично выделяя средства из бюджета на безопасность, чтобы помочь разработчикам писать более качественный код. Если нет денег на покупку коммерческого программного обеспечения, обратитесь к вышеупомянутым инструментам с открытым исходным кодом. Не забывайте про наши рекомендации о том, как научить инженеров эффективно пользоваться этими инструментами.

Если команда работает с платформой коллективной инспекции кода, например Gerrit или Review Board, то обратите себе на пользу сложившиеся технологические процессы и собранные данные, чтобы:

- обеспечить гарантированную инспекцию высокорискового кода;
- выборочно контролируйте инспекции, следя за тем, чтобы к ним относились ответственно;
- используйте информацию в комментариях, в качестве контрольного журнала для аудиторов соответствия нормативным требованиям;
- включайте результаты статического анализа в виде комментариев к инспекциям кода, чтобы помочь инспекторам находить больше проблем.

Опирайтесь на коллективное владение кодом в гибких методиках

Если код открыт для инспекции и работы всем членам команды, то включите специалистов по безопасности в команду. Тогда им не нужно будет изобретать особые причины, чтобы изучить код на предмет наличия проблем. И если они знают, что делают, и пользуются доверием других членов команды, то им не придется получать разрешение на исправление уязвимостей – лишь бы только соблюдали принятые в команде соглашения и технологические процессы.

Рефакторинг: поддержание простоты и безопасности кода

В ясном, чистом, единообразно написанном коде появление ошибок (в т. ч. связанных с безопасностью) менее вероятно, а изменять и исправлять его проще и безопаснее. Инспектировать такой код тоже гораздо легче: если инспектор не понимает код, то он только зря будет тратить время на то, чтобы уяснить, что же в нем происходит. И упустит шанс найти дефекты.

Именно поэтому после обнаружения Heartbleed команда OpenSSL потратила несколько месяцев, чтобы переформатировать код, удалить неиспользуемые участки и вообще привести всё в порядок. Команда сочла необходимым сделать это, прежде чем переходить к более важным и фундаментальным улучшениям[11].

По счастью, многие гибкие команды, особенно практикующие экстремальное программирование, понимают важность написания чистого кода и соблюдения единых правил кодирования. Если вы работаете в такой команде, то исполнять функции инспектора будет гораздо проще – и полезнее.

Если нет, то постарайтесь убедить команду в пользе рефакторинга – еще одной стандартной гибкой практике. *Рефакторинг* – четко определенный согласованный набор паттернов и методов вычистки и реструктурирования кода мелкими шагами – встроен в большинство современных IDE. Программист может переименовывать поля, методы и классы, выделять методы и классы, изменять сигнатуры методов и вносить другие, более крупные структурные изменения безопасно и с предсказуемым результатом – особенно если имеется страховочная сеть хороших регрессионных тестов, но на эту тему мы поговорим в следующей главе.

---

[11] Matt Caswell «Code Reformat Finished», OpenSSL Blog, Feb 11th, 2015 (*https://www.openssl.org/blog/blog/2015/02/11/code-reformat-finished/*).

Дополнительная информация о рефакторинге

Дополнительные сведения о написании чистого кода и рефакторинге можно почерпнуть из следующих книг, которые обязан прочитать любой программист:

1) Bob Martin «Clean Code, a Handbook of Agile Software Craftsmanship», Prentice Hall. (Роберт К. Мартин «Чистый код. Создание, анализ и рефакторинг», Питер, 2017);
2) Kent Beck and Martin Fowler «Refactoring: Improving the Design of Existing Code», Addison-Wesley Professional. (Мартин Фаулер «Рефакторинг. Улучшение проекта существующего кода», Вильямс, 2017);
3) Michael Feathers «Working Effectively with Legacy Code», Prentice Hall. (Майкл К. Физерс «Эффективная работа с унаследованным кодом», Вильямс, 2016).

Применяя средства рефакторинга в своих IDE, инспекторы могут производить рефакторинг на скорую руку, отбрасывая полученные результаты, пока не станут понимать код лучше. А могут предлагать варианты рефакторинга в комментариях к инспекции или передавать автору внесенные изменения в виде отдельной заплаты. Но если у инспектора возникает желание переформатировать код в своей IDE или подвергнуть его рефакторингу до начала инспекции, значит, команда что-то делает неправильно.

Базовые вещи – вот путь к безопасному и надежному коду

Если вы успешно миновали стадию начального понимания – потому ли, что код был чистым изначально, или потому что вы сами его вычистили, или потому что работали с ним настолько долго, что докопались до смысла, – тогда можно приступить к инспекции того, что код реально делает – или не делает.

По ходу дела инспектор, даже не являющийся крупным специалистом по безопасному кодированию, может обращать внимание на следующие вещи:

1) зашитые в код пароли и другие учетные данные, пути, магические числа и другие вещи, которые могут представлять опасность;
2) тестовый или отладочный код, оставленный по небрежности;

Все входные данные – зло

Если говорить о безопасности, то самое важное, на что должны обращать внимание инспекторы, – безопасное обращение с данными. Майкл Ховард (Michael Howard) из Microsoft, один из авторов книги «Writing Secure Code» (Microsoft Press) (Майкл Ховард, Дэвид Лебланк «Защищенный код», Русская редакция, 2005), говорил, что если нужно оставить всего одно правило, обязательное для разработчиков, то это осознание того, что «все входные данные – зло».

Проверка типа, длины и диапазона значений данных – часть защитного программирования. Это утомительно, но каждый разработчик должен научиться понимать, как это важно – в ходе обучения профессии или на собственном горьком опыте, когда пришлось разбираться в причинах краха программы или устранять уязвимость. По счастью, это та область, где могут помочь инструменты статического анализа: анализ зараженности (taint analysis) и прослеживание потока данных могут выявить отсутствующие проверки.

Но контроля входных данных недостаточно, чтобы сделать современное мобильное или веб-приложение безопасным, поскольку у базовых технологий, на которые мы полагаемся для решения задач, – веб-браузеров, движков баз данных, интерпретаторов XML – есть проблемы с проведением четкого различия между командами и данными. Атакующие научились использовать это для атак методом межсайтового скриптинга, внедрения SQL и других, не менее опасных.

В дополнение к контролю данные необходимо кодировать, экранировать или подвергать еще какой-то обработке, чтобы сделать их безопасными в браузере или при записи в базу данных. Применение подготовленных команд SQL предотвращает внедрение, поскольку четко разграничивает команды и подставляемые в них данные. Современные веб-каркасы, например Angular.js, Ember, React, Rails и Play, предоставляют встроенные средства защиты от XSS и других атак внедрением, нужно только использовать их правильно – и при условии своевременного обновления после обнаружения новых уязвимостей.

Для глубокоэшелонированной обороны можно воспользоваться безопасными заголовками HTTP, например Content Security Policy (CSP). Ознакомьтесь с информацией о безопасных заголовках, подготовленной технической командой Twitter: *https://github.com/twitter/secureheaders*.

3) проверять, что секретные данные зашифрованы, представлены в виде маркеров или подвергнуты еще какой-то безопасной обработке. Вопрос о том, правильно ли производится шифрование, следует оставить специалисту, но применение шифрования как такового – прикладная проблема, которую может разглядеть любой член команды, знакомый с требованиями;
4) неправильная обработка ошибок. Обработка ошибок – еще один аспект защитного программирования, а поскольку протестировать его трудно и занимаются этим редко, очень важно обращать внимание на обработку ошибок в процессе инспекции кода;
5) проверка того, что контроль доступа применяется правильно и единообразно. Что позволено видеть и делать различным пользователям системы, определяется бизнес-правилами, это не задача для дорогих магов в остроконечных шапках, колдующих над безопасностью;
6) последовательный аудит всех операций добавления, изменения и удаления;
7) последовательное применение одобренных каркасов и стандартных библиотек, особенно при работе с криптографическими функциями и кодировании выходной информации;
8) потокобезопасность, включая проблему момента проверки и момента использования, а также взаимоблокировки: эти ошибки трудно найти во время тестирования, поэтому на конкурентность, синхронизацию и блокировки следует обращать особое внимание во время инспекции.

Ни один из пунктов этого списка не требует специальных знаний в области безопасности. Не нужно привлекать аудитора безопасности, чтобы убедиться в том, что команда пишет чистый, надежный и защищенный код.

Инспекция функций и средств контроля, относящихся к безопасности

Инспекция вещей, относящихся к безопасности, гораздо сложнее, чем прочего кода приложения. Чтобы найти проблемы, инспектор должен понимать все нюансы безопасности, а также уметь читать код. По счастью, этот код изменяется нечасто, и если все сделано правильно, то большая его часть написана с использованием механизмов, встроенных в мобильный или веб-каркас, либо с помощью специализирован-

ных библиотек, например Apache Shiro (*http://shiro.apache.org/*) или комплекта криптографических средств Google KeyCzar (*https://github.com/google/keyczar*).

Пожалуй, лучшим справочным пособием по инспекции кода на предмет безопасности являются контрольные списки в проекте OWASP ASVS (*https://www.owasp.org/index.php/Category:OWASP_Application_Security_Verification_Standard_Project*). Хотя ASVS задуман как средство аудита, этот список может пригодиться инспекторам – и кодировщикам – для проверки того, что все важные вопросы, особенно касающиеся средств контроля безопасности, учтены. Пропустите нудные вступительные слова об аудите и сразу переходите к контрольным спискам.

Давайте посмотрим, как использовать ASVS при проведении инспекции важных функций и соображений, относящихся к безопасности. В разделе об аутентификации перечислено, на что обращать внимание, например:

- все страницы и прочие ресурсы по умолчанию требуют аутентификации, за исключением тех, которые явно сделаны открытыми для анонимных пользователей;
- поля, содержащие пароли, не отображаются при вводе;
- аутентификация производится на стороне сервера, а не только на стороне клиента;
- ошибки в коде аутентификации безопасны, т. е. противник не сможет войти в систему в случае ошибки (иными словами, по умолчанию доступ запрещен и разрешается только в случае, когда все шаги завершились успешно);
- все подозрительные решения в процессе аутентификации протоколируются.

В ASVS рассматривается управление сеансами, контроль доступа, обработка вредоносных входных данных, криптографическая защита данных на устройствах хранения, обработка и протоколирование ошибок, защита данных, безопасность связи и т. д.

Инспекция кода на предмет угроз от инсайдеров

Угрозы со стороны инсайдера-злоумышленника, который может заложить в систему бомбу замедленного действия, троян или еще какой-то вредоносный код либо вмешаться в логику приложения с целью совершения мошенничества, сравнительно невелики, но все же реальны.

К счастью, инспекция кода для поиска «честных» ошибок поможет отловить и ограничить риск многих инсайдерских угроз. Не важно, совершена ошибка случайно и по глупости или обдуманно и с дурными намерениями, искать-то все равно нужно то, что Брентон Колер из компании Cigital называет «красными флажками»[12].

1. Мелкие случайные (или кажущиеся случайными) ошибки в функциях, относящихся к безопасности, включая аутентификацию и управление сеансами, контроль доступа, криптографию и обращение с секретными данными.
   Как отмечает Брюс Шнайер, тривиальные, но имеющие крайне разрушительные последствия ошибки типа допущенной Apple ошибки «goto fail» в коде SSL, могут стать поводом насторожиться: «Было ли это сделано специально? Понятия не имею. Но если бы я хотел сделать нечто подобное специально, то поступил бы именно так»[13].
2. Поддержка черных ходов (или чего-то, что можно было бы использовать как черный ход), например зашитых в код или конфигурационные файлы URL, IP или других адресов, идентификаторов и паролей, сверток пароля либо ключей пользователей. Скрытые административные команды, параметры и опции на этапе выполнения.
   Такой код часто предназначен для поддержки во время эксплуатации и поиска неисправностей (либо случайно оставлен после отладки и тестирования), но может быть использован и во зло. В любом случае, черные ходы – потенциально опасные бреши, которыми может воспользоваться противник.
3. Тестовый, отладочный или диагностический код.
4. Встроенные команды оболочки.
5. Логические ошибки при обработке денежных сумм (например, перевод «копеек» на свой счет, см. *https://en.wikipedia.org/wiki/Salami_slicing*), или ошибки при задании лимитов или обработке данных кредитной карты, или в функциях обработки команд и контроля, или в особо ответственном коде, доступном из сети.
6. Некорректная обработка ошибок или исключений, которая может оставить брешь в системе.
7. Отсутствие протоколирования или аудита, лакуны в порядковой нумерации.

---

[12] Brenton Kohler «How to eliminate malicious code within your software supply chain», Synopsys, March 9, 2015 (*https://www.synopsys.com/blogs/software-security/eliminate-malicious-code-from-software-supply-chain*)

[13] Bruce Schneier «Was the iOS SSL Flaw Deliberate?», Schneier on Security, February 27, 2014.

8. Чрезмерно заковыристый или не имеющий смысла код, особенно если он относится к хронометрируемой логике, криптографическим или иным высокорисковым функциям. Инсайдер-злоумышленник, вероятно, захочет запутать свои намерения. К этому моменту должно быть ясно, что даже если подобный код написан без злого умысла, все равно ему не место в системе.
9. Самомодифицируемый код – по тем же причинам, что и выше.

Если вас беспокоит возможный тайный сговор между разработчиками, то можно осуществлять регулярную ротацию инспекторов или назначать их случайным образом, чтобы убедить команду подходить к этому серьезно. Если ставки достаточно высоки, то можно нанимать внешних инспекторов для аудита кода, как делает организация Linux Foundation Core Infrastructure Initiative, которая оплачивает экспертный аудит пакетов OpenSSL, NTP и OpenSSH.

Необходимо также осуществлять контроль на всех этапах: запись в репозиторий, сборка, тестирование и развертывание, чтобы гарантировать отсутствие манипуляций по пути и быть уверенным в том, что развернуто именно то, что было записано, собрано и протестировано разработчиками. В главе 13 описаны рекомендуемые действия по блокировке репозиториев и сборочных конвейеров. Тщательно относитесь к управлению секретами и ключами. Применяйте контрольные суммы или цифровые подписи, а также средства обнаружения изменений типа OSSEC, чтобы выявлять неожиданные и неавторизованные изменения важных конфигурационных параметров и кода.

Но все это следует делать в любом случае. Эти меры минимизируют риск инсайдерских атак и помогают обнаруживать, что противник каким-то образом скомпрометировал сеть или среду разработки и сборки.

Сухой остаток

Инспекции кода могут быть действенным инструментом обнаружения уязвимостей на ранних стадиях, при условии что проводятся надлежащим образом.

- Инспекции следует проводить так, чтобы они не тормозили работу команды. Требование проводить инспекции только силами группы безопасности создает узкое место, которое разработчики всеми силами будут стараться обойти.
- Дружественная оценка непосредственно перед записью кода в главную ветвь с помощью запросов на включение кода в Git или

похожей технологической операции – наверное, самый эффективный способ гарантировать, что инспекция кода действительно имеет место.

Группу безопасности следует привлекать только к инспекциям высокорисковых функций и каркасного кода, имеющего отношение к безопасности, но такой код обычно изменяется редко.

- Такие инструменты инспекции кода, как Gerrit, Review Board или Phabricator, умеют автоматически навязывать последовательный технологический процесс, что делает инспекции проще и прозрачнее, особенно в распределенных командах. Члены команды могут видеть отзывы друг друга и опираться на них, а ведение электронного учета удовлетворит аудиторов.
- Инспекция кода отнимает время и требует серьезного умственного напряжения. Инспекции более эффективны, если проводятся часто и понемногу. К счастью, именно так гибкие команды и вносят большинство изменений.
- Обучайте разработчиков безопасному кодированию, раздайте краткие контрольные списки, чтобы они понимали, чего избегать при написании кода и на что обращать внимание во время инспекции.
- Разработчики должны чувствовать себя спокойно во время инспекции: когда просят дать отзыв о своей работе, когда дают отзыв, когда задают вопросы, встретив что-то непонятное. Следите за тем, чтобы инспекторы занимались кодом, а не кодировщиком.

  Инспекции кода – удачная возможность для непрерывного обучения и совершенствования. Используйте заключения инспекторов для улучшения наставления по кодированию и шаблонов. Обменивайтесь полученным опытом в процессе ретроспективного анализа.
- Инспектировать нужно любой код, вне зависимости от автора, а особенно код, написанный старшими членами команды, потому что они часто берут на себя более трудные и рискованные задачи. Помогайте команде выявлять высокорисковый код (помечайте высокорисковые истории в Jira, Pivotal Tracker и других средствах учета историй) и следите за тем, чтобы этот код инспектировали один или несколько опытных разработчиков.

  Код следует инспектировать, даже если он был создан в процессе парного программирования, поскольку в ходе инспекции обнаруживают не те проблемы, что при парном программировании.

- Автоматизированные средства статического анализа кода – не замена ручной инспекции. Тут уместно И, а не ИЛИ. Пользуйтесь инструментами статического анализа, чтобы составить хорошее наставление по кодированию и выявить небезопасные зависимости, опасные функции и типичные ошибки кодирования. Лучше прогонять эти средства до того, как код увидят инспекторы, это сделает их работу проще, быстрее и гораздо эффективнее.
- Хотя автоматизированные инструменты статического анализа кажутся дешевым и легким способом обеспечить инспекцию всего кода (по крайней мере, если вы пользуетесь бесплатными программами с открытым исходным кодом), это не просто «включил и заработало». Их следует внедрять аккуратно и осторожно, объясняя разработчикам, что инструмент делает, и побуждая их использовать его последовательно.

  Поскольку разные инструменты находят разные проблемы и устроены по-разному, для получения хорошего покрытия кода нужно использовать несколько инструментов.
- Разработчики ненавидят ложноположительные предупреждения. Выделите время, чтобы разобраться в работе инструментов статического анализа и настроить их так, чтобы получать эффективную и надежную отдачу.
- Организуйте статический анализ по принципу самообслуживания. Поместите инструменты на рабочее место каждого разработчика, чтобы они могли видеть ошибки в IDE прямо в процессе ввода кода. Включите статический анализ в сборочные конвейеры непрерывной интеграции или непрерывной поставки.
- Не забывайте инспектировать конфигурационные файлы и тесты: это важная часть кодовой базы, с которой нужно обращаться, как с любым другим кодом.

Инспекции и статический анализ кода должны войти в «определение готовности»: контракт между членами команды, описывающий, когда работу над функциональностью или исправлениями можно считать законченной и переходить к следующей задаче. Команда должна решить, какой код подлежит инспектированию (все изменение или только высокорисковый код), сколько привлекать инспекторов, когда проводить инспекции (до или после записи кода в репозиторий), какие средства автоматизированной инспекции включать в конвейер непрерывной интеграции и что делать с результатами инспекции.

Глава 11

Гибкое тестирование безопасности

Одно из важнейших изменений, привнесенных гибкой разработкой, – отношение к тестированию. Поскольку гибкая команда продвигается вперед очень быстро, тестирование в большой степени, а зачастую и целиком, полагается на автоматизацию. Если работающее ПО нужно поставлять каждую неделю или если любое изменение передается в эксплуатацию немедленно (методика непрерывного развертывания), то ручное тестирование не «прокатит».

Передавать код независимой команде тестировщиков в конце спринта в гибких методиках считается антипаттерном. Тестирование следует производить в том же процессе, что и изменения. Укрепление (hardening) спринтов, когда команда отводит отдельное время, чтобы заняться тестированием, отладкой и исправлением кода до передачи его в эксплуатацию, – практика, которой большинство современных команд также избегает.

Организации, которые полагаются на тестирование силами независимых команд, на приостановку процесса для ручного тестирования на проникновение, на плановое сканирование или на аудит безопасности, должны пересмотреть свой подход к работе.

Как производится тестирование в гибких методиках?

Во многих командах, практикующих гибкие методики и DevOps, нет специальных тестировщиков. Разработчики принимают на себя ответственность за тестирование своего кода, потому что вынуждены это де-

лать. Владелец продукта или еще какой-то член команды, играющий роль «заказчика», может писать сценарии приемочного тестирования, пользуясь предоставленными командой инструментами. Но за все остальные тесты и за их успешное выполнение отвечают разработчики. Тестирование становится неотъемлемой частью кодирования, а не отдельным этапом, как в каскадной модели.

Как меняются роли и правила в гибком тестировании

В методике быстрой инкрементной гибкой разработки роли разработчиков и тестировщиков существенно изменились и продолжают меняться, по мере того как команда принимает на вооружение непрерывную поставку и другие практики DevOps.

Если вам интересно узнать об этих изменениях, о том, что они означают для разработчиков, руководителей проектов и особенно для тестировщиков, и о том, как извлечь максимум пользы из тестирования в гибкой среде, то рекомендуем книги «Agile Testing: a Practical Guide for Testers and Agile Teams» и «More Agile Testing: Learning Journeys for the Whole Team», написанные Лайзой Криспин (Lisa Crispin) и Дженет Грегори (Janet Gregory) (издательство Addison-Wesley Professional).

Поскольку разработчики постоянно вносят изменения в код, когда ставят эксперименты, производят рефакторинг и реагируют на отзывы, им необходимо как-то предотвратить непреднамеренное нарушение текущего поведения системы. Для этого натягивается страховочная сетка, состоящая из автоматизированных регрессионных тестов, которые прогоняются несколько раз в день. Эти тесты должны быть:

- дешевыми и простыми, чтобы команда могла запускать их часто;
- быстрыми и эффективными, чтобы команда действительно запускала их часто;
- повторяемыми и предсказуемыми, чтобы команда могла полагаться на результаты.

Те же требования относятся и к тестированию безопасности. Тесты безопасности должны быть быстрыми, повторяемыми, эффективными и по возможности автоматизированными. В той мере, в какой это достижимо, тестирование безопасности должно укладываться в технологические процессы точно так же, как остальные виды тестирования, – не создавая ненужной задержки и не увеличивая затрат.

Кто допускает ошибки, тот побежден

Все мы знаем о сильной зависимости между качеством кода и безопасностью. Чем больше в программе ошибок, тем хуже обстоит дело с безопасностью.

Исследования доказали, что до половины программных уязвимостей вызвано простыми ошибками кодирования. Это не просчеты при проектировании и не непонимание черной магии безопасности. Просто тупые ошибки из-за небрежности, например в результате копирования и вставки или объединения кода, непроверенных входных данных, неправильной или вовсе отсутствующей обработки ошибок, не там поставленных скобок.

В институте программной инженерии Карнеги-Меллона выяснили, что от 1 до 5 процентов программных дефектов составляют уязвимости[1]. Следовательно, зная, сколько ошибок имеется в программе, можно приблизительно оценить, насколько она безопасна.

Учитывая, что в большинстве программ встречается от 15 до 50 ошибок на каждые 1000 строк кода (и это даже после того, как код был подвергнут инспекции и тестированию), в небольшом мобильном или веб-приложении, содержащем, скажем, 50 000 строк кода, легко может насчитываться свыше 100 уязвимостей[2]. Напомним, что почти все современные приложения включают большой объем открытого исходного кода, поэтому, даже если вы сами все делаете правильно с точки зрения безопасности, создатели открытого кода могли быть не столь осторожны, поэтому разумнее считать, что количество уязвимостей еще больше. Проблемы качества ПО – и вместе с ними риски безопасности – растут, как снежный ком, с увеличением размера кодовой базы. В крупных системах плотность дефектов гораздо выше, а значит, эти системы гораздо более уязвимы.

Многие резонансные уязвимости, включая Heartbleed и ошибку «goto fail» (EX1-A) в реализации SSL компанией Apple, были вызваны ошибками кодирования, которые могли и должны были быть найдены в ходе инспекции кода или при дисциплинированном применении автономного тестирования. Никаких потаенных знаний о безопасности не требовалось. Просто квалифицированное защитное кодирование и пристальное внимание к тестированию.

---

[1] Carol Woody, Ph. D.; Robert Ellison, Ph. D.; William Nichols, Ph. D. «Predicting Software Assurance Using Quality and Reliability Measures» Software Engineering Institute, Technical Note, December 2014 (*https://resources.sei.cmu.edu/asset_files/TechnicalNote/2014_004_001_428597.pdf*).

[2] Vinnie Murdico «Bugs per lines of code», Tester's World, April 8, 2007 (*http://amartester.blogspot.com/2007/04/bugs-per-lines-of-code.html*).

Пример 11.1. Apple Goto Fail... сможете найти ошибку?

```
static OSStatus
SSLVerifySignedServerKeyExchange(SSLContext *ctx, bool isRsa, SSLBuffer signedParams,
                                 uint8_t *signature, UInt16 signatureLen)
{
   OSStatus err;
   ...

   if ((err = SSLHashSHA1.update(&hashCtx, &serverRandom)) != 0)
      goto fail;
   if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0)
      goto fail;
      goto fail;
   if ((err = SSLHashSHA1.final(&hashCtx, &hashOut)) != 0)
      goto fail;
   ...

fail:
   SSLFreeBuffer(&signedHashes);
SSLFreeBuffer(&hashCtx);
   return err;
}
```

Ясно, что принятая командой программа обеспечения безопасности должна базироваться на программах контроля качества, инспекций и тестирования, которые уже внедрены или должны быть внедрены. Рассмотрим, как производится тестирование в методике гибкой разработки и где в этом процессе следует учесть потребности безопасности.

Пирамида гибкого тестирования

Мы уже встречались с пирамидой гибкого тестирования (*https://martinfowler.com/bliki/TestPyramid.html*) на страницах этой книги. Теперь разберемся, как она работает (см. рис. 11.1).

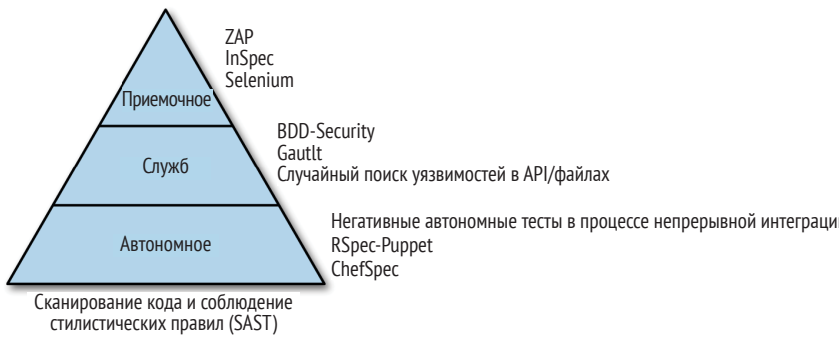

Рис. 11.1. Тестирование безопасности в контексте пирамиды гибкого тестирования

Пирамида тестирования предполагает большое количество (тысячи, десятки тысяч) автоматизированных низкоуровневых автономных тестов в основании. Это тесты методом белого ящика на уровне методов или функций. Они пишутся разработчиками с целью убедиться, что код делает то, что должен. При этом используется каркас тестирования типа xUnit (JUnit для Java) или TestNG.

Автономные тесты проще всего писать, дешевле всего прогонять и легче всего изменять (и это хорошо, потому что эти тесты приходится изменять чаще других).

Тестовые двойники, mock-объекты и заглушки в автономном тестировании

В процессе создания автономных тестов разработчик пишет тестовый код для каждого метода или функции и создает тестовые фикстуры или тестовые двойники – mock-объекты и заглушки, – чтобы заменить зависимости тестовыми реализациями.

Заглушка (stub) просто подменяет зависимость тестируемой функции. Она возвращает фиксированные ответы и может сохранять информацию о том, какие данные получала от функции в процессе тестирования. В одном тесте может быть несколько заглушек.

Mock-объект – это «умный» подставной объект, который решает, должен ли автономный тест завершиться успешно или неудачно. Он проверяет, сделала ли тестируемая функция то, что от нее ожидается, и завершает тест с ошибкой, если фактический результат не совпадает с ожидаемым.

В середине пирамиды находятся API, рассматриваемый как черный ящик, и *интеграционные тесты*, которые проверяют поведение на уровне служб и опубликованные API взаимодействия служб. Для написания этих тестов служат такие каркасы, как FitNesse, Cucumber или JBehave.

Поскольку инструменты типа Cucumber применяются также для создания высокоуровневых приемочных тестов с применением пользовательского интерфейса, граница между этими уровнями пирамиды иногда бывает размыта. Главное, что нужно понимать, – это тот факт, что тестирование на этом уровне направлено не на взаимодействие с пользователем, а на взаимодействие между службами, проверку API служб и предметной логики, занимающей более высокие уровни абстракции, чем автономные тесты.

Наверху пирамиды расположены *приемочные тесты* уровня пользовательского интерфейса: комплексные, ориентированные на пользователя тесты, которые проверяют важную функциональность и ключевые сценарии работы с приложением. Сквозные приемочные тесты призваны продемонстрировать пользователям и владельцам системы, что она готова к работе. Писать и прогонять их труднее всего, и к тому же они самые хрупкие (поскольку проверяют много разных «подвижных деталей»). Поэтому количество таких тестов должно быть сведено к минимуму.

Некоторые команды пишут приемочные тесты в виде скриптов с помощью таких инструментов, как Selenium WebDriver, Sahi или Watir. Скрипты может писать владелец продукта, бизнес-аналитик или тестировщик, а реализуются они командой разработчиков. Приемочное тестирование можно также производить вручную в процессе демонстрации заказчику, особенно когда тесты обходятся дорого или с трудом поддаются автоматизации.

Пирамида гибкого тестирования – это альтернативный способ решить, куда и как направить усилия по тестированию. Она противоположна «тестовому рожку мороженого», подразумевающему большой объем ручного тестирования, или тестам типа запись–воспроизведение на уровне пользовательского интерфейса, которые прогоняются независимой командой тестировщиков, в то время как автоматизированных тестов низкого уровня, написанных разработчиками, мало или нет совсем.

Автономное тестирование и TDD

Мы начнем с нижнего уровня пирамиды и попытаемся понять, как и почему разработчики пишут автономные тесты, какие последствия это влечет для безопасности системы.

Разработчики пишут автономные тесты, чтобы доказать самим себе, что код делает именно то, что должен делать. Обычно эти тесты пишутся после внесения изменений в код и прогоняются до записи кода в репозиторий. Тесты сохраняются в репозитории вместе с кодом, так чтобы они принадлежали всей команде, и прогоняются в процессе непрерывной интеграции.

Многие разработчики пишут один или несколько автономных тестов как первый шаг на пути исправления ошибки: сначала пишут тест для воспроизведения ошибки, затем прогоняют его, чтобы убедиться, что он действительно отлавливает ошибку, потом исправляют ошибку и

снова прогоняют тест, чтобы доказать, что исправление сделало свое дело (а если ошибка по какой-то случайности появится снова, то этот тест обнаружит ее в процессе будущего регрессионного тестирования).

Некоторые команды (особенно практикующие экстремальное программирование) настолько «одержимы тестами», что делают следующий шаг, называя свой подход «разработкой через тестирование» (test-driven development – TDD). Прежде чем писать сам код, они пишут тесты, описывающие, что этот код должен делать. Они прогоняют тесты, убеждаются, что все они не проходят, затем пишут код и прогоняют тесты снова, добавляя новый код между прогонами. Так и переходят от написания тестов к написанию кода и обратно. Тесты направляют их мысли и действия.

Впоследствии, в процессе рефакторинга, реструктуризации, вычистки или иного изменения, кода они могут полагаться на то, что созданная страховочная сетка выловит ошибки.

Команды, работающие таким способом, естественно, достигают более высокого уровня покрытия автономными тестами, чем команды, пишущие тесты «в последнюю очередь», и существуют свидетельства в пользу того, что предварительное написание тестов приводит к коду более высокого качества, который проще понять и изменить[3].

Последствия автономного тестирования для безопасности системы

Но даже в командах, помешанных на тестах, существуют пределы полезности автономного тестирования для безопасности. Дело в том, что автономные тесты проверяют то, о чем разработчик знает: в них содержатся утверждения об ожидаемом поведении. Возникновения проблемы с безопасностью, как и внимания со стороны испанской инквизиции, ожидают редко, если вообще ожидают.

Есть некоторые уязвимости, которые можно обнаружить в ходе автономного тестирования – если подходить к нему достаточно серьезно.

Например, уязвимость Heartbleed в OpenSSL и Goto Fail в программном обеспечении Apple можно было бы найти путем тщательного автономного тестирования, как Майк Блэнд (Mike Bland), бывший «наемный тестировщик» (Test Mercenary) в Google, объяснил в статье о культуре тестирования (*https://martinfowler.com/articles/testing-culture.html*), опубликованной в блоге Мартина Фаулера.

[3] Laurie Williams, Gunnar Kudrjavets, Nachiappan Nagappan «On the Effectiveness of Unit Test Automation at Microsoft», 2009 20th International Symposium on Software Reliability Engineering (2009): 81–89 (*https://collaboration.csc.ncsu.edu/laurie/Papers/Unit_testing_cameraReady.pdf*).

Эти получившие широкую огласку уязвимости были вызваны мелкими, допущенными по невнимательности, низкоуровневыми ошибками в коде, имеющем прямое отношение к безопасности. Они ускользнули от внимания разработчиков высочайшего класса, да и от всего сообщества приверженцев открытого исходного кода. В случае Heartbleed проблема возникла из-за того, что не проверялась длина данных, а в случае Goto Fail – из-за копирования и вставки или объединения кода. Блэнд написал серию простых, но тщательно продуманных автономных тестов, которые доказали, что эти ошибки можно было отловить, а заодно сделать код проще и понятнее путем рефакторинга, повышающего его тестопригодность.

К сожалению, как мы увидим, к автономному тестированию редко подходят настолько тщательно – и, уж конечно, не в случае команд Apple или OpenSSL.

Другие уязвимости бывают вызваны фундаментальными просчетами или невежеством. Например, вы вообще забыли реализовать некое средство контроля, потому что не знали, что это надо сделать. Забыли вставить контроль доступа, доверились данным, того не заслуживающим, не параметризовали запрос к базе данных, чтобы предотвратить внедрение SQL.

Такие ошибки вы не найдете с помощью автономного тестирования. Потому что, не зная, что должны что-то сделать, вы не будете писать проверяющий это тест. Вам придется положиться на кого-то, кто знает больше о безопасном кодировании и, возможно, найдет эти ошибки во время инспекции кода (см. предыдущую главу), или надеяться, что они будут обнаружены сканером или в процессе тестирования на проникновение.

Нам не по пути успеха

Большинство тестов (если не все), написанных разработчиками, – позитивные, идущие по «пути успеха» (*https://resources.sei.cmu.edu/library/*), поскольку разработчик хочет доказать себе и пользователю, что код работает. А лучший способ сделать это – написать подтверждающий тест.

Отделы контроля качества (если таковые имеются) в большинстве организаций работают точно так же – тратят большую часть времени на написание и прогон тестов по контрольным спискам, доказывая себе и заказчику, что все работает, как специфицировано.

Таким образом, мы получаем набор тестов, покрывающих в лучшем случае часть успешных сценариев работы с приложением, и почти ничего сверх того.

Беда в том, что противник не желает все время следовать по пути успеха. Он намеренно ищет такие обходные пути в нужные ему части кода, которые разработчики даже не рассматривали.

На самом деле *очень, очень трудно* заставить разработчика мыслить и действовать так же, как атакующий, потому что его этому никто не учил и деньги ему платят не за это. У разработчиков достаточно забот с тем, чтобы понять и наглядно представить, что система должна делать, и реализовать это в коде. В отличие от хакеров, они не тратят – и не могут себе этого позволить – многие часы, обдумывая, как взломать систему, и не ищут мелкие ошибки или несогласованности. Как правило, у разработчика едва хватает времени на написание тестов, доказывающих, что код работает.

И последствия такого положения дел для безопасности велики. Вернемся к статье доктора Дэвида Уилера «How to Prevent the Next Heartbleed», где он рассматривает автономное тестирование с точки зрения безопасности.

> Многие разработчики и организации создают в основном тесты, проверяющие, что должно происходить при правильных входных данных. В общем-то, это имеет смысл: обычный пользователь будет недоволен, если программа не выдает правильный результат, получив правильные входные данные, и большинство пользователей даже не пытаются выяснить, что программа делает, получив неправильные данные. Если ваша единственная цель – быстро выявить проблемы, на которые пользователь мог бы пожаловаться при повседневной работе с программой, то такое, в основном позитивное, тестирование годится.

Даже при разработке через тестирование требуется сначала писать тесты, которые описывают, что должна делать функция, – а не то, чего она делать не должна. В результате разработчики имеют кучу тестов, но почти все они ориентированы на успешное выполнение. И, быть может, есть несколько тестов, проверяющих обработку исключений, да и то в простейших случаях.

Высокого покрытия позитивными тестами недостаточно для полной уверенности. Чтобы удовлетворить требования к безопасности, нам нужны еще и негативные тесты. Снова процитируем Уилера:

> Преимущественно позитивное тестирование бесполезно для построения безопасного программного обеспечения… В комплект регрессионных тестов необходимо включать тесты,

проверяющие некорректные значения каждого поля (в случае числовых полей нужно хотя бы проверять на меньше нуля, равно нулю и больше нуля), каждый переход состояний протокола, каждое специфицированное правило (что произойдет, если это правило не выполнено?) и т. д.

Специалист по безопасности, разработчик или тестировщик, пекущийся о безопасности и желающий быть уверенным в своих тестах, должен обязательно позаботиться об исчерпывающем негативном тестировании библиотек безопасности (например, криптографических функций) и другого важного платформенного кода и высокорисковой функциональности.

А теперь пройдемся по пирамиде тестирования снизу доверху и подумаем, в какие места процесса непрерывной интеграции или непрерывной поставки следовало бы добавить тестирование безопасности.

Тестирование на уровне служб и средства BDD

Помимо низкоуровневых автономных тестов, нам нужен набор осмысленных интеграционных тестов и тестов уровня служб, которые проверяют различные API системы. Это тесты методом черного ящика, которые проверяют, как работают важные функции в реальных условиях.

Обычно тесты такого рода пишутся с использованием каркаса разработки, управляемой поведением (behavior-driven development – BDD). Такая разработка начинается с записи спецификаций – историй, которые понятны заказчику и допускают тестирование, – на высокоуровневом языке, похожем на английский.

Есть два BDD-каркаса, которые были написаны специально для тестирования безопасности и могут запускаться в конвейерах непрерывной интеграции или непрерывной поставки. Они выполняют автоматизированные тесты и каждый раз проверяют, были ли внесены какие-то изменения в код или конфигурацию.

Gauntlt («придирайся к своему коду»)

BDD-каркас тестирования Gauntlt (*http://gauntlt.org/*), основанный на Ruby, позволяет легко писать тесты безопасности для проверки приложения и его конфигурации. В его состав входят адаптеры атак, которые обертывают детали применения инструментов тестирования на проникновение, а также несколько файлов с примерами атак:

- проверка конфигурации SSL с помощью sslyze;
- проверка уязвимости к внедрению SQL с помощью sqlmap;
- проверка конфигурации сети с помощью nmap;
- проведение простых атак на веб-приложения с помощью curl;
- сканирование в поисках типичных уязвимостей с помощью arachni, dirb и garmr;
- проверка конкретных серьезных уязвимостей типа Heartbleed.

Пользователь может расширить или настроить эти атаки либо использовать их как образцы для создания новых атак. В Gauntlt включен также обобщенный адаптер атак, позволяющий выполнить любую командную программу, которая использует для ввода-вывода stdin и stdout. С его помощью можно без особых усилий сконструировать собственные атаки.

BDD-Security

BDD-Security (*https://github.com/continuumsecurity/bdd-security*) – еще один высокоуровневый каркас тестирования безопасности с открытым исходным кодом, написанный на Java. Он включает предопределенный набор тестов для проверки SSL (используется также программа sslyze) и сканирования среды выполнения на наличие инфраструктурных уязвимостей с помощью Nessus.

Но одна из самых полезных возможностей BDD-Security – интеграция с Selenium WebDriver, популярным средством автоматизации функциональных тестов. Включены шаблоны и примеры кода, которыми можно воспользоваться для создания собственных автоматизированных тестов аутентификации и контроля доступа, а также сканирования веб-приложений с помощью OWASP ZAP.

Заглянем под капот

И Gauntlt, и BDD-Security в качестве инструмента автоматизированного тестирования используют программу Cucumber (*https://cucumber.io/*), которая позволяет писать тесты на предметно-ориентированном языке (DSL) Gherkin (сам язык реализован на Ruby), применяя следующий синтаксис:

```
Given {предусловия}

When {выполнить шаги теста}

Then {должны/не должны получиться результаты}
```

Каждый тест возвращает результат «прошел – не прошел», так что тесты легко вставлять в конвейер непрерывной интеграции или поставки.

Теперь посмотрим внимательнее, как устроены атаки Gauntlt.

Файл атаки Gauntlt имеет расширение *.attack*. Каждый файл атаки содержит один или несколько сценариев, состоящих из нескольких шагов Given/When/Then:

```
Feature: Attack/Check description

Background: set up tests for all scenarios

Scenario: specific attack logic in Given/When/Then format

Given "tool" is installed

When I launch a "tool" attack with:

   """

   whatever steps

   """
Then it should pass with/should contain/and should not contain:

   """

   results

   """
```

Команда

```
gauntlt --steps
```

выводит список предопределенных шагов и псевдонимов атак для типичных сценариев, которые вы можете включать в свои атаки:

```
launch a/an "xxxxx" attack with:

the file should (not) contain
```

Разбор результата осуществляется с помощью регулярного выражения, которое определяет, прошел тест или нет.

Gauntlt и BDD-Security упрощают совместную работу безопасников и разработчиков, предоставляя простой общий язык для описания тестов и ряд удобных в использовании инструментов. Если команда уже применяет разработку, управляемую поведением, то эти инструменты станут естественным дополнением.

Другие команды, уже инвестировавшие в разработку собственных инструментов и каркасов тестирования или не проявляющие интереса к подходу BDD и желающие быть ближе к техническим деталям, могут написать собственные скрипты, которые будут делать примерно то же самое. Цель состоит в том, чтобы получить набор автоматизированных тестов, которые проверяют конфигурацию безопасности и поведение работающей системы и выполняются при каждой сборке и каждом развертывании.

Приемочное тестирование

Приемочные тесты обычно прогоняются на уровне пользовательского интерфейса, при этом браузер или мобильный клиент под управлением драйвера выполняют тесты, проверяющие основные функции приложения. Автоматизированные приемочные тесты необходимо писать для относящейся к безопасности функциональности, например аутентификации и управления пользователями и паролями, а также для тех рабочих процессов, которые имеют дело с денежными суммами или конфиденциальными данными. Эти функции должны работать безукоризненно, поэтому нуждаются в высоком уровне покрытия тестами. При этом они редко изменяются, что делает их подходящими кандидатами для автоматизированного приемочного тестирования.

Эту задачу позволяют решить такие инструменты, как Selenium WebDriver (*https://www.seleniumhq.org/*), PhantomJS (*http://phantomjs.org/*) или Sahi (*http://sahipro.com/*). Они умеют программно запускать и управлять браузером или веб-клиентом, осуществлять навигацию, находить элементы UI, выполнять различные действия от имени пользователя, например ввод данных или нажатие на кнопку, а также отвечать на вопросы о состоянии объектов UI или данных.

Функциональное тестирование и сканирование безопасности

Тестировщики проникновения применяют различные инструменты, чтобы понять, а затем и атаковать систему, в том числе:

- прокси для перехвата трафика между браузером или мобильным клиентом и приложением; это позволяет изучить и модифицировать запросы и ответы;
- веб-роботы для обхода всех ссылок на сайте приложения, чтобы составить карту поверхности атаки;

- сканеры приложения на наличие уязвимостей, которые используют собранную информацию для атаки на приложение, путем записи вредоносных значений в каждый параметр или поле запроса.

Инструменты тестирования на проникновение типа Arachni (*http://www.arachni-scanner.com/*) или Burp (*https://portswigger.net/burp*), а также службы сканирования по запросу, предоставляемые компанией WhiteHat или Qualys, успешно находят уязвимости типа внедрения SQL и XSS, равно как и другие виды уязвимостей, не обнаруживаемые средствами статического анализа кода. К ним относятся, в частности, бреши в управлении сеансами типа CSRF (межсайтовая подделка запроса), серьезные ошибки конфигурации и нарушения контроля доступа.

Не нужно быть профессиональным тестировщиком проникновения, чтобы воспользоваться этими инструментами в своей программе тестирования безопасности. Но придется поискать такие инструменты, которые легко поддаются автоматизации, просты в настройке и использовании и быстро передают разработчикам сведения об обнаруженных проблемах в системе безопасности.

Краткое пособие по ZAP

Неплохой отправной точкой может стать OWASP Zed Attack Proxy, или ZAP (*https://www.owasp.org/index.php/OWASP_Zed_Attack_Proxy_Project*), – популярное средство тестирования безопасности с открытым исходным кодом.

Хотя ZAP – достаточно мощное средство, применяемое даже профессиональными тестировщиками проникновения, задумывалось оно как инструмент разработчиков и тестировщиков, не имеющих экспертных знаний в области безопасности. То есть для него характерна простота освоения и эксплуатации – по крайней мере, в качестве инструмента безопасности.

Мы рассмотрим работу с ZAP и его место в циклах быстрого тестирования, поскольку это позволит проиллюстрировать проблемы встраивания сканирования в конвейеры непрерывной интеграции и поставки и поскольку описываемые подходы применимы и к другим инструментам.

Проще всего опробовать ZAP в режиме быстрого запуска – «Quick Start». ZAP запрашивает URL-адрес, обходит ссылки приложения с помощью робота-паука, а затем проводит простые запрограммированные атаки против найденных потенциальных уязвимостей. Атаке

подвергается только открытая поверхность атаки системы, атаковать таким образом функции, требующие аутентификации пользователя, невозможно. В режиме быстрого запуска вы можете почувствовать, на что способен ZAP. Если он найдет серьезные уязвимости на этой ранней стадии тестирования, то устранить их нужно будет немедленно: если ZAP сумел найти их с такой легкостью, то без сомнения это сделает и противник – если уже не сделал.

Если вы хотите глубже разобраться в ZAP и протестировать свое приложение тщательнее, то можете настроить ZAP как перехватывающий прокси и прогнать несколько функциональных тестов приложения: войти в него и пройтись по нескольким ключевым формам и функциям.

ZAP протоколирует все, что вы делаете, и строит модель приложения и его работы. По ходу дела он анализирует ответы приложения и ищет типичные простые ошибки – это называется *пассивным сканированием*. Закончив прогон ручных тестов, вы можете попросить ZAP выполнить активное сканирование только что протестированных страниц. ZAP пытается вставить вредоносные значения в каждое встретившееся ранее поле или параметр и наблюдает, как система реагирует на эти атаки.

Обнаружив проблему, ZAP сообщает о ней, прилагая запрос и ответ в качестве доказательства успешной атаки, объяснение сценария атаки и прочую информацию, которая поможет понять, что произошло и как это исправить.

Последовательность шагов довольно проста.

1. Проверьте, что работаете с последней версией ZAP, это позволит воспользоваться актуальными свежими возможностями и правилами сканирования. ZAP обновляется еженедельно, так что можете либо брать последнюю недельную сборку (и использовать правила в стадии альфа- и бета-тестирования), либо работать с последней официальной стабильной версией.
2. Запустите ZAP.
3. Немного повозитесь с конфигурированием браузера, чтобы настроить ZAP как прокси (руководство пользователя ZAP – вам в помощь).
4. Войдите в приложение и перейдите на страницу, которую хотите протестировать, затем выполните тестовые сценарии для находящейся там формы. Если тестируемая функция поддерживает разные типы пользователей, повторите эти шаги для каждого типа.
5. Ознакомьтесь с результатами пассивного сканирования. ZAP проверяет HTTP-заголовки, куки и возвращенные значения пара-

метров, пытаясь найти типичные ошибки, связанные с безопасностью.
6. Выполните активное сканирование и ознакомьтесь с результатами. ZAP проводит серию запрограммированных атак против встретившихся полей и параметров.
7. Сравните полученные результаты с предыдущими, чтобы выявить новые проблемы и отфильтровать ложноположительные сигналы.
8. Если хотите пойти еще дальше, можете попробовать обойти ссылки в оставшейся части приложения и выполнить активное сканирование и этих страниц тоже. Можете также прочитать в документации о некоторых дополнительных возможностях тестирования на проникновение и случайного сканирования, встроенных в ZAP.

Но все это требует времени: настроить ZAP и браузер, выполнить ручные сценарии, проанализировать результаты и отфильтровать шум. А времени у разработчиков, особенно в гибких командах, как раз и нет.

ZAP в конвейере непрерывной интеграции

Лучше запускать ZAP автоматически в конвейере непрерывной интеграции или поставки.

Начните с простого набора тестов «на дым» с применением базового сканирования ZAP (Baseline Scan – *https://github.com/zaproxy/zaproxy/wiki/ZAP-Baseline-Scan*). Baseline Scan – это Docker-контейнер, в котором уже находится последняя версия ZAP вместе с последними правилами, настроенная на выполнение сокращенного теста вашего приложения в режиме быстрого запуска. По умолчанию робот обходит ваше приложение в течение одной минуты, затем выполняется пассивное сканирование и формируется отчет о результатах. Все это занимает не больше нескольких минут.

Базовое сканирование задумано как проверка состояния, которую можно часто выполнять в конвейере сборки и развертывания (и даже в производственном режиме), чтобы убедиться, что приложение сконфигурировано безопасно – включая HTTP-заголовки и прочие политики безопасности. Именно так эта возможность применяется в Mozilla, где и была разработана.

Проверки безопасности можно включить и в процедуру автоматизированного приемочного тестирования, применяя такой же подход, как при ручном тестировании. Возьмите набор автоматизированных функ-

циональных приемочных тестов, написанных, например, с помощью Selenium WebDriver, и прогоните их в режиме прокси через экземпляр ZAP, работающий без пользовательского интерфейса.

Это даст уверенность в том, что, по крайней мере, в основных пользовательских функциях приложения нет очевидных проблем с безопасностью.

Все эти шаги можно оформить в виде скрипта, который будет автоматически выполняться сервером непрерывной интеграции/развертывания.

Для того чтобы задать инструкции ZAP – сформировать отчет о результатах пассивного сканирования, пройтись по ссылкам приложения и дождаться завершения робота, задать политики сканирования и пороги уведомлений, провести активную атаку – можно использовать Javascript или встроенный в ZAP скриптовый язык ZEST, интерфейс командной строки или REST API.

ZAP умеет представлять результаты в формате HTML, XML или JSON, т. е. можно написать скрипты для анализа результатов, фильтрации ложноположительных и сравнения с предыдущими результатами для выявления новых проблем.

Если, например, вы пользуетесь сервером Jenkins, то можете воспользоваться официальным подключаемым модулем ZAP для Jenkins (*https://wiki.jenkins-ci.org/display/JENKINS/zap+plugin*), который упрощает выполнение команд ZAP прямо из Jenkins и проверку результатов в сборочном конвейере.

Совместное использование BDD-Security и ZAP

Можно также использовать высокоуровневый каркас тестирования типа BDD-Security, который мы уже кратко рассматривали. BDD-Security обертывает выполнение сканирований ZAP и тестов Selenium описанием поведения. Вы записываете истории, касающиеся безопасности в формате Cucumber *Given/When/Then*. Каркас берет на себя детали подготовки и запуска инструментов тестирования, разбора и фильтрации результатов и определения состояния прошел – не прошел.

В примере 11.2 приведена включенная в состав BDD-Security история, описывающая сканирование с помощью ZAP и проверку внедрения SQL.

Пример 11.2. История для BDD-Security

```
  @app_scan
Feature: Automated Application Security Scanning
```

```
Run automated application level tests against the application using OWASP ZAP

Background:
  Given a new scanning session
  And a scanner with all policies disabled
  And all existing alerts are deleted
  And the application is navigated
  And the application is spidered

@cwe-89
Scenario: The application should not contain SQL injection vulnerabilities
  And the SQL-injection policy is enabled
  And the attack strength is set to High
  And the alert threshold is set to Low
  When the scanner is run
  And the following false positives are removed
   |url |parameter |cweId |wascId |
  And the XML report is written to the file build/zap/sql_injection.xml
  Then no medium- or higher-risk vulnerabilities should be present
```

API сканирования

Комбинацию OWASP ZAP и BDD-Security (дополненную собственным кодом для расширения некоторых связующих классов) можно также использовать для сканирования приложений, доступных через REST API. Подход примерно такой же, как при тестировании веб-приложения, когда ZAP используется как перехватывающий прокси, а для изучения функций API применяются другие инструменты, после чего проводится пассивное и активное сканирование раскрываемой поверхности API.

Более полный и точный подход к сканированию API предполагает использование расширений ZAP для SOAP и OpenAPI/Swagger (*https://zaproxy.blogspot.ru/2017/04/exploring-apis-with-zap.html*), которые позволяют импортировать определения API, помогая тем самым ZAP понять и исследовать API.

Еще один вариант – воспользоваться Tinfoil (*https://www.tinfoilsecurity.com/*), коммерческим сканером безопасности приложений, который автоматически строит полную модель вашего REST API, применяя Swagger или другие подобные системы, а затем осуществляет глубокое сканирование и интеллектуальный фаззинг API и схемы аутентификации с целью найти дефекты безопасности и другие проблемы в работающем приложении.

В состав BDD-Security включены истории для различных видов сканирования, а также атак методом внедрения SQL, включения и внедрения на стороне сервера, перенаправления на внешний ресурс, внедрения LDAP, внедрения XPath и обобщенные атаки на шифры, связанные с дополнением (padding oracle attack).

BDD-Security в сочетании с Selenium можно использовать для написания и выполнения приемочных тестов аутентификации, которые проверяют правильность настройки аутентификации и корректность обработки ошибок. Можно также написать тесты контроля доступа, проверяющие, что пользователь может видеть и делать только то, что ему разрешено. Для этого нужно будет модифицировать демонстрационный код на Java, входящий в комплект поставки BDD-Security[4].

Трудности, возникающие при сканировании приложений

Польза, полученная от сканирования, зависит от нескольких факторов:

- качества самого инструмента и правильности его настройки;
- покрытия приложения автоматизированными приемочными тестами и качества робота-паука (если он вызывается для обхода других ссылок);
- выбора правил;
- времени, отведенного на прогон тестов.

Может понадобиться много времени, чтобы обойти ссылки в приложении, пытаясь идентифицировать все страницы и параметры, и еще больше – чтобы провести атаки против точек входа, найденных сканером. Может оказаться практически нецелесообразно и даже невозможно «напустить» на приложение робота и выполнить активное сканирование в сборочном конвейере, если веб-приложение достаточно велико. Автоматизированные инструменты часто сбиваются с пути, зависают, работают дольше отведенного времени и даже «грохаются» в процессе длительного сканирования.

Как мы уже говорили, нацеливание сканирования на ключевые части приложения, которые были исследованы в процессе автоматизированного функционального тестирования, более продуктивно и, наверное, более эффективно, чем применение грубой силы – обход ссылок роботом и их сканирование. Но даже такие тесты могут отнимать много времени.

---

[4] Дополнительные сведения о совместном использовании BDD-Security и ZAP см. в презентации Майкла Брантон-Сполла (Michael Brunton-Spall) «Building securely with agile» (*https://www.youtube.com/watch?v=jkxCLW0x650*).

Для ускорения сканирования можно предпринять несколько очевидных шагов.

1. Сканирование потребляет много процессорного времени и памяти. Чем более мощную машину вы можете выделить для работы сканера и тестируемой системы, тем быстрее пойдет сканирование.
2. Настроить набор правил. Так, по умолчанию ZAP применяет к каждому параметру полный набор правил активного сканирования с умеренной силой. Не всегда это необходимо. Например, если в приложении не используется база данных на основе SQL, то и выполнять атаки внедрением SQL ни к чему.
3. Разбить сканирование на части и выполнять их параллельно. Воспользуйтесь технологией быстрой подготовки (provisioning), чтобы запустить несколько экземпляров приложения и сканировать каждый с разными правилами или нацеливая на разные функции.
4. Выполнять инкрементное сканирование только для новых или изменившихся URL-адресов и не сканировать код, который не изменился или, по крайней мере, не должен был измениться.

Еще одна проблема, которую необходимо решить, – точность. Сканеры, рассматривающие приложение как черный ящик, подобно инструментам статического анализа, часто выдают сотни, а то и тысячи ложноположительных результатов, которые необходимо анализировать и отфильтровывать.

У некоторых сканеров до сих пор имеются проблемы с интерпретацией сложного Javascript-кода или другого динамического контента в пользовательском интерфейсе. Они путаются или не находят проблем, которые должны были бы найти.

Стоит попробовать несколько сканеров и выбрать тот инструмент и подключаемые модули, которые лучше всего отвечают особенностям вашего проекта. Потратьте время, чтобы понять, как работает сканер, и правильно настроить его, и только потом попытайтесь встроить его в сборочный конвейер.

Современные средства тестирования типа ZAP обладают достаточной гибкостью и расширяемостью с помощью API, скриптовых языков и подключаемых модулей. Очень многие уже воспользовались этой гибкостью, чтобы по-разному реализовать автоматизированное сканирование. Это вселяет оптимизм, но и запутывает. Не существует никакой «рекомендованной практики» для сканирования вашего приложения.

Ищите в Google идеи и примеры и ставьте эксперименты, чтобы понять, что вам больше всего подходит.

Сначала наладьте свой конвейер

Настроить сканирование безопасности, чтобы оно корректно работало в автоматизированном конвейере, нелегко. Прежде чем приступать к этому, настройте свой сервер непрерывной интеграции или поставки, отладьте технологические процессы и накопите опыт автоматизации функционального тестирования.

Еще одна заковыристая проблема, которую предстоит решить, – научить инструмент автоматической аутентификации, особенно если собираетесь воспользоваться роботом-пауком или хотите тестировать контроль доступа. При неаутентифицированном сканировании проверяются только общедоступные функции и страницы приложения, которые не представляют особого интереса и ценности для вас – и для противника.

Сканирование – не то же самое, что тестирование на проникновение

Нужно понимать, что описанное выше автоматизированное сканирование – не то же самое, что тестирование на проникновение. Такие инструменты, как ZAP, проводят запрограммированный набор хорошо известных атак в поисках хорошо известных уязвимостей и типичных ошибок. Это всего лишь один из первых – и самых простых – шагов, которые должен предпринять опытный тестировщик проникновения.

Но если автоматизированное сканирование не выявило проблем, то ваша уверенность повышается, а тестировщику проникновения – и противнику – придется поработать усерднее, чтобы найти в системе реальные уязвимости.

Если вы хотите больше узнать о тестировании на проникновение, то рекомендуем начать с «Руководства по тестированию» OWASP (*https://www.owasp.org/index.php/OWASP_Testing_Project*). В нем объясняется, как настроить и провести тестирование на проникновение. Приведены контрольные списки и методы разведки, построения карты среды и приложения, получения цифровых отпечатков технологического стека. Также предложены тесты для управления удостоверениями, функций аутентификации, управления сеансами, авторизации, различных видов атак внедрением и сведения о том, как искать бреши в бизнес-логике.

У большинства инструментов имеются различные режимы аутентификации, помогающие решить эту проблему. К примеру, ZAP реализует типичные методы аутентификации и с помощью скриптов поддерживает сложные и нестандартные схемы аутентификации и повторной аутентификации. ZAP также позволяет выполнить шаги аутентификации вручную и сохранить полученную информацию о сеансе в репозитории кода, чтобы впоследствии использовать для автоматизации сканирования.

Тестирование инфраструктуры

Новые инструменты типа Docker, Vagrant, Packer и Ansible ускоряют и упрощают создание пакета и подготовку среды для разработки и тестирования. Вместо того чтобы днями и неделями ждать, пока системный администратор настроит систему, разработчик может просто взять созданные сообществом сценарии для Chef или Ansible либо общедоступные образы Docker и за несколько минут развернуть временный экземпляр в частном или публичном облаке – с полной исполняющей средой и всеми необходимыми инструментами.

Преимущества, которые дает гибкой команде возможность самостоятельно подготавливать среду разработки и тестирования, очевидны. Команда полностью контролирует конфигурацию среды и время ее создания. Ей не нужно несколько дней или недель дожидаться ответа от группы эксплуатации. Она может легко и без особых затрат экспериментировать с новыми инструментами и платформами. И навсегда забыть о проблеме «на моей машине работает», поскольку может гарантировать, что среды разработки и тестирования сконфигурированы одинаково.

Но вместе с тем появляются новые риски.

- Как и любой открытый исходный код, образы Docker или другие конфигурационные рецепты или шаблоны, особенно скачанные из репозиториев сообщества, могут содержать ошибки, устаревшие пакеты и прочие уязвимости, которые разработчики легко могут не заметить.
- Для экономии времени – и поскольку не понимают технических деталей или не обращают на них внимания – разработчики часто оставляют конфигурации по умолчанию, а это почти всегда небезопасно.
- Если разработчики самостоятельно готовят для себя инфраструктуру, то группы безопасности и эксплуатации сознательно исклю-

чаются из процесса, а следовательно, изменения производятся без их ведома и надзора.
- Эти небезопасные конфигурации потенциально могут стать мишенью для противника и, что еще хуже, просочиться в производственную систему.

Во всех этих инструментах конфигурация – Docker-файлы, сценарии Ansible, рецепты Chef, облачные шаблоны – задается в виде кода. А это означает, что конфигурацию инфраструктуры можно автоматически сканировать и тестировать, как любой другой код.

Вы можете просканировать конфигурационный код или образы на предмет наличия типичных ошибок и уязвимостей, а также рисков несоответствия нормативным требованиям. Существует, к примеру, целый ряд инструментов для статического сканирования образов и контейнеров Docker, в том числе:

- Docker Bench for Security (*https://github.com/docker/docker-bench-security/*) – сканирует контейнеры Docker на соответствие стандартам Центра интернет-безопасности для Docker (CIS Benchmark for Docker);
- Docker Security Scanning (*https://docs.docker.com/datacenter/dtr/2.4/guides/ admin/configure/set-up-vulnerability-scans/*) – дополнительная служба, разработанная Docker для сканирования образов в закрытых репозиториях на предмет наличия известных уязвимостей;
- Clair (*https://github.com/coreos/clair/*) – сканер уязвимостей с открытым исходным кодом для образов Docker из CoreOS.

Вы также можете написать собственные тесты для следующих целей:

- реализовать методику разработки через тестирование для инфраструктуры: сначала определить конечное состояние конфигурации в виде тестов, а затем написать конфигурационный код;
- выявить ошибку кодирования на ранней стадии: проверка синтаксиса и автономное тестирование важны для конфигурационного кода с динамическим содержимым, потому что в противном случае многие ошибки невозможно обнаружить до начала выполнения;
- обнаруживать регрессию в результате переработки кода (вычистки и реструктуризации) инженерами, перехода на новые версии программ или миграции на новые платформы;
- проверять состояние системных конфигураций, поддерживать соответствие между системами и средами, вылавливать «снежинки» (одноразовые конфигурации);

- в качестве документации для конфигурации;
- навязывать выполнение политик и наставлений: политик эксплуатации, соответствия нормативным требованиям, безопасности, наставлений по укреплению.

Как и в случае пирамиды тестирования приложений, существуют различные инструменты и подходы для разных уровней кода тестирования инфраструктуры.

Проверка правил оформления

Инструменты из этой категории (linting) проверяют синтаксис, соблюдение правил форматирования и наставлений по кодированию. Они помогают удостовериться, что код будет работать правильно, и особенно важны для динамических скриптовых языков:

- Puppet parser с флагом validate, Puppet-lint;
- Foodcritic, Rubocop для Chef;
- Ansible-lint.

С точки зрения безопасности, ценность такой проверки кода минимальна.

Автономное тестирование

Используя в автономных тестах заглушки или mock-объекты вместо реальной исполняющей среды, вы можете проверить правильность логики на холостых прогонах. В результате вносить изменения в код становится безопаснее.

- RSpec-Puppet;
- Chefspec.

Как и средства проверки синтаксиса, автономные тесты вряд ли позволят обнаружить серьезные ошибки, связанные с безопасностью.

Приемочное тестирование

Приемочные тесты – лучшее место для проверок безопасности и соответствия нормативным требованиям. Разверните чистую тестовую среду, выполните код рецепта или сценария и сравните фактический результат с ожидаемым. Как и в случае приемочных тестов приложения, эти тесты обходятся дороже, работают медленнее, но дают более полные результаты:

- Bats – Bash Automated Testing System;
- Beaker для Puppet;
- Goss;
- Serverspec;
- InSpec – тесты, которые можно явно привязать к нормативным требованиям;
- RSpec – создайте собственный каркас тестирования вместо использования предметно-ориентированного языка, встроенного в Serverspec.

Рассмотрим несколько популярных инструментов, применимых для тестирования различных спецификаций или языков задания конфигурации на различных платформах. Это особенно полезно на крупных предприятиях, где команды могут применять много разных подходов к управлению конфигурацией, поскольку для всех них есть стандартный каркас тестирования. Кроме того, это облегчает переход от одного метода управления конфигурацией на другой.

Test Kitchen

Test Kitchen (*https://kitchen.ci/*) – расширяемый каркас управления тестированием, который можно использовать для подготовки и прогона тестов инфраструктурного кода. Как следует из слова «kitchen» (кухня) в названии, он был разработан для тестирования рецептов Chef, но с помощью подключаемых модулей может быть распространен на тестирование кода для Puppet, Ansible, SaltStack и других инструментов управления конфигурацией.

Test Kitchen дает возможность писать тесты с использованием различных каркасов тестирования, в т. ч. Serverspec, RSpec, Bats и Cucumber. Он также позволяет прогонять тесты на платформе Linux или Windows, поддерживает Vagrant, Docker и большинство облачных платформ.

Serverspec

Serverspec (*https://serverspec.org/*) – расширение каркаса тестирования RSpec, написанного на Ruby, ориентированное специально на поддержку тестирования инфраструктуры. С помощью Serverspec можно писать высокоуровневые приемочные тесты, проверяющие, что состояние конфигурации системы (файлы, пользователи, пакеты, службы, порты) совпадает с ожидаемым. Эти тесты можно прогонять до или после внесения изменений.

Поскольку Serverspec проверяет конечное состояние, не имеет значения, какие инструменты использовались для внесения изменений в конфигурацию. ServerSpec подключается к каждой системе по SSH, выполняет ряд пассивных проверок конфигурации и возвращает результаты для сравнения. Поэтому проверку безопасно запускать для тестирования, а также для аудита производственной системы на соответствие нормативным требованиям. Вот простой пример проверки с помощью Serverspec:

```
describe package('httpd'), :if => os[:family] == 'redhat' do
  it { should be_installed }
end
```

Существует вариант Serverspec под названием AWSpec (*https://github.com/k1LoW/awspec*), предназначенный для выполнения аналогичных тестов в облаке Amazon AWS.

В главе 13 мы увидим, как инструменты типа Serverspec можно использовать для написания тестов безопасности и проверки работающей системы.

Создание автоматизированного конвейера сборки и тестирования

Скорость, с которой команды, практикующие гибкие методики и DevOps, поставляют ПО, продолжает возрастать. Scrum-команды, которые раньше применяли месячные спринты, теперь все чаще поставляют новый код каждую неделю. DevOps-команды, практикующие непрерывное развертывание, могут вносить изменения в производственную систему по нескольку раз на дню. В качестве крайнего примера приведем компанию Amazon, в которой тысячи разработчиков, сгруппированных в небольшие команды, «на две пиццы», каждый день непрерывно и автоматически вносят в свои системы тысячи изменений.

Для работы с такой скоростью необходимы новые подходы к проектированию систем, сборке и развертыванию кода, оптимизированные под быстрые инкрементные изменения. Но все это по-прежнему покоится на идеях и фундаментальных принципах гибкой и бережливой разработки.

Рассмотрим отдельные шаги в направлении быстрого и непрерывного развертывания и поговорим о том, что это означает с точки зрения тестирования и безопасности.

Ночная сборка

Чтобы двигаться вперед быстрее, команда должна собирать систему часто: чем больше интервал между объединениями и сборками кода, тем больше риск конфликтов и недопонимания.

Еще в 1990-х годах Microsoft ввела в обиход практику «ночной сборки». Разработчикам рекомендовалось записывать изменения в репозиторий каждый день, а каждую ночь запускалась задача, которая собирала программу, поэтому на следующее утро изменения оказывались доступны всем членам команды. Со временем команды стали добавлять тесты, запускаемые после сборки, которые вылавливали типичные ошибки. Всякий, кто «портил сборку» записью некорректного кода, должен был купить пиццу или понести другое наказание, в том числе присматривать за сборкой, пока ее не испортит кто-нибудь еще[5].

Чтобы система, особенно крупная и унаследованная, собиралась без ошибок, необходимо приложить немало усилий. Способность делать это ежедневно – большой шаг вперед. Регулярная сборка работоспособной системы сплачивает команду: состояние разработки открыто для всех, и возрастает уверенность в собственной способности поставить продукт. И у каждого члена команды появляется шанс исправить ошибки и неправильные предположения на ранней стадии.

Непрерывная интеграция

Следующий шаг к повышению открытости и скорости – непрерывная интеграция.

Разработчики учатся записывать код в репозиторий небольшими порциями, часто несколько раз в день, чтобы все члены команды видели и могли использовать изменения, внесенные другими. При каждой записи в репозиторий запускается сервер непрерывной интеграции (например, Jenkins или Travis) и производятся автоматическая сборка и проверка кода, чтобы никакие изменения не привели к порче сборки.

Это делается путем выполнения комплекта автоматизированных тестов, в основном автономных. Тесты должны выполняться быстро: чем меньше времени на это уходит, тем чаще разработчики будут записывать изменения.

Ниже перечислены типичные шаги процесса непрерывной интеграции:

---

[5] Joel Spolsky «The Joel Test: 12 Steps to Better Code», Joel on Software, August 9, 2000 (*https://www.joelonsoftware.com/2000/08/09/the-joel-test-12-steps-to-better-code/*).

- проверка кода, выполняемая «без отрыва от производства», в IDE разработчика;
- ручная инспекция кода до записи в основную ветвь репозитория или объединения с кодом других членов команды;
- компиляция и анализ предупреждений компилятора;
- проверки во время сборки;
- инкрементный статический анализ, выполняемый сервером непрерывной интеграции;
- автономное тестирование;
- сохранение артефактов сборки в репозитории артефактов, например: Apache Archiva, Artifactory или Sonatype Nexus.

Этих шагов должно быть достаточно для уверенности в корректности изменений и самой сборки. После этого нужно проверить, что система готова к выпуску релиза.

Непрерывная поставка и непрерывное развертывание

Сам смысл DevOps – в применении практики и инструментов непрерывной интеграции для реализации конвейеров непрерывной поставки или непрерывного развертывания.

В случае непрерывной поставки изменения поступают с этапа разработки на этап тестирования, а затем в производственную систему в виде последовательности автоматизированных шагов с применением набора инструментов, обеспечивающих сквозной контроль. Можно доказуемо утверждать, что в любой момент времени имеется система, готовая к эксплуатации.

В случае непрерывного развертывания все это доводится до стадии эксплуатации: разработчики записывают в репозиторий изменение или исправление, и если оно проходит все автоматизированные шаги и проверки в конвейере, то сразу же развертывается в производственной системе. Так работают компании Netflix, Amazon и Etsy.

Любое изменение кода или конфигурации автоматически запускает процесс непрерывной интеграции, сборки системы и прогона набора быстрых автоматизированных тестов. Если сборка прошла успешно, то ее артефакты автоматически упаковываются, а затем развертываются в тестовой среде для интеграционного и приемочного тестирования.

Если все тесты проходят, то код передается в промежуточную технологическую систему для репетиции развертывания в производственной среде и комплексного тестирования.

- Выполнить все предыдущие шаги непрерывной интеграции.
- Взять последнюю удачную сборку из репозитория артефактов и упаковать ее.
- Подготовить тестовую среду (с помощью какого-нибудь автоматизированного инструмента типа Chef, Puppet, Vagrant, Docker или CloudFormation).
- Выполнить развертывание в тестовой среде (репетиция шагов развертывания).
- Запустить приложение (проверить наличие ошибок).
- Выполнить тесты «на дым» и проверить состояние среды.
- Прогнать приемочные тесты.
- Прогнать интеграционные тесты.

Если все эти шаги завершаются успешно, то:

- подготовить промежуточную технологическую среду;
- выполнить развертывание в промежуточной среде (репетиция шагов развертывания и выпуска в среде, аналогичной производственной);
- повторить запуск приложения, тесты «на дым», приемочные и интеграционные тесты;
- выполнить нагрузочные и эксплуатационные тесты.

Успешное завершение этих действий доказывает, что изменение корректно и все шаги его развертывания работают без ошибок.

Экстренное тестирование и инспекция

Некоторые тесты и инспекции должны выполняться вне конвейера непрерывной поставки, потому что занимают слишком много времени или требуют ручной передачи из рук в руки (или то и другое сразу). Результаты таких тестов и проверок можно записать в журнал пожеланий команды. Приведем примеры экстренных тестов и инспекций:

- ручное исследовательское тестирование и тестирование удобства пользования (usability);
- глубокое сканирование (статический анализ кода или сканирование приложения) крупных систем должно выполняться отдельно от общего процесса, а его результаты анализируются и рассортировываются вручную;
- тестирование на проникновение;
- возможно, фаззинг API или файлов.

Фаззинг в конвейерах непрерывной интеграции и поставки

Фаззинг (fuzz testing, или fuzzing) – это автоматизированное тестирование надежности методом грубой силы. Здесь негативное тестирование доводится до крайности – автоматически генерируются полуслучайные данные для тестирования функций API или импорта файлов. Цель состоит в том, чтобы выявить ошибки контроля входных данных и посмотреть, что произойдет, если некорректные данные все-таки просочатся (переполнение буфера, целочисленное переполнение, затирание памяти и т. д.).

Фаззинг обычно применяется для тестирования встраиваемого ПО и сетевых протоколов. Эта техника также стала важной частью программ безопасности приложения в компаниях Microsoft, Adobe и Google (особенно в команде, работающей над Chrome). Исследователи безопасности часто применяют фаззинг для поиска уязвимостей, и фаззеры используются в вышеупомянутых инструментах сканирования приложений на уязвимость.

При включении фаззинга в автоматизированный сборочный конвейер необходимо будет ограничить время выполнения тестов и придумать, как автоматически вернуть результаты. Инструменты фаззинга, например Peach (*https://www.peach.tech/*), предоставляют возможность ограничить время прогона и с помощью подключаемых модулей встраиваются в популярные серверы непрерывной интеграции, например Jenkins. Как и ZAP, Peach можно установить в виде прокси между вашими тестами и тестируемым API – тогда одни и те же тесты будут многократно прогоняться с различными случайными данными.

Но у фаззинга есть ряд недостатков. Чтобы обрести уверенность, нужно прогнать очень много тестов, что может занять несколько часов или даже дней. Тестируемая система или служба может «упасть», и тогда ее нужно будет запустить заново и сбросить в начальное состояние. Фаззинг не всегда дает понятную обратную связь, особенно в случае краха системы – кто-то должен вручную проанализировать тесты, трассировку стека и сообщения об ошибках, чтобы понять, что, где и почему случилось.

Поэтому в большинстве организаций фаззинг вряд ли стоит считать хорошей отправной точкой для автоматизации тестирования.

Передача в эксплуатацию

После того как сборка, тестирование и развертывание успешно завершены, последние изменения можно считать готовыми к передаче в эксплуатацию. При непрерывном развертывании этот шаг выполняется автоматически, а при непрерывной поставке – после ручного одобрения или других проверок.

В любом случае, понадобятся те же артефакты, инструменты и шаги, которые уже использовались на предыдущих этапах для развертывания производственной системы, поскольку они уже отрепетированы, отлажены и многократно доказали свою работоспособность. Тем самым большинство рисков и неудобств развертывания снимается.

Рекомендации по созданию успешного автоматизированного конвейера

Каждая последующая стадия конвейера зависит от предыдущих – от простых тестов и проверок в простой среде постепенно производится переход к тестам и проверкам в среде, более приближенной к реальной.

Мы собираем и упаковываем версию-кандидат один раз на ранней стадии конвейера, сохраняем артефакты сборки в безопасном месте и передаем эту версию на последующие шаги, пока не останется никаких возражений против ее выпуска.

Даже если вы еще не готовы к внедрению непрерывной интеграции или поставки, все равно программу тестирования следует строить так, чтобы тесты можно было запускать автоматически. Это означает, что тесты должны быть повторяемыми и детерминированными: они должны давать один и тот же результат при каждом прогоне. Кроме того, они должны давать недвусмысленный индикатор успешного или неудачного завершения, который можно проверить во время выполнения.

Для экономии времени постройте последовательность тестов, так чтобы раньше выполнялись самые важные тесты – те, что проверяют существенные функции, и те, что часто «ломаются». Если имеются «капризные» тесты, в которых время от времени возникают проблемы из-за хронометража или еще какие-то, то изолируйте их, чтобы они не портили сборку.

По мере увеличения количества тестов и шагов сканирования имеет смысл разделить их на параллельные потоки и запускать одновременно. Для этого можно воспользоваться публичными или частными облаками – создать в них тестовые фермы и решетки сканирования, чтобы тесты и проверки, занимающие несколько часов, выполнялись за считанные минуты.

Место тестирования безопасности в конвейере

Мы уже видели, как можно автоматизировать некоторые аспекты тестирования безопасности. Автоматизированные тесты можно поместить на разные стадии конвейера непрерывной интеграции или поставки.

Тесты и проверки, выполняемые до непрерывной интеграции:

- статическая проверка кода в IDE до записи в репозиторий с использованием встроенных средств проверки кода или подключаемых модулей;
- сканирование в поисках секретных данных перед объединением кода;
- инспекции кода.

Тесты и проверки безопасности в ходе непрерывной интеграции:

- проверки на этапе сборки, включая обнаружение новых компонентов и компонентов с известными уязвимостями (мы уже рассматривали этот вопрос в главе 6);
- автономное тестирование, особенно негативные тесты ключевых функций;
- инкрементный статический анализ, соблюдение правил оформления и проверки использования запрещенных функций и других опасных практик;
- проверки безопасности «на дым»: быстрые и простые проверки, оформленные в виде сценариев, например для Gauntlt.

Тесты и проверки безопасности в ходе непрерывной поставки:

- целенаправленное сканирование приложения (с помощью ZAP, Arachni или другого инструмента класса DAST);
- автоматизированные атаки с применением Gauntlt;
- автоматизированные приемочные тесты функций безопасности (аутентификация, контроль доступа, управление удостоверениями, аудит и криптозащита) с применением BDD-Security и (или) Selenium WebDriver.

Если команда не опирается на автоматизированный сборочный конвейер в ходе развертывания изменения или если она несерьезно относится к ошибкам тестирования в конвейере, то добавление проверок безопасности не принесет значимого эффекта.

Перед тем как добавлять тестирование безопасности в конвейер, убедитесь, что конвейер настроен правильно и что команда использует его корректно и последовательно.

- Все изменения записываются в репозиторий кода.
- Члены команды часто производят запись.
- Автоматизированные тесты выполняются быстро и одинаково.
- Если тест завершается неудачно, то команда прекращает работу и немедленно исправляет ошибки до внесения новых изменений.

Приступая к автоматизации сканирований и тестов безопасности, для начала запускайте их в отдельном потоке конвейера, чтобы ошибки тестов не приводили к немедленной остановке сборки. Проанализируйте результаты тестирования, чтобы принять решение о пороге отказов, затем настройте инструменты, так чтобы они не слишком тормозили сборку и чтобы команда не тратила впустую время, расследуя ложноположительные предупреждения. Когда все это заработает, можете настроить конвейер так, чтобы ошибка при проверке на безопасность приводила к его остановке.

Место ручного тестирования в гибких методиках

Хотя в гибких методиках и особенно в DevOps автоматизированное тестирование играет главную роль из-за необходимости быстрой обратной связи и реагирования на ошибки, у ручного тестирования (и тестировщиков) тоже есть важное место, особенно когда речь заходит о безопасности.

Ручное регрессионное тестирование не масштабируется на высокоскоростные гибкие среды и является пустой тратой времени. А при гибкой разработке от растранжиривания времени следует избавляться любой ценой. Однако бывают случаи, когда ручное тестирование оказывается полезным. Например, тестировать удобство пользования должны реальные люди: тестировщики или разработчики, оценивающие собственный продукт, или настоящие пользователи, работающие с бета-версией.

Еще один ценный вид ручного тестирования – *исследовательское тестирование*. Его цель – выяснить, как работает приложение, и найти его слабые места. Тестировщики-исследователи пытаются довести систему до предела возможностей, чтобы найти дефекты раньше пользователей – и противников.

Исследовательские тесты обычно не оформляются в виде скриптов. Тестировщик может начать с проторенной приемочным тестом дорожки, а затем уйти в сторону, где, как ему кажется, может встретиться что-то более интересное или важное.

Он может попытаться выполнять действия в другом порядке, нажимать кнопки, которые нажимать не следует, хулиганить со ссылками и данными, переходить назад и вперед – и смотреть, что при этом получается. Тестировщик-исследователь импровизирует на ходу, стремясь лучше понять, как работает система. Он записывает все, что делал, что увидел и что работало неправильно, а затем составляет отчет о найденных дефектах.

Как сломать программу

Чтобы глубже разобраться в исследовательском тестировании, стоит почитать пионерскую работу Джеймса Баха «Exploratory Testing Explained» (*http://www.satisfice.com/articles/et-article.pdf*).

В книгах Джеймса Уиттекера (James Whittaker) «How to Break Software» (Pearson), «How to Break Web Software» (Addison-Wesley Professional) и «How to Break Software Security» (Addison Wesley) объясняется враждебный подход к тестированию приложений.

Хотя некоторые из упоминаемых инструментов, возможно, уже устарели, сам подход и модели атак, описываемые Уиттекером, все еще актуальны. Например, атаки методом произвольного перехода по URL-адресам веб-приложения – простой способ протестировать средства контроля в основных рабочих процессах. Попробуйте перейти сразу на конкретный шаг или пропустить несколько шагов, указав прямой URL-адрес страницы. Попробуйте пойти назад, пропустить шаги утверждения. Если приложение не возразило, значит, вы нашли серьезную ошибку, которой противник или мошенник не преминет воспользоваться.

Тестирование на проникновение – крайняя и высокоспециализированная форма исследовательского тестирования, когда тестировщик примеривает на себя роль атакующего. Тестировщики проникновения применяют перехватывающие прокси, сканеры и другие инструменты, чтобы выявить уязвимости, а затем и эксплуатировать их. Чтобы делать это эффективно, нужны специальные технические знания и опыт.

В большинстве команд нет штатных тестировщиков проникновения, но они могут многого добиться с помощью враждебного исследовательского тестирования. Это можно проделывать неформально, когда разработчики объединяются в пары с тестировщиками или друг с другом,

чтобы объяснить некоторую функцию и изучить проблемы, коль скоро таковые обнаружатся.

Исследовательское тестирование следует применять к таким функциям безопасности, как вход в систему и восстановление забытого пароля, а также к особо важным рабочим процессам, например: покупки в интернет-магазине, онлайновый банкинг или обработка платежей.

В кратком пособии по ZAP мы уже видели, что ручное исследовательское тестирование – полезный и необходимый первый шаг на пути к автоматизации тестирования. Он нужен, чтобы лучше понять приложение и инструменты, а также определить, какие сценарии автоматизировать в первую очередь.

Враждебное исследовательское тестирование, когда разработчиков и тестировщиков просят думать нестандартно или пытаться взломать собственную систему, дает информацию, которую невозможно получить при структурированном, повторяющемся, автоматизированном тестировании. Таким способом можно найти серьезные ошибки в части удобства пользования, надежности и безопасности. Но еще важнее – что можно узнать, где слабые места в системе и в рабочих процессах. Обнаружив серьезные дефекты в части безопасности и надежности, вы должны внимательно изучить код и проект и в ходе последующих инспекций и тестирования понять, что вы могли упустить и что еще, возможно, следует исправить.

Такое тестирование дорого обходится и не масштабируется. Оно требует много времени и сильно зависит от индивидуальных навыков, опыта и интуиции тестировщиков. Применяйте его в той мере, в какой возможно, но помните, что полагаться на ручное тестирование в стремлении сделать систему безопасной нельзя.

Как добиться, чтобы тестирование безопасности работало в гибких методиках и DevOps?

Чтобы тестирование безопасности успешно работало, нужно осознавать пределы и ограничения гибкой разработки и непрерывной поставки.

Для исчерпывающего тестирования и аудита на основе контрольно-пропускных пунктов не хватает времени. Необходимо разбить тестирование на небольшие шаги, допускающие автоматизацию, чтобы их можно было выполнять при каждом изменении. Акцентируйте внима-

ние на тестах, которые смогут выловить типичные и важные ошибки на ранних стадиях.

Правильно автоматизировать тестирование нелегко. Нужно понимать предметную область, проект системы, платформу среды выполнения, а также порядок сборки и конфигурирования – только тогда систему можно будет представить для тестирования безопасности. Вы должны владеть инструментами тестирования и методами непрерывной интеграции, уметь писать тесты, выработать стратегию реагирования на ошибки тестов и научиться отсекать ложноположительные сигналы. И все это необходимо делать быстро, чтобы не отставать от других членов команды, которые непрерывно вносят новые изменения.

Чтобы все это сошлось, необходимо сочетание обширных технических знаний с навыками в области тестирования и безопасности, а это значит, что нужно объединить усилия разработчиков, тестировщиков и безопасников:

- группа безопасности помогает разработчикам писать критерии приемки и тестовые сценарии для историй, касающихся безопасности;
- группа безопасности может инспектировать автономные тесты для высокорискового кода и автоматизированные тесты для инфраструктурных сценариев;
- попробуйте подойти к обеспечению безопасности через тестирование: пусть безопасники напишут для команды тесты раньше, чем команда приступит к написанию кода, и пусть эти тесты станут частью целей, контролируемых во время приемки;
- безопасники могут помочь команде внедрить в сборочный конвейер сканирование приложения как черного ящика и сканирование инфраструктуры;
- безопасники могут участвовать в исследовательском тестировании, особенно функций безопасности, и продемонстрировать команде, как превратить сеанс исследовательского тестирования в тест безопасности, включив в тест атаку с помощью перехватывающего прокси типа ZAP.

Ищите способы упростить тесты и сделать их общим достоянием. Воспользуйтесь шаблонами, которые предоставляют инструменты типа Gauntlt и BDD-Security, чтобы создать стандартные тесты, общие для нескольких команд и систем.

Ищите способы привлечь разработчиков к написанию тестов безопасности и владению их результатами. Перейдите на режим самообслу-

живания в тестировании безопасности, сделав его настолько простым, чтобы разработчики могли создавать и запускать тесты самостоятельно. Никаких двусмысленностей, никаких предупреждений, для интерпретации которых нужен крутой специалист.

Анализируйте, что получилось, и улучшайте. Заполняйте пробелы в тестировании по мере необходимости: когда обнаруживаются уязвимости, когда команда меняет направление работы, перерабатывает проект или модернизирует технологию, а также при изменении ландшафта угроз.

Сухой остаток

В гибких командах разработчики отвечают за тестирование собственной работы и применяют автоматизацию тестирования и непрерывную интеграцию, чтобы быстро выявлять проблемы.

- Тестирование силами разработчиков чаще всего идет по «успешному пути», призванному продемонстрировать, что функция работает. Однако противник сходит с этого пути, стремясь найти в проекте и коде граничные случаи и слабые места, допускающие эксплуатацию. Разработчики должны понимать важность негативного тестирования, особенно для библиотек, имеющих отношение к безопасности, и другого высокорискового кода.
- Динамическое сканирование методом черного ящика с помощью инструментов типа OWASP ZAP можно включить в конвейер непрерывной поставки для отлавливания типичных уязвимостей в мобильных и веб-приложениях. Но это нелегко.
- Начинайте с малого. Экспериментируйте. Изучите инструменты вдоль и поперек и только потом предлагайте их команде разработчиков.
- Группы безопасности могут принять на вооружение разработку через тестирование (тесты пишутся раньше, чем код), чтобы приучить команду думать о безопасности в процессе тестирования приложения и инфраструктуры.
- В автоматизированных каркасах тестирования безопасности, например Gauntlt и BDD-Security, имеются шаблоны, с помощью которых можно создать тесты безопасности «на дым», а затем распространить их на разные команды и разные системы.
- Тестируйте не только приложение, но и инфраструктуру. Для проверки правильности и безопасности конфигурации инфраструктуры применимы такие инструменты, как Serverspec и InSpec.

- Автоматизированные тесты, включая сканирование, должны работать быстро и давать разработчикам ясное указание «прошел – не прошел».
- Ручные приемочные тесты не масштабируются на гибкие среды. Но ручное исследовательское тестирование и тестирование на проникновение могут выявить важные проблемы, пропущенные автоматизированными тестами, и предоставить команде ценную информацию о слабых местах и рисках в системе и процессах разработки.

Тестирование безопасности должно входить составной частью в «определение готовности»: контракт, заключенный командой с самой собой и с организацией, на поставку работающего ПО в каждом релизе. Команда должна ясно понимать свои обязательства по соблюдению нормативных требований и безопасности и согласовать виды тестирования, необходимые для выполнения этих обязательств. Какие типы тестов безопасности и сканирования необходимы выполнять, и как часто? Какие обнаруженные результаты приводят к аварийному завершению сборки? Какой уровень покрытия автоматизированными тестами необходим для библиотек, относящихся к безопасности, и для иного высокорискового кода?

Глава 12

Внешние инспекции, тестирование и рекомендации

В мире наблюдается нехватка знаний по информационной безопасности вообще и по безопасности приложений в особенности. Это означает, что для подготовки и неукоснительного выполнения программы по безопасности, возможно, придется обратиться за помощью к сторонним организациям.

Тестирование на проникновение, вознаграждение за обнаружение ошибок, оценка уязвимостей и прочие внешние инспекции могут открыть вашей организации доступ к широкому сообществу с его знаниями, творческой энергией и накопленным опытом.

По мере накопления собственного опыта в обеспечении безопасности потребность в услугах внешних консультантов может снижаться, но не следует думать, что когда-то она полностью отпадет. Даже если внутри организации имеются сильные технические специалисты по безопасности, все равно для страховки – и чтобы не выдавать желаемое за действительное – привлечение чужого опыта не помешает.

В состав многих нормативно-правовых требований, распространяющихся на вашу эксплуатационную среду, может входить требование о внешнем тестировании или аудите безопасности с целью независимого подтверждения того, что предприняты все меры надлежащей осмотрительности для защиты систем, клиентов и данных.

Например, в стандарте PCI DSS оговорено, что системы и приложения, составляющие среду, на которую распространяется действие стандарта, должны инспектироваться внешними тестировщиками (в PCI они называются квалифицированными оценщиками безопасности, QSA) как ежегодно, так и всякий раз после внесения в среду существенного изменения. Эти тестировщики должны в своей работе следовать отраслевым

стандартам и методологиям и представить подробный отчет о результатах проверки. Инспектируемая сторона должна предпринять действия по устранению всех обнаруженных существенных недостатков.

В отрасли безопасности существует целый сектор, занятый исключительно предоставлением услуг по оценке соответствия нормативным требованиям и консультированию организаций в части того, как ориентироваться в различных аспектах этих требований. Это пример сторонней поддержки безопасности, которая не просто желательна, но обязательна.

Организация внешней инспекции иногда оказывается пугающим процессом. А извлечь из этих инспекций что-то полезное бывает еще труднее. В этой главе мы постараемся объяснить вам, как строить отношения со сторонними специалистами по безопасности, и, что, наверное, важнее, как получить максимальную отдачу от потраченных на них денег.

Заявление об отказе от ответственности

Все мнения и рекомендации, приведенные в этом разделе, являются ориентировочными и не должны рассматриваться как реклама какого-то определенного подхода, компании или методики обеспечения внешней гарантии.

Проще говоря, мы не пытаемся вам что-то продать или посоветовать, как потратить деньги на услуги консультантов. Но наш опыт показывает, что помощь экспертов со стороны может быть действительно полезной, и мы полагаем, что внешние инспекции играют важную роль в доведении программ безопасности приложений до состояния зрелости.

Почему нужны внешние инспекции?

Внешние инспекции безопасности преследуют ряд целей, помимо проверки соответствия нормативным требованиям.

Независимость

Внешние инспекторы не заинтересованы в том, чтобы представить команду или организацию в лучшем виде. На самом деле у них прямо противоположные цели: они хотят найти серьезные проблемы, чтобы вы потом снова пригласили их и попросили помочь в устранении этих проблем. С точки зрения тестировщи-

ка проникновения, худший исход – когда ничего серьезного не найдено, поэтому он будет лезть из кожи вон, чтобы избежать такой ситуации!

Опыт

Нанимать профессиональных тестировщиков и аудиторов безопасности не только чревато испытаниями, но и дорого. В мире наблюдается нехватка таких специалистов, при этом для многих тестов требуются узкоспециальные знания о конкретной технологии или контексте. Держать таких специалистов в штате может оказаться не по карману организации, поэтому поиск их на стороне может иметь практический и финансовый смысл.

Поддержка со стороны руководства и передача на более высокий уровень

Серьезные затраты на внешнюю инспекцию могут оказаться весомым аргументом, который заставит руководство и исполнительные органы обратить внимание на риски, стоящие перед организацией. Проблемам, выявленным сторонними инспекторами, часто придают больший вес, чем тем, что найдены в ходе внутренних инспекций, и их проще передать в вышестоящую инстанцию. Руководство вашей организации обязано управлять командой, так чтобы обеспечить безопасность и уменьшить риски, – внешние инспекции могут стать ценным подспорьем в выполнении этих обязательств, а заодно послужат поводом продемонстрировать, что руководство ответственно относится к этим вопросам.

Обучение на чужом опыте и совершенствование

Чтобы понять, дает ли результаты ваш подход к обеспечению безопасности, нужен какой-то способ измерения эффективности. Внешняя инспекция не дает идеально полной оценки всех аспектов программы безопасности, но предоставляет количественные и качественные данные, на основе которых можно определить сильные и слабые стороны программы и использовать их для ее совершенствования.

Объективность

Внешние инспекторы не понимают вашу среду так же хорошо, как вы. И хотя недостаток конкретных знаний иногда порождает проблемы в отношениях с внешними консультантами, объективность их оценок может оказаться бесценной. У них нет той

предвзятости, априорных предположений и предрассудков, которые характерны для внутренней команды, поэтому их мнение с полным основанием можно назвать *свежим взглядом*. Этот взгляд может увидеть такие проблемы, которые вы вообще не замечали или не ожидали встретить, – несмотря на незнакомство с вашим бизнесом и системами.

Ожидания заказчиков

Наконец, если вы создаете продукт или службы для кого-то другого (возможно, за деньги), то заказчик вправе ожидать, что вы привлечете внешних консультантов для оценки безопасности как самого продукта, так и среды, в которой он эксплуатируется. Все чаще в процесс закупки включается требование о предоставлении доступа к полному отчету о результатах сторонней экспертизы или к пояснительной записке для руководства, а также к информации о мерах, предпринятых для устранения недостатков. Если вы не сможете представить заказчику таких документов, то вполне можете потерять заказ вследствие недоверия к вашему решению.

Доказательство отрицания

Перед тем как инвестировать в тестирование безопасности (внутреннее или внешнее), вы должны полностью смириться с мыслью о том, что сколько бы денег ни потратить на процесс тестирования, вы никогда не сможете доказать безопасность системы или процесса. Максимум, на что можно рассчитывать, – что привлеченные тестировщики не смогут найти никаких проблем. Доказательство отрицания, т. е. что приложение не является небезопасным, возможно только в очень частных случаях с применением техники формального доказательства и специально спроектированных языков (и все это выходит далеко за рамки этой книги).

Следует также понимать, что приложения и среда, в которой они функционируют, динамически изменяются, поэтому оценка их безопасности справедлива только на протяжении некоторого периода, а то и всего одной точки во времени. И это сугубо и трегубо относится к гибким средам, где кодовая база может изменяться, а иногда и развертываться непрерывно.

Но не впадайте в отчаяние! Оценка безопасности приложения или среды призвана заставить вас сделать все возможное для сверки своего проекта и его реализации с моделью угроз и привлечения стороннего опыта и взгляда на мир для противопоставления вашему собственному. Цель редко состоит в том, чтобы доказать стопроцентную неуязвимость мишени, нужно лишь доказать, что вы предприняли все меры надлежащей осмотрительности, чтобы убедиться в ее безопасности.

Существует немало служб проверки безопасности, и важно выбрать ту, которая отвечает вашим потребностям. Это означает необходимый баланс опыта, стоимости и глубины оценки.

Рассмотрим наиболее распространенные из предлагаемых вариантов оценки и инспекции, а также их ограничения и типичные заблуждения по поводу того, что именно они дают.

Оценка уязвимости

Оценка уязвимости – обычно самый дешевый вид внешней оценки. Она позволяет получить общее представление о безопасности среды, но без глубины, характерной для других подходов. В этом случае вы подряжаете стороннюю организацию, которая использует ряд инструментов в качестве сканеров уязвимости, ставя своей целью найти типичные ошибки в конфигурации, программы с известными проблемами, для которых не установлены исправления, учетные данные со значениями по умолчанию и другие хорошо известные риски безопасности. По завершении сканирования результаты оформляются в виде отчета, который можно вручить аудиторам. Отчет содержит сводную оценку риска и общий план устранения недостатков.

Даже если вам доступны инструменты, с помощью которых можно выполнить автоматизированное сканирование на безопасность самостоятельно, все равно имеет смысл нанять кого-то, имеющего более богатый опыт сканирования и оценки его результатов. Сами по себе инструменты для оценки уязвимости редко бывают сложными, но организация, для которой это основной бизнес, наверняка сумеет правильно настроить их с учетом конкретной среды, поможет интерпретировать результаты и подскажет, как лучше всего устранить недочеты.

Профессионал, которому много раз приходилось работать с этими инструментами, не будет полагаться на умолчания, он должен знать, как их правильно настроить и какие политики и подключаемые модули дают наиболее полезные сведения.

Типичные недостатки автоматизированных инструментов сканирования на безопасность – объем генерируемой информации, недостаток окружающего контекста и огромное количество ложноположительных сигналов. Поэтому человек, знакомый со сканированием на уязвимость и соответствующим инструментарием, оправдает потраченные на него деньги тем, что проанализирует результаты, чтобы сократить их объем, убрать ложноположительные предупреждения и добавить зависящий от среды контекст, чего сами инструменты сделать не могут.

Инструменты и сканеры для оценки уязвимости

Существует целый ряд доступных инструментов для оценки уязвимости, которые можно использовать на разных уровнях технологического стека. Среди них имеются как инструменты с открытым исходным кодом, так и коммерческие сканеры и платформы тестирования.

Более подробно мы рассмотрим средства сканирования на уязвимость и соответствие нормативным требованиям в главе 13.

Нанимая внешнего эксперта, вы прежде всего ожидаете, что он должным образом профильтрует результаты и расположит их в порядке важности, чтобы вы могли максимизировать рентабельность, работая над устранением недостатков. Ваши собственные системные администраторы и эксплуатационники могут отнестись к находкам пренебрежительно («да такого никогда не случится»), тогда как объективный сторонний специалист поможет понять, какие риски следует устранить в первую очередь.

Кроме того, некоторые уязвимости сами по себе могут показаться не несущими особого риска, поэтому на них не обращают внимания. Но опытный профессионал, знакомый с тем, как совокупность нескольких уязвимостей позволяет организовать хитроумную компрометацию системы, может увидеть, когда малозначительные уязвимости в сочетании с другими рисками среды способны создать более серьезный риск.

Важно различать оценку уязвимости и другие, более сложные формы оценки. По самой своей природе оценка уязвимости способна только дать перечень (потенциальных) уязвимостей и ошибок конфигурации. Она хороша, когда нужно найти то, что лежит на поверхности, и получить общее представление о состоянии среды с точки зрения безопасности, но не претендует на полноту и творческий подход к тестированию безопасности вашей среды. Получить заключение «здоров» от процедуры оценки уязвимости – конечно, вещь хорошая, но не впадайте в ошибку, посчитав, что оно гарантирует отсутствие дефектов в среде и ее *безопасность*.

Тестирование на проникновение

Мы уже кратко говорили о тестировании на проникновение и о том, как квалифицированный тестировщик безопасности в «белой шляпе» пытается подобраться к вашей сети или приложению. Тестирование на

проникновение – одна из наиболее распространенных форм оценки безопасности и, к сожалению, одна из наименее понятных публике.

Тестирование на проникновение неотделимо от определения программы тестирования

Тестирование на проникновение – это инспекция в определенных границах. Это значит, что обычно вы поручаете команде протестировать определенный компонент или уровень технологического стека и отводите на это фиксированное время.

Тестирование на проникновение может быть ограничено любым аспектом развертывания приложения, и важно, чтобы этот аспект был зафиксирован в момент заключения договора.

Тестирование может быть ограничено:

Сетью

Развернутой инфраструктурой приложения, включая программы и службы, к которым открыт доступ.

Приложением

Приложением целиком со всеми его развернутыми компонентами, исключая уровень операционной системы и хостинга.

Мобильным приложением

Мобильными компонентами приложения, обычно отделенными от операционной системы, в которой они работают.

Интеграцией

Точками интеграции развернутых компонент, в состав которых иногда включают сторонние системы и пересечение архитектурных границ доверия.

API

Раскрываемые API отдельно от компонент пользовательского интерфейса. Сюда может входить или не входить тестирование конфигурации шлюза к API или компонентов координации (orchestration).

Определение границ играет роль также при обсуждении времени, отведенного для оценки, и того, на чем акцентировать внимание: базовый проект и архитектура или реализация.

В хорошей процедуре тестирования на проникновение сочетается несколько факторов:

- дисциплинированный и структурированный подход к определению программы тестирования и управлению его проведением;

- тестировщики, имеющие доступ к подходящему инструментарию, поддерживающему эффективную и глубокую оценку;
- знание и опыт работы с оцениваемыми системами и языками;
- способность на основе этих принципов быстро и творчески адаптироваться к изменениям среды.

Это может показаться общим местом, но, заказывая тестирование на проникновение, вы получаете то, за что платите, а если срезать углы или нанимать неопытных либо низкоквалифицированных тестировщиков, то можно получить лишь немногим больше, чем дает общая оценка уязвимости.

Тестирование на проникновение чаще всего является последней проверкой безопасности, перед тем как передавать новую основную версию системы в эксплуатацию. Большинство организаций подряжают специализированную компанию с хорошей репутацией, отводят ей для работы примерно две недели, получают отчет, а потом спешно стараются исправить все, что можно, за оставшееся до сдачи проекта время. Во многих случаях дата сдачи не подлежит изменению, поэтому исправления и прочие меры по смягчению рисков могут даже не войти в начальную версию.

Такие организации лишают себя возможности получить настоящую пользу от тестирования на проникновение: узнать, как мыслит и действует противник, и воспользоваться этой информацией, чтобы улучшить свои системы, а также способы их проектирования, сборки и тестирования. Этот подход можно описать термином *защита в ответ на атаку* (attack-driven defense), то есть, глядя на свое приложение или среду глазами противника, непрерывно стремиться к улучшению выбранных защитных механизмов в порядке их важности.

Ваша цель состоит не в том, чтобы «пройти тест» или «сдать экзамен». Организация, которая так смотрит на вещи, может «смухлевать» в игре с тестировщиками проникновения, предъявив им не всю работу или ограничив сообщаемую им информацию и надеясь в результате свести к минимуму количество найденных проблем. Зачастую она оправдывается тем, что тестирование методом черного ящика более реалистично, т. к. тестировщик, как и противник, изначально имеет в своем распоряжении одну и ту же ограниченную информацию.

Но чем больше информации у тестировщика проникновения, тем лучше он сделает свою работу и тем больше вы сможете узнать. Тестирование методом *серого ящика*, когда тестировщик знает, как ра-

ботает система, хотя и не так хорошо, как разработчики, не идет ни в какое сравнение с методом черного ящика. Следующий шаг – тестирование методом белого ящика, когда тестировщикам предоставляется доступ ко всему исходному коду и разрешается комбинировать анализ кода, инспекцию архитектуры и динамическое практическое тестирование.

В зависимости от того, что разрешено и не разрешено делать внешней команде тестирования на проникновение, можно разработать много разных сценариев атак. Но чтобы извлечь реальную пользу из столь различных вариантов, организация безопасности должна быть достаточно зрелой. Гораздо чаще встречается ситуация, когда рассматриваемое решение впервые подвергается агрессивной оценке безопасности и выделенный на эти цели бюджет ограничен. В таких случаях лучше всего будет просто не мешать тестировщикам и дать им возможность наилучшим образом применить свои знания, чтобы сначала выявить те части системы, где они видят наибольшее количество красных флажков, а затем копнуть глубже и продемонстрировать успешные атаки и их последствия.

В этом смысле оценка методом белого ящика позволит тестировщикам не тратить зря время, чтобы понять ограничения системы и встроенные в нее средства безопасности, а сразу получить ответы на эти вопросы, заглянув в исходный код. Обычно оценка методом белого ящика в течение одной-двух недель обеспечивает куда большее покрытие, а также более полное исследование направлений атаки, чем тестирование методом черного или серого ящика. К тому же зачастую вы получите более конкретные рекомендации по устранению недостатков, поскольку тестировщики видели код и вполне могут точно указать ошибку, стоящую за обнаруженной проблемой. Это намного упрощает работу команды, отвечающей за своевременное исправление ошибок.

Как мы видели в главе 11, ручное тестирование, включая и тестирование на проникновение, не может поспеть за темпом гибкой разработки или непрерывной поставки в случае DevOps. Но чтобы удовлетворить нормативным требованиям (как в стандарте PCI DSS) и обеспечить контроль, все равно может понадобиться планировать тестирование на проникновение на регулярной основе, обычно не реже раза в год. Однако его следует рассматривать не как контрольно-пропускной пункт, а, скорее, как подтверждение успехов и ценный опыт для всей команды.

Команда красных

Команда красных проводит активную атаку против вашей работающей системы, чтобы проверить не только безопасность среды в режиме повседневной эксплуатации, но – и это важнее – вашу способность обнаруживать инциденты и реагировать на них, а заодно и другие защитные механизмы.

Цели и подходы отличаются от тестирования на проникновение. Вместо того чтобы пытаться обнаружить и расположить по важности уязвимости, подлежащие устранению, команда красных активно эксплуатирует уязвимость (или цепочку уязвимостей) либо применяет методы социальной инженерии, чтобы проникнуть в сеть и посмотреть, сколько времени потребуется эксплуатационникам или безопасникам, чтобы обнаружить их присутствие и как-то отреагировать. Такое приглашение внешней команде напасть на вашу среду, как это сделал бы реальный противник, известно также под названием *имитация атаки*, или *целеустремленная атака*. На действия команды красных не накладывается почти никаких ограничений, а иногда даже декларируется, что *разрешены все приемы*.

Команды красных находят важные уязвимости в системе. В таких мероприятиях нередко даже обнаруживаются уязвимости нулевого дня как во внутренних, так и в коммерческих приложениях, используемых в организации.

Но истинный их смысл – научить группы эксплуатации и безопасности, как улучшить методы борьбы с реальными атаками, которые проводит хорошо мотивированный и квалифицированный противник.

Конечные результаты деятельности команды красных, вообще говоря, сильно отличаются от результатов тестирования на проникновение. Команда красных зачастую представляет результаты в виде дневников атак, деревьев атак, содержащих те пути сквозь среду, которые привели к достижению цели, и подробных описаний нестандартных инструментов и эксплойтов, написанных специально под данный заказ. Это позволяет гораздо глубже понять склад ума атакующего, когда ответ на вопрос *почему* не менее важен, чем на вопрос *что*.

Направления неудачных атак, т. е. пути, которые тестировщики пробовали, но безуспешно, также очень важны, поскольку дают оценку эффективности имеющихся средств контроля и некоторые качественные показатели инвестиций в защиту.

Привлечение команд красных обходится недешево, и для извлечения максимальной пользы в организации уже должна присутствовать до-

статочно зрелая система безопасности. На ранних этапах становления этой системы деньги лучше потратить на проведение более обширных или более частых тестов на проникновение, чем на более интересные, но и гораздо более затратные имитации атак.

Сравнение ограничений по времени и целям с имитацией атаки

При планировании программы внешнего тестирования или инспекции важно понимать, что вы можете определить условия участия сторонней организации.

Обычно тестирование на проникновение ограничивается во времени: команде выделяется *x* дней или недель на тестирование предъявленных систем и представление результатов анализа.

С другой стороны, команда красных может быть ориентирована на достижение цели без ограничений по времени или набору систем. При таком подходе команде ставится цель, например: «изменить транзакцию» или «получить права суперпользователя в производственной системе». На то, как команда добьется этого, налагается значительно меньше ограничений, поэтому результаты могут получиться более репрезентативными.

Если говорить о временных рамках, то действия команды красных, в которых принимали участие авторы этой книги, продолжались от шести недель до шести месяцев, причем иногда в договоре даже встречалась фраза «столько времени, сколько понадобится».

Но если вы чувствуете, что созрели для плодотворной работы с командой красных, то полученный опыт может оказаться наиболее поучительным и взывающим к смирению из всего, что когда-либо испытывала ваша команда безопасности. Это мероприятие может дать крайне содержательные качественные данные, которые позволят ответить на вопросы, недоступные тестированию на проникновение и оценке уязвимости, в большей степени ориентированным на количественные результаты.

Еще раз повторим, что в центре внимания должно быть обучение. Полученные в итоге глубокие и всесторонние знания о том, как группа мотивированных и квалифицированных противников проникла в вашу среду, должны быть использованы для уточнения подходов к мерам обнаружения и защиты от вторжений.

Некоторые организации, особенно крупные поставщики финансовых и облачных служб, содержат штатные команды красных, которые постоянно прогоняют тесты и проводят имитации атак, результаты кото-

рых используются для повышения безопасности и отработки реакции на инциденты. Мы еще вернемся к этому подходу в главе 13.

Вознаграждение за обнаружение ошибок

Если и был какой-то тип внешней оценки, который поднял переполох в СМИ за те полтора-два года, что предшествовали написанию этой книги, так это вознаграждение за обнаружение ошибок (bug bounties). Но всякий переполох неизбежно и к глубокому сожалению сопровождается недопониманием, недоразумениями и прямой дезинформацией. В общем, будьте покойны – то, что ваша организация еще не объявила награды за обнаруженные ошибки, вовсе не означает, что она преступно пренебрегает своими обязанностями, что бы по этому поводу ни писали в Интернете. На самом деле обеспокоиться обвинениями в небрежении нужно, скорее, тем, кто попался на эту удочку, но при этом не обеспечил свою безопасность в такой степени, чтобы справиться с последствиями. Тем, у кого нет солидной программы безопасности приложений, надо бы сначала заняться этим, а уже потом думать о вознаграждении за обнаружение ошибок. А иначе ошибок найдут столько, что вы не будете успевать их исправлять.

При всем при том вознаграждение за обнаруженные ошибки, безусловно, заслуживает внимания, но применять его надо, когда ваша программа обеспечения безопасности уже достаточно зрелая. В следующих разделах мы попытаемся продраться сквозь рекламную чушь и понять, что действительно может дать объявление награды за найденные ошибки.

Как работает программа вознаграждения

Прежде всего определим, что вообще такое вознаграждение за найденные ошибки.

Идея проистекает из того непреложного факта, что любую систему или приложение с выходом в Интернет постоянно атакуют, нравится вам это или нет. Некоторые атаки по природе своей автоматизированы и являются результатом всякого рода сканирований, которые прочесывают весь Интернет, другие более целенаправленны – тут уж на водительском месте сидит не робот, а человек. Но в любом случае вы должны смириться с тем, что активные попытки найти уязвимости в ваших системах происходят прямо сейчас, когда вы читаете эти строки.

Программа вознаграждения за найденные ошибки – всего лишь рациональное следствие признания сего факта. Раз уж так сложилось, то

поощрим всех желающих покопаться в наших работающих системах, а главное – призовем их сообщать нам о том, что они накопали.

Более того, вы публично обещаете, что за все предъявленные вам проблемы, которые вы сможете воспроизвести (или удовлетворяющие некоторому заранее сформулированному критерию), нашедший получит некоторое вознаграждение. Будучи изложен в таких словах, этот подход кажется вполне логичным и рациональным, но не так давно большинство организаций по умолчанию натравливало адвокатов на любого, кто привлекал внимание к обнаруженным проблемам. К сожалению, до сих пор существуют организации, которые тянут в суд людей, сообщивших им о найденной уязвимости, вместо того чтобы выслушать их, попытаться понять и вознаградить. Впрочем, будет справедливо сказать, что такое поведение все чаще считается архаичным, и мы надеемся, что читатели данной книги не пойдут по этому пути.

Количество организаций, готовых выплачивать вознаграждение за найденные ошибки, за недолгое время значительно выросло, и теперь это не только такие крупные игроки, как Google, Facebook, Microsoft и им подобные, но даже армия США, что, казалось бы, совсем неожиданно, ну и многие, многие другие со всего веба. Программа вознаграждения принесет безусловную пользу любой организации, укомплектованной высококвалифицированным штатом специалистов по безопасности – лучшими из лучших.

Подготовка к программе вознаграждения за найденные ошибки

Но решение запустить такую программу и вступить в контакт с уже сформировавшимися сообществами, трепещущими в ожидании охоты, не стоит принимать без должной подготовки.

Прежде всего вы должны понимать, что программа вознаграждения подразумевает привлечение сообщества, а это потребует финансовых и временных затрат в той или иной форме. Считать вознаграждение за найденные ошибки дешевым вариантом тестирования на проникновение неверно от начала до конца: вы ставите на кон доброе имя своей организации и строите отношения между работающими у вас специалистами по безопасности и более широким сообществом. Рассматривать это сообщество как способ сэкономить на найме профессиональных тестировщиков проникновения или поставить галочку – мол, мы *действительно* думаем о безопасности – значит напрашиваться на неизбежные неприятности.

Ниже приводится список рекомендаций, которые стоит иметь в виду, размышляя о запуске программы вознаграждения. Этот список появился в результате непростого опыта авторов, не побоявшихся прибегнуть к этому относительно новому способу оценки безопасности.

Своими или сторонними силами?

Одно из первых решений, которые вам предстоит принять, – привлекать ли к поиску ошибок за вознаграждение только собственный персонал или поручить организацию программы сторонней компании. Внутренняя программа оставляет контроль над вознаграждением в ваших руках, так что вы можете установить его точно так, как считаете нужным. Если у вас уже есть сложившаяся группа безопасности, то это может обойтись дешевле, чем платить третьей стороне за то, за что штатные безопасники и так уже получают зарплату, – имеются в виду классификация и анализ ошибок.

Но у сторонних программ имеется сложившееся сообщество готовых включиться исследователей, оно поможет отфильтровать сигнал от шума в потоке сообщений об ошибках. Программа может стартовать в очень короткие сроки. На момент написания книги наибольшей популярностью пользовались две службы организации программ вознаграждения: Hackerone (*https://hackerone.com/*) и Bugcrowd (*https://bugcrowd.com/*). Сторонняя компания не может сделать всё, и неизбежно наступает момент, когда люди, знакомые с приложением, должны будут сопоставить найденную проблему с моделью угроз, а также решить, как с ней быть.

Правила проведения

В реальных атаках против ваших систем никаких правил нет, но разумно ожидать, что участники программы вознаграждения за найденные ошибки будут соблюдать определенные правила. Обычно в правилах оговариваются такие вещи, как доменные имена или приложения, считающиеся мишенями атаки, типы уязвимостей, за которые вознаграждение выплачивается и, наоборот, не выплачивается, типы запрещенных атак (например, все, приводящее к отказу от обслуживания, обычно исключается). В правилах могут быть также юридические оговорки, обусловленные законами и положениями, действующими в компании или в стране ее пребывания. Безусловно, имеет смысл проконсультироваться с юристами по поводу формулировки правил программы вознаграждения.

В Интернете нет недостатка в примерах правил проведения атак за вознаграждение, которые можно было бы взять за основу, но неплохой отправной точкой могут служить «Правила Google участия в программе вознаграждения за найденные уязвимости» (*https://www.google.com/about/appsecurity/reward-program/index.html*), поскольку это устоявшаяся и пользующаяся уважением программа.

Вознаграждение

Мы рекомендуем с самого начала объявить, на какое вознаграждение может рассчитывать лицо, обнаружившее уязвимость. Так вы положите начало открытым и честным отношениям с сообществом. В качестве вознаграждения могут выступать наличные, сувениры типа футболок или стикеров (если обнаружение уязвимости – единственный способ получить такой сувенир, тем лучше, эксклюзивность всегда в цене) или публичное признание самой ошибки.

Во многих программах имеется шкала вознаграждений в зависимости от серьезности найденной проблемы. Иногда вознаграждение увеличивается, если это не первая находка данного участника – чтобы поощрить человека к продолжению сотрудничества и более глубокому знакомству с приложениями. Хотя в большинстве программ предлагается широкий спектр вознаграждений, стоит отметить, что к вашей программе будут относиться более серьезно, если вы расплачиваетесь наличными, а не футболками и стикерами (хотя футболки и стикеры тоже нужно посылать обязательно!).

Размер наградного фонда

У большинства компаний бюджет программы вознаграждения за найденные ошибки ограничен, поэтому вы должны решить, какую сумму сможете выплатить участникам в течение квартала или года. У разных организаций требования и ограничения разнятся, но один из авторов книги следовал такому эвристическому правилу: годовой фонд программы должен быть равен стоимости одного полного теста на проникновение. При этом бюджет программы легко рассчитать, и финансирование легко обосновать как расширение существующей программы тестирования на проникновение.

Зал славы

Создание зала славы или иной способ признания заслуг тех, кто проявил себя в программе вознаграждения за найденные ошиб-

ки и тем помог сделать ваше приложение безопаснее, – очень важный штрих для сообщества, с которым вы завязали контакты. Имена, аккаунты в Твиттере, ссылки на сайты, а также тип уязвимости и дата ее обнаружения – типичные атрибуты, которые должны присутствовать в зале славы. Некоторые компании идут дальше и оформляют зал славы как игру, в которой участники, нашедшие несколько ошибок, получают специальные рейтинги или знаки отличия. Прекрасный пример забавного и подстегивающего самолюбие зала славы создан на Github (*https://bounty.github.com/*). Проявите изобретательность в способах признания заслуг исследователей – и это окупится участием и одобрением.

Структура отчетов

Если разрешить отчеты о найденных проблемах в свободной форме, то могут возникнуть шероховатости, которые лучше бы предотвратить в самом начале. Предложив участникам некую структурированную форму извещения о проблемах, вы будете знать, что получаете всю информацию, необходимую, чтобы своевременно проанализировать и рассортировать сообщения. Не все участники – профессионалы в области безопасности, и, ясно указав, какая информация вас интересует, вы поможете им включать в отчет все, что вам нужно знать. Это не значит, что отчеты в свободной форме следует полностью запретить, просто вы должны попытаться предложить участникам какие-то ориентиры.

Пример формы отчета приведен на странице компании Etsy, посвященной программе вознаграждения (*https://www.etsy.com/bounty*). У такого подхода много преимуществ, и помимо бесконечной переписки с участником в попытке получить именно те сведения, которые вам нужны. Одно из них – возможность вести точный учет показателей программы: вида найденных проблем, их частоты и изменения тенденций со временем. Другое – взаимно-однозначное соответствие между отчетами и проблемами, поскольку включение нескольких проблем в один отчет только приводит к путанице. Каждой проблеме можно назначить учетный номер и связать с ним ответственных за ее разрешение и потраченное на это время.

Быстрая реакция, открытое и уважительное общение

Как уже было сказано, установление правильных отношений с сообществом исследователей, принимающих участие в поиске ошибок, – ключ к успеху программы. Зачастую первый шаг в этом

направлении прост: без задержки ответить нашедшему уязвимость и держать его в курсе процесса проверки и исправления.

Важно также понимать, что ни ваш родной язык, ни английский – язык общения в Интернете, – необязательно является родным для участника программы, поэтому, просто для того чтобы понять, о чем речь, может потребоваться несколько раундов переписки. Иногда это всех раздражает, но вежливая просьба уточнить с ясным выражением наилучших намерений – лучший способ построить доверительные и уважительные отношения с сообществом.

Кстати, следует вырабатывать умение вежливо отшивать авторов, которые присылают заведомо ошибочные сообщения или демонстрируют непонимание обсуждаемого вопроса (или компьютеров вообще!), поскольку вы наверняка будете получать кучу чепухи.

Своевременная оплата

Тут и говорить не о чем: коль скоро сообщение подтверждено и описанная проблема отвечает всем правилам, то крайне важно вовремя заплатить автору. Вы должны заранее обдумать, как собираетесь выплачивать денежное вознаграждение. В разных программах это делают по-разному. Иногда отправляют предоплаченные кредитные карты (бывает даже, что с индивидуальным дизайном), а иногда переводят средства с помощью PayPal или на банковский счет.

Учитывая разнообразие стран и налогового законодательства, проведение платежа может оказаться непростым делом. Например, на момент написания книги невозможно было переводить деньги через PayPal в Индию и Турцию – страны, из которых вы, скорее всего, будете получать много сообщений. Вы должны заранее изыскать способы выплаты вознаграждения исследователям из этих и других стран, чтобы не возникало неожиданных задержек платежей. Платежи – один из тех аспектов программы, где сторонняя служба может избавить вас от хлопот, поскольку контактировать с участниками и выплачивать им причитающееся будет именно она, так что не стоит недооценивать пользу такого сотрудничества.

Это дорога в одну сторону

Запустив публичную программу вознаграждения за найденные ошибки, будет довольно трудно свернуть ее впоследствии. Так что, решившись, помните, что это стиль жизни, которого вы

должны будете придерживаться, а не преходящее увлечение, от которого можно будет легко отказаться, если надоест.

Не стоит недооценивать первоначальный всплеск интереса

Запуская программу вознаграждения, вы в каком-то смысле открываете шлюзы. Будьте готовы к первой волне спроса, поскольку именно на этом этапе складываются ваши отношения с сообществом. Первое впечатление всегда важно. В первые две недели количество сообщений, вероятно, будет гораздо больше, чем в среднем на протяжении программы, поэтому зарезервируйте время и ресурсы на проверку, разборку и сортировку этого потока. Не исключено, что команда, отвечающая за обработку сообщений, в течение этих двух недель не сможет заниматься больше ничем. Запускайте программу в такое время, когда загруженность позволяет.

Обязательно будут дубликаты, однотипные сообщения и просто мусор

С самого начала нужно иметь в виду, что всегда найдутся люди, которые хотят обыграть систему – заработать, сделав как можно меньше, или даже получить деньги, указав на то, что вообще не является проблемой. Смиритесь с тем фактом, что большинство полученных отчетов не станут поводом для вознаграждения, это просто шум.

Следует также понимать, что стоит появиться известию о новой уязвимости или проблеме, как все сообщество набросится на каждый сайт, пытаясь найти ее и получить за это награду. У вас должен быть четкий план по работе с повторными сообщениями о действительно существующей уязвимости (часто награду получает только тот, кто прислал сообщение первым). Вы также должны решить, что делать с сообщениями, в которых указывается на риск, с которым вы решили смириться, или на нечто, являющееся задуманной функцией системы.

А вы уверены, что хотите запустить программу вознаграждения?

Наше головокружительное путешествие в мир, где платят за найденные ошибки, почти окончено, поэтому впору спросить, какие могут быть причины не хотеть принять участие в мероприятии, о котором говорят как о величайшем достижении человечества? Для начала подумайте, от

кого вы это услышали. Чаще всего рекламой занимаются люди, которые кровно заинтересованы в как можно более широком распространении таких программ – и желательно на их платформе. Но даже если отвлечься от чисто маркетинговых аспектов этой шумихи, есть и другие причины отказаться от участия в программе вознаграждения.

Программы вознаграждения как средство получить больше от тестировщиков проникновения

Как бы полезна ни была программа вознаграждения за найденные ошибки, она никогда не заменит опытную команду тестировщиков проникновения. Но иногда ее можно использовать, чтобы повысить эффективность этих тестов. Идея в том, что если вы платите за тестирование вашей среды на проникновение, то тестировщики захотят предъявить какие-то результаты. Если они вернутся с пустыми руками, то вряд ли их позовут снова. Если критических проблем найти не удается, то тестировщики часто включают в отчет тривиальные или малозначимые вещи просто потому, что обязаны что-то показать. Это не лучший способ потратить ваши деньги и их время.

Если вы запустили программу вознаграждения за найденные ошибки, то ваши приложения будут постоянно сканироваться в поисках низко висящих плодов, за срывание которых выплачивается небольшая премия. Это лишит тестировщиков легких находок, которыми можно было бы заполнить отчет. В таких случаях компания, занимающаяся тестированием, вероятно, бросит на вашу систему дополнительные силы или привлечет более опытных сотрудников, поскольку срок истекает и надо же что-то предъявить. Находящиеся в отчаянном положении тестировщики иногда проявляют чудеса изобретательности и могут найти глубоко скрытые ошибки.

Таким образом, программа вознаграждения может не только способствовать непрерывной оценке и обнаружению уязвимостей, но и заставит сторонние компании, работающие за солидные деньги по контракту, трудиться интенсивнее и приносить более ценные результаты.

Вы должны располагать уже сложившимся механизмом реагирования

Хотя ваши системы подвергаются атакам постоянно, распахнуть окно в Интернет с криком «А ну, ребята, налетай» – это уже совсем другой уровень. Вы должны быть уверены в своей способности оценить проблемы, о которых вам сообщили, отреагировать на них и внести исправления – и все это в сжатые сроки. Если вы не

готовы ко всему этому, то программа вознаграждения за найденные ошибки принесет больше вреда, чем пользы.

Постоянство программы вознаграждения имеет как плюсы, так и минусы. Привлечение тестировщиков по графику позволяет зарезервировать время для сортировки и ответных действий. А сообщения о новых находках в рамках программы вознаграждения могут поступать в любое время дня и ночи и иногда требуют немедленной реакции (даже если и проблемы-то никакой нет). Это грозит эмоциональным выгоранием команды, проявляющимся по-разному, но всегда плохо.

Низкое отношение сигнала к шуму

Если вы думаете, что, запустив программу вознаграждения, получите в ответ круглосуточное высококачественное тестирование на проникновение, то это зря. Большая часть сообщений – ложноположительные сигналы. А из реальных найденных проблем большинство – малозначимые уязвимости. Но все они требуют расследования, каких-то действий, переписки – даром ничто не дается. Вы, конечно, можете договориться о помощи с компанией, отвечающей за проведение программы, но тут столкнетесь со следующей проблемой.

Передача программы вознаграждения на сторону означает, что марку вашей безопасности будет представлять кто-то другой

Партнерские отношения со сторонним поставщиком платформы проведения программ вознаграждения означают, что вы передаете на сторону и марку своей безопасности (см. главу 15) – вещь, которую трудно приобрести и очень легко потерять. Поручив третьей стороне представлять безопасность своих приложений и среды, вы даете ей возможность непосредственно влиять на доверие, испытываемое к вашим продуктам и организации клиентами и более широкой публикой. Такое решение не стоит принимать с легким сердцем.

Граница между найденной за вознаграждение уязвимостью и инцидентом очень зыбкая

Когда попытка найти ошибку в расчете на награду превращается в настоящий инцидент в системе безопасности? На этот вопрос трудно дать определенный ответ, и тут у вас нет никакого контроля, кроме опубликования правил. Участники программы стремятся максимизировать серьезность и последствия найден-

ных уязвимостей, поскольку им платят тем больше, чем *страшнее* проблема (в отличие от тестировщиков проникновения, которые обычно получают фиксированную оплату вне зависимости от того, что найдут).

Это не раз приводило к ситуациям, когда *чрезмерно ретивый* (говоря вежливо) охотник за наградой, пытающийся продемонстрировать серьезность своей находки, или кто-то, недовольный ответом или полученным вознаграждением, пересекал границу, за которой начинается реальный инцидент, что приводило к печальным последствиям для реальных данных клиентов и требовало формальной и полноценной реакции. Цена всего одного такого инцидента вкупе с неблагоприятным пиаром сводит на нет всю программу вознаграждения.

Программа вознаграждения за найденные ошибки может иметь неблагоприятную огласку

Имейте в виду, что иногда член привлеченного сообщества может оказаться недовольным тем, как вы классифицировали найденную проблему, ответили ему или заплатили, и будет искать удовлетворения, публично огласив свою версию истории. Угрозы *опубликовать это в моем блоге* или *рассказать обо всем этом в Твиттере* встречаются регулярно и время от времени приводят к не слишком позитивной огласке. И еще одно предостережение – вы должны предполагать, что в один прекрасный момент все общение с сообществом охотников за наградами может выйти на публику, поэтому ведите себя достойно, каким бы надоедливым и неприятным ни казался собеседник.

Отдача со временем становится меньше, а затраты растут

Нелицеприятная правда заключается в том, что с ростом количества программ вознаграждения за найденные ошибки сообщество исследователей, благодаря которому эти программы приносят успех, не растет пропорционально. Это означает, что, для того чтобы привлечь интерес сообщества, вы должны сделать свою программу более заманчивой, чем у соседа. То есть увеличить выплаты, из-за чего затраты на программу возрастут.

Следует также учитывать, что наибольшая рентабельность будет наблюдаться в начале программы, когда обнаруживается большинство простых ошибок, а количество по-настоящему серьезных проблем с опасными последствиями со временем будет уменьшаться. В пределе это означает, что, начиная с какого-то

момента, программа вознаграждения перестает быть экономически выгодным предприятием.

Окончательное решение о том, стоит ли ввязываться в это дело, принять непросто, и только вы сами можете определить, пришло для этого время или еще нет. Просто будьте осторожны и не действуйте очертя голову, не взвесив предварительно все плюсы и минусы для организации.

Стандарт обнаружения уязвимостей ISO 29147

Международная организация по стандартизации сочла обнаружение уязвимостей настолько важным процессом в системе обеспечения безопасности, что посвятила ему отдельный стандарт. В стандарте ISO 29147 (*http://bit.ly/iso-publicly-available-standards*) описаны процесс управления обнаружением уязвимостей и требования к нему. В настоящее время его можно скачать бесплатно.

Вне зависимости от того, как вы собираетесь построить программу вознаграждения за найденные ошибки: с помощью сторонней компании или собственными силами, – этот стандарт объяснит, как подходить к уязвимостям, найденным внешними исследователями, и что вы обязаны сделать для оповещения о них и устранения ошибок.

Инспекция конфигурации

Инспекция конфигурации – это ручная или основанная на сканировании инспекция конфигурации системы на предмет соблюдения передовых практик или стандартов типа эталонных тестов CIS, которые мы рассмотрим в главе 13.

Это может быть проверка состояния платформы выполнения: операционной системы, виртуальных машин или контейнеров и исполняющей среды типа Apache или Oracle – всего, что трудно настроить правильно.

В той мере, в какой это возможно, такую проверку следует заменить регулярным запуском автоматизированного сканирования на соответствие нормативным требованиям.

Аудит безопасности кода

Даже если команда обучена безопасному кодированию и последовательно пользуется средствами автоматизированной инспекции кода, бывают случаи, когда нужно пригласить специалиста по безопасности

для аудита кода. Это особенно ценно на ранних стадиях проекта, когда вы выбираете и настраиваете каркасы и хотите, чтобы кто-то подтвердил правильность выбранного подхода. Надобность во внешних инспекторах возникает также в случае взлома или во исполнение нормативных требований.

Большинство аудиторов начинает с автоматизированной инспекции кода, чтобы найти очевидные ошибки и прочие низко висящие плоды. Затем они концентрируются на функциях и средствах контроля безопасности, а также на защите конфиденциальных данных. Но хороший инспектор найдет и отметит в отчете и другие виды логических ошибок, проблемы в конфигурации среды и неудачные проектные решения.

Аудит кода может стоить дорого, поскольку требует ручной работы и специальных навыков. Необходимо найти людей с основательными знаниями в области безопасности и навыками программирования, которые были бы знакомы с вашей средой разработки и могли быстро разобраться в проекте и принятых командой соглашениях о кодировании.

Аудиторы кода сталкиваются с теми же проблемами, что и другие инспекторы. Им может помешать неудачное оформление, плохо структурированный или слишком сложный код. К тому же аудит – это изматывающая работа. Требуется, по меньшей мере, пара дней, чтобы инспекторы разобрались в коде достаточно хорошо, чтобы начать искать важные проблемы, и еще несколько дней, чтобы выйти на плато.

Поэтому имеет смысл ставить таким инспекциям жесткие временные рамки и в это время быть готовым оказать аудиторам любую возможную помощь, объяснять код и проект, отвечать на возникающие вопросы. Прикрепление к аудиторам опытных разработчиков поможет извлечь максимум пользы, а заодно станет для команды отличной возможностью больше узнать о том, как проводить инспекции безопасности кода своими силами.

Криптографический аудит

Криптографический аудит выделен в отдельную категорию оценки по одной простой причине: это действительно трудно и требует специальных навыков. По криптографическому аудиту меньше всего опытных специалистов и труднее всего получить помощь. Нередко приходится становиться в очередь и планировать аудит на отдаленное будущее, поскольку не хватает людей, владеющих этим искусством. Но не поддавайтесь искушению выбрать простой путь и заменить настоящего специалиста по криптографическому аудиту кем-то более доступным.

Есть две основные причины, вынуждающие обращаться к специалисту по криптографическому аудиту:

1. Вы хотите проверить, правильно ли пользуетесь криптографией в широком смысле. Тут можно задавать такие вопросы: *Получаю ли я желаемые свойства, используя криптографию таким образом? Не получилось ли так, что я случайно подорвал гарантии, которые дает определенный набор криптографических примитивов?*
2. Вы проектируете и (или) реализуете собственные криптографические примитивы и хотите, чтобы кто-то подтвердил правильность проекта и реализации алгоритмов.

Если вы попадаете во вторую категорию, остановитесь. Погуляйте, выпейте стопочку, а затем сядьте и еще раз подумайте, действительно ли реализация новых или даже новаторских криптографических примитивов – лучшее решение стоящей перед вами задачи. Криптография – это сложно, чертовски сложно, и даже совершенно невинный, на первый взгляд, просчет может привести к полному краху.

Для большинства ситуаций, с которыми вы можете столкнуться, существуют готовые, хорошо себя зарекомендовавшие компоненты и паттерны, решающие вашу задачу, так что никакой необходимости изобретать что-то новое нет. Если, несмотря на прогулку и выпивку, вы все же пребываете в убеждении, что нужно придумать новый алгоритм или изменить существующий, или если для используемого вами языка нет реализации нужного вам алгоритма, то лучшее, что мы можем посоветовать, – с самого начала проекта принять в штат необходимых специалистов или обратиться за помощью к стороннему эксперту. Но даже после этого понадобится еще одна пара глаз, чтобы проверить то, что получилось.

Если ваши потребности относятся к первой категории, то все равно правильно применить криптографические примитивы нелегко, и взгляд со стороны был бы крайне полезен. Результаты первой в вашей жизни криптографической инспекции могут оказаться для вас сюрпризом. Даже в тех способах применения, которые казались вам совсем простыми, могут таиться нюансы – вплоть до того, какой компилятор используется для генерации двоичного кода. Не исключено, что вы поставите под вопрос собственное понимание компьютеров и работы с числами вообще. Это хорошо. Как раз за это вы и платите.

Если в вашем решении криптография применяется сплошь и рядом, то надеемся, что написанное выше подчеркнет важность профессионального взгляда. К сказанному остается добавить, что после тща-

тельной проверки кода, имеющего отношение к криптографии, его лучше оставить в покое и изменять в самом крайнем случае. Лучше бы написать к нему интерфейс в виде четко определенного API.

Выбор сторонней компании

При выборе стороннего партнера или консультанта нужно принимать во внимание ряд факторов.

Зачем вообще вам нужна помощь?

- Потому что это нормативное требование?
- Для заблаговременной проверки состояния?
- В качестве меры в ответ на непрохождение аудиторской проверки или взлом?

Какой объем сотрудничества со стороны вашей команды необходим для успеха всего предприятия?

Инспекцию конфигурации или стандартную оценку безопасности можно провести почти без участия вашей команды, по крайней мере до тех пор, пока не придет время изучать отчет. Для вас важны стоимость, доступность и подтверждение квалификации.

Но в случае аудита кода или игры с участием команды красных стороны должны сотрудничать гораздо теснее, а это значит, что вы должны больше знать о привлекаемых людях и о том, сможете ли вы сработаться.

С какого рода оценщиком безопасности вы собираетесь работать?

Существуют разные типы внешних компаний, предлагающих различные услуги.

Фабрики тестирования безопасности

Компании, специализирующиеся на информационной безопасности с четко структурированными предложениями, соответствующими нормативным требованиям. Часто хорошо автоматизированы, действуют в соответствии с определенной методикой, обеспечивают масштабируемость и повторяемость результатов.

Эксперты-консультанты и бутики

Индивидуалы или небольшие группы специалистов, опирающиеся в основном на знания и опыт, а не на методики. Могут подстроиться под конкретного клиента, пользуются собственными инструментами. В этих случаях вы получаете то, за что платите: людей, которые присутствуют на конференциях хакеров-черношляпников, Defcon и OWASP, преподавателей на курсах по

безопасности. Услуги таких компаний дороги (в зависимости от репутации и доступности), их не очень много.

Поставщики услуг для предприятий

Услуги по обеспечению безопасности, предоставляемые в составе более полного пакета ИТ-услуг, включающего другие консультативные услуги, обучение и инструменты для различных программ и проектов. Прекрасно масштабируются, обеспечивают повторяемость, дорого.

Аудитор или оценщик

Услуги хорошо известной аудиторской фирмы, например CA, или другого сертифицированного оценщика (например, PCI QSA). Проверяют по контрольному списку, оценивают соответствие систем конкретным нормативно-правовым требованиям или какому-либо стандарту.

Составить представление о фабрике тестирования безопасности довольно просто, нужно лишь взглянуть на ее методику. Выбрать бутик или частного консультанта сложнее, потому что необходим высокий уровень доверия к его знаниям, опыту и подходу к работе. Ведь вы должны быть уверены, что он сохранит конфиденциальность и станет работать аккуратно, особенно если будет допущен в рабочую сеть.

Опыт работы с продуктами и организациями, похожими на ваши

Насколько важно, чтобы инспекторы или тестировщики имели опыт работ в конкретной отрасли? Или с конкретными технологиями? Все зависит от характера оценки. Сканирование на уязвимости не зависит от отрасли, как и тестирование сети на проникновение. Но для проведения аудита кода или тестирования приложения на проникновение, очевидно, нужен кто-то, знакомый с языком (языками) и прикладной платформой. Знакомство с предметной областью или отраслью значительно упростит его – и вашу – работу, но не является определяющим критерием.

Активная исследовательская работа и повышение квалификации

Важно ли вам, чтобы у тестировщиков или инспекторов были сертификаты?

В этой области базовой оценкой квалификации является сертификат CISSP (Certified Information Systems Security Professional), или SANS (Escal Institute of Advanced Technologies), или сертификат этического хакера (Certified Ethical Hacker). Но очевидно, что гораздо лучше заполучить эксперта, который активно работает в области, чем кого-то, сдавшего экзамен, но не имеющего реального практического опыта.

Наличие сертификатов – это не самое важное в жизни, и мы рекомендуем не переоценивать их значимость.

Встречи с техническими специалистами

Если вы работаете с бутиком или подрядчиком, то важно встретиться с людьми, которые будут заниматься вашим заказом, и выстроить с ними взаимоотношения, особенно если собираетесь работать вместе в течение достаточно длительного времени. При работе с фабрикой тестирования или поставщиком услуг для предприятия это и не обязательно, и не практично: тут важнее понимать, насколько полна применяемая ими методика и удобна ли вам модель привлечения вашей команды.

Существенно также, какого рода работу вы хотите поручить сторонним специалистам. Лично знать инженера или аналитика, который выполняет сканирование на уязвимости, не так важно. Но важно, чтобы вы доверяли консультанту, который проводит аудит вашего кода, чтобы вас устраивал его опыт и нравился стиль работы – чтобы он не оказывал негативного влияния на команду.

Наконец, не забывайте о пословице «мягко стелет, да жестко спать». Нам доводилось слышать о компаниях, которые на вступительной встрече представляют вам старших, самых опытных или хорошо известных консультантов, чтобы заключить контракт, а затем поручают работу младшим сотрудникам. Попросите перечислить имена тех, кто будет работать над вашим заказом, и – в зависимости от работы – определить способы связи с ними на время выполнения оценки (часто используются электронная почта, телефон, IRC или мессенджер Slack).

Оценка результатов оплаченной работы

Очень важно понимать, за что именно вы платите деньги.

Вы платите, чтобы кто-нибудь сертифицированный прогнал программу сканирования и представил «официальный» отчет? Или за высококвалифицированных экспертов, которые потратят дополнительное время, чтобы написать специальные инструменты и эксплойты для оценки именно вашей системы?

Гонка уступок

С ростом индустрии безопасности стало понятно, что без внешних игроков не обойтись. Тогда и ставки значительно выросли, что обусловило все более острую конкуренцию между поставщиками услуг.

Очень быстро цена стала определяющим фактором, что заставило поставщиков искать способы удешевления оценки. Чаще всего это делается путем замены квалифицированных специалистов автоматизированными инструментами. Там, где раньше по определению подразумевался наем частного умельца, поднаторевшего в искусстве компрометации систем, теперь все чаще речь идет о продаже высокоавтоматизированных служб, в которых нет места опыту и интуиции человека.

Мы не хотим сказать, что нанять квалифицированного хакера теперь невозможно. Мы просто предупреждаем, что вы должны заранее оценить рынок интересующих вас услуг и понять, что именно хотите получить. Как часто бывает в жизни, вы получаете то, за что платите, и если цена кажется слишком хорошей, чтобы быть правдой, то так оно, скорее всего, и есть.

Не тратьте чужое время попусту

Для оценки безопасности (если не считать вознаграждение за найденные ошибки и целенаправленное привлечение команды красных) чаще всего отводится ограниченное время. Позаботьтесь о том, чтобы не транжирить это время зря. Заранее подготовьте все необходимое для тестировщиков проникновения или аудиторов. Заведите для них учетные данные и предоставьте необходимые для работы права доступа. Пусть всегда будет доступен человек, готовый им помочь, ответить на вопросы, ознакомиться с результатами и вообще устранять с их пути все препятствия.

Проверка найденных проблем

Позаботьтесь о том, чтобы команда ясно понимала суть найденных проблем и способов их устранения. В каждом случае должно быть указано, в чем состоит проблема, каков уровень риска, где и как была обнаружена проблема, какие действия следует предпринять для ее воспроизведения и как ее исправить – все на понятном вам языке.

Если вы не понимаете, что вам предъявили, или не согласны с выводами, проведите испытание. Избавьтесь от ненужного шума и ложноположительных сигналов, разрешите все недопонимания (а недопонимания есть всегда). Убедитесь, что вы понимаете и согласны с уровнями риска,

чтобы можно было рассортировать проблемы и правильно назначить им приоритеты.

Но при этом дайте себе зарок никогда не впадать в тяжкий грех – не пытаться уговорить внешних оценщиков подправить отчет, изменить серьезность найденных проблем или вообще удалить их, чтобы вы и ваша команда выглядели не так плохо. Удивительно, как часто заказчики тестов просят переклассифицировать «серьезные» или «высокорисковые» проблемы, присвоив им уровень пониже, чтобы не показывать это начальству. Если вы оказались в такой ситуации, значит, у вас или у вашей организации проблемы более фундаментального характера. Цель внешней оценки в том и заключается, чтобы найти проблемы, а ваша задача – исправить их и учиться на собственных ошибках. Не поддавайтесь искушению подвести мину под этот процесс и лишиться той пользы, которую он может принести организации и клиентам.

Настаивайте на устраивающей вас форме результатов

Не нужно просто принимать отчет в форме PDF. Настаивайте, чтобы результаты были представлены в формате, который легко загрузить в диспетчер уязвимостей или в ваш журнал пожеланий с помощью функции загрузки данных в формате XML или воспользовавшись API взаимообмена между инструментами.

Интерпретируйте результаты в контексте

Оценщики часто представляют список уязвимостей, классифицированных согласно OWASP Top 10 или иной схеме, ориентированной на соответствие нормативным требованиям. Организуйте результаты, так чтобы с ними было удобно работать в контексте вашей организации, – чтобы было понятно, как расставить приоритеты и кто должен заниматься разрешением проблем.

Подключайте технических специалистов

К работе с внешними оценщиками следует привлекать не только группу безопасности. Нужно подключать и разработчиков, и эксплуатационников, и владельца продукта, чтобы все понимали процесс и относились к нему серьезно.

Измеряйте улучшение со временем

После того как найденная в результате оценки информация загружена в диспетчер уязвимостей или в программу управления проектом

типа Jira, можно строить отчеты об изменении показателей со временем, например:

- сколько уязвимостей обнаружено?
- насколько они серьезны?
- сколько времени требуется для их устранения?

Где находятся ваши риски? Эффективна ли оценка с точки зрения нахождения реальных проблем? Улучшается ли ваша программа обеспечения безопасности?

Имея исходную информацию, на эти вопросы нетрудно получить ответы.

Храните сведения о состоявшихся инспекциях и ретроспективном анализе и делитесь результатами

Используйте ретроспективный анализ, принятый в гибких методиках, или посмертный анализ в DevOps для обсуждения серьезных найденных проблем. Считайте, что это «почти попадание»: инцидент, который мог бы произойти, но, по счастью, был вовремя перехвачен.

В главе 13 мы увидим, что посмертный анализ может стать ценным инструментом для осмысления и разрешения проблем. Он способен усилить команду, сплотив ее для решения важных задач. Заодно это поможет сформировать чувство общего владения – ведь все члены команды видят, что проблемы устраняются.

Распределяйте устранение проблем между командами, чтобы способствовать передаче знаний

В крупных организациях важно, чтобы результаты были общим достоянием всех команд, это помогает учиться на чужом опыте. Ищите случаи, из которых могло бы извлечь урок несколько команд или несколько систем, тогда пользы от каждой оценки будет больше. Приглашайте членов других команд на посмертный анализ, чтобы они учились на найденных вами проблемах и высказывали свою точку зрения о возможных путях решения.

Время от времени ротируйте оценщиков или меняйте местами тестировщиков

Как мы уже говорили, одно из основных достоинств внешней инспекции – возможность узнавать о новых и разных взглядах на вещи и

учиться у специалистов, обладающих уникальным опытом. Поэтому полезно установить контакт с несколькими внешними инспекторами и время от времени ротировать их.

- В первый год придется потратить много времени и сил, чтобы выбрать компанию, заключить контракты, подписать соглашения о неразглашении и прочие бумаги, понять ее модель совместной работы, привыкнуть к тому, как ее сотрудники думают и разговаривают. А им, в свою очередь, нужно понять, как работают и разговаривают ваши люди.
- Во второй год дело идет лучше, поскольку готовить договоры стало быстрее и проще, и вам легче понять результаты и работать с ними.
- Когда всё становится слишком просто и предсказуемо, пора переключаться на другую компанию.

Понятно, что это может оказаться невозможно, если вы на стратегическом уровне связаны с партнером по оказанию услуг предприятию. Но тогда можно поискать другие способы изменить правила игры и извлечь из этого пользу, например попросить провести иные виды оценки.

Сухой остаток

Внешние инспекции могут отпугнуть, особенно если речь идет о наиболее опытных командах. Но если правильно выбрать поставщика и услуги и научиться извлекать из этих мероприятий максимум пользы, то они могут оказаться весьма ценными.

- Внешняя компания может предложить широкий спектр услуг: выбирайте те, что соответствуют конкретной системе и проекту.
- Внешние инспекции могут предложить объективность и специальные знания, которые трудно найти внутри организации.
- Для извлечения максимальной пользы из результатов тестирования нужно приложить усилия. Работайте с проводившей тестирование компанией, добиваясь полного понимания результатов и представления их в форме, допускающей загрузку в ваши конвейеры и системы.
- Подходите к выбору компании с умом и не забывайте, что это большой рынок: вы можете задавать вопросы и поискать альтернативы.
- Сначала воспользуйтесь более простыми и дешевыми видами оценки, например оценкой уязвимости, с целью выловить оче-

видные проблемы. И только потом переходите к более дорогим мероприятиям. Не платите дорогим консультантам за поиск проблем, которые можно найти за меньшие деньги, а то и собственными силами.

- Не пытайтесь сузить область исследования, заказывая тестирование или аудит, которые вы заведомо «проскочите». Предоставьте тестировщикам проникновения и другим инспекторам необходимый доступ и информацию, чтобы они могли проделать тщательную работу, дайте им шанс найти реальные проблемы.
- Вознаграждение за найденные ошибки может стать эффективным способом поиска уязвимостей в системе. Но помните, что подготовка и проведение такой программы – дорогостоящее и серьезное предприятие. Если вы думаете, что это просто способ задешево получить тестирование на проникновение, то вас ждут неприятные сюрпризы. Эффективная программа вознаграждения не может быть дешевой, а запустив ее, пойти на попятную будет очень трудно.

  Программа вознаграждения за найденные ошибки – это больше про взаимодействие с сообществом, чем про тестирование. Будьте готовы обходиться со всеми корреспондентами вежливо и терпеливо, содержательно отвечать на их сообщения об ошибках и выплачивать справедливое вознаграждение. В противном случае вы можете нанести ущерб марке своей организации, стать причиной перебоев в ее работе и даже спровоцировать злоумышленные действия. Кто-то в организации должен тратить массу времени на просмотр и осмысление всего найденного и зачастую еще больше времени, чтобы вежливо объяснить человеку, который тоже потратил свое время, чтобы найти, воспроизвести и описать проблему, почему то, что он нашел (и за что хочет получить деньги), на самом деле не является дефектом, по крайней мере дефектом, который вы собираетесь исправлять.
- Если вы не собираетесь использовать внешние инспекции, чтобы учиться у экспертов, то только зря тратите их время и свои деньги.

Глава 13

Эксплуатация и безопасность

Гибкие команды часто упираются в стену эксплуатации, поскольку поставляют изменения гораздо быстрее, чем эксплуатационники могут их переварить. Но стена между разработчиками и эксплуатационниками начинает разрушаться по мере того, как функции эксплуатации передаются в облако, а сами группы эксплуатации претерпевают метаморфозу, ведущую к гибкости.

Именно в этом и состоит цель движения DevOps: применить идеи, ценности, инженерные практики и инструменты, заимствованные у гибких методик, к эксплуатации и постараться сблизить разработчиков и эксплуатационников. Такие команды разработчиков, вместо того чтобы сбросить готовую работу эксплуатационникам и техникам и перейти к следующему проекту, делят ответственность за передачу системы в эксплуатацию и надзор за ее работой в течение всего срока жизни системы.

Разработчики напрямую участвуют в подготовке, настройке и развертывании исполняющей среды, мониторинге и реагировании на инциденты и в других действиях, обычно возлагаемых на службу эксплуатации. По мере перевода инфраструктуры в форму кода эксплуатационники начинают использовать те же инженерные практики и инструменты, что и разработчики, учатся применять рефакторинг, разработку через тестирование и непрерывную интеграцию. Работодатели все чаще предлагают работу в гибридной роли, например на должность инженера по надежности сайта (site reliability engineers – SRE), названную так вслед за Google: это инженеры, в равной мере владеющие навыками эксплуатации и разработки.

Но вне зависимости от того, работают программисты в составе команды DevOps или нет, они должны понимать, как правильно и безопасно развернуть и настроить свою систему, как система будет работать в условиях реальной нагрузки и в случае отказа и какая информация

и инструменты нужны эксплуатационникам, чтобы эффективно эксплуатировать систему, вести мониторинг ее работы, а также находить и устранять неисправности. И они должны делать все возможное, чтобы избежать трений с эксплуатационниками и наладить обратную связь.

В этой главе мы рассмотрим естественные пересечения между разработкой, эксплуатацией и безопасностью, а также вопрос о том, как гибкая разработка порождает новые вызовы – и возможности – для эксплуатации и безопасности.

Укрепление системы: настройка безопасных систем

В фокусе внимания публичных дискуссий о безопасности находится в основном обнаружение в приложениях дефектов и изъянов, которые можно разными хитрыми способами использовать к собственной выгоде, но на самом деле для безопасности не меньшую роль играет правильная конфигурация и настройка рабочей среды. Спроектировать и написать безопасное приложение важно, но недостаточно.

На одном из первых мест для тестировщика проникновения – и противника – стоит описание среды, позволяющее понять, какие существуют возможности для атаки. Комбинация систем, приложений, их взаимосвязей и людей, которые ими пользуются, часто называется *поверхностью атаки*. Именно об этих частях нужно знать, чтобы проложить потенциальные маршруты к компрометации.

Важно отметить, что, с точки зрения атакующего, люди так же важны и потенциально уязвимы, как технологии. Поэтому, с точки зрения обороняющегося, при укреплении системы нужно ясно понимать, как эта система используется людьми в реальности, а не в идеализированном сценарии.

Неправильно настроенные брандмауэры, оставленные открытыми порты, административные пароли по умолчанию, устаревшие операционные системы, пакеты программ с известными уязвимостями, пароли, оставленные для всеобщего обозрения на внутренней вики-странице или записанные в репозиторий Git, – эти и прочие широко распространенные ошибки, которых можно было бы легко избежать, открывают возможности проникнуть в среду и тем облегчить задачу атакующего.

Многие инструменты разработчиков, упоминаемые в этой книге, спроектированы так, чтобы начать работу с ними можно было сразу же, без лишних церемоний. Но, как мы увидим в этой главе, по мере того

как команда все сильнее проникается идеологией гибкой разработки и начинает развертывать код чаще, наращивая уровень автоматизации, это создает серьезные риски безопасности.

Исполняющие среды ничуть не лучше. Базы данных, веб-серверы и даже операционные системы и средства обеспечения безопасности тоже упаковывают так, чтобы их было просто запустить с параметрами по умолчанию. Противник об этом знает и понимает, как этим воспользоваться себе во благо.

Это означает, что, после того как система подготовлена, необходимые пакеты установлены и все наконец заработало правильно, нужно выполнить дополнительные шаги, чтобы укрепить систему на случай атаки. Это относится прежде всего к производственным системам, но, как мы увидим, распространяется даже на среды сборки и тестирования. И конечно, любая система, содержащая компоненты с выходом в Интернет, нуждается в еще более придирчивом внимании.

Смысл укрепления системы – в том, чтобы сократить поверхность атаки. Джастин Шу (Justin Schuh) из Google кратко сформулировал это в своем Твиттере: «Главный смысл безопасности – свести к минимуму поверхность атаки. 90% вашей работы состоит именно в этом. Остальные 10% – удача».

Перечислим некоторые типичные методы укрепления:

- заблокировать или удалить учетные записи, существующие по умолчанию, и изменить учетные данные по умолчанию;
- создать эффективное разделение ролей между принципалами-людьми и системой (т. е. не работать от имени root!);
- удалить неиспользуемые программные пакеты и отключить в скриптах автозапуска демоны, ненужные системной роли;
- выключить службы и закрыть все сетевые порты, кроме абсолютно необходимых;
- проверить, что оставленные пакеты актуальны и для них установлены все имеющиеся исправления;
- ограничить права доступа к файлам и каталогам;
- правильно настроить аудит и уровни протоколирования;
- включить правила брандмауэра на уровне хоста в качестве меры эшелонированной обороны;
- включить мониторинг целостности файлов, например программу OSSEC или Tripwire, чтобы вовремя обнаруживать неавторизованное изменение программ, конфигурационных файлов и файлов данных;

- проверить, что включены и правильно работают встроенные платформенные механизмы безопасности, например шифрование всего диска или SIP (System Integrity Protection – защита целостности системы) (*https://en.wikipedia.org/wiki/System_Integrity_Protection*) в macOS либо ALSR/DEP в Windows.

Ниже мы увидим, что это не полный список, а лишь отправная точка.

Краткое введение в укрепление сервера Linux можно найти в книге Lee Brotherston, Amanda Berlin «The Defensive Security Handbook» (издательство O'Reilly).

Аналогичные действия следует проделать для всех уровней, от которых зависит ваше приложение: сеть, ОС, виртуальные машины, контейнеры, базы данных, веб-серверы и т. д.

Детальному обсуждению каждого из этих вопросов посвящены целые книги, и все это находится в постоянном движении, поскольку программное обеспечение и системы изменяются, добавляются новые функции. Знакомство с передовыми практиками укрепления технологий, составляющих исполняющую среду, – один из ключевых аспектов эксплуатационной безопасности в целом, эту тему следует изучать настолько глубоко, насколько вы можете себе позволить.

Нормативно-правовые требования к укреплению

В некоторых нормативно-правовых документах, например в стандарте PCI DSS, сформулированы конкретные требования к укреплению систем. В требовании 2.2 PCI DSS указано, что в организации должны быть документированные политики конфигурирования системы, включающие следующие процедуры:

- изменение заводских параметров и удаление ненужных учетных записей, созданных по умолчанию;
- изоляция функций каждого сервера (физическая, с помощью разделения на виртуальные машины и (или) контейнеры), чтобы на одном сервере не работали одновременно приложения или службы с разными требованиями к безопасности;

- запуск только тех процессов и служб, которые необходимы, и удаление ненужных скриптов, драйверов, подсистем и файловых систем;
- настройка системных параметров с целью предотвратить некорректное использование.

Понимание того, какие конкретно требования к укреплению включены в нормативно-правовые документы, которым должна соответствовать организация, очевидно, является ключом к соблюдению соответствия, и вы будете обязаны продемонстрировать это аудиторам.

Стандарты и рекомендации, относящиеся к укреплению

Помимо нормативно-правовых документов, для различных платформ и технологий существуют стандарты и рекомендации по укреплению, в которых объясняется, что и как делать. Отметим некоторые из наиболее подробных рекомендаций:

- Руководства по технической реализации безопасности Министерства обороны США (US DoD STIGs – Security Technical Implementation Guides) (*http://iase.disa.mil/stigs/Pages/index.aspx*) для укрепления платформ, используемых в проектах МО США;
- Стандарты конфигурации в правительстве США (United States Government Configuration Baseline – USGCB) (*https://usgcb.nist.gov/*);
- Эталонные правила Центра интернет-безопасности (Center for Internet Security – CIS) (*https://benchmarks.cisecurity.org/*).

Существуют еще руководства по укреплению конкретных продуктов, публикуемые производителями, в т. ч. Red Hat, Cisco, Oracle и прочими, а также руководства, написанные практическими специалистами по различным аспектам безопасности и доступные бесплатно.

Центр интернет-безопасности – это межотраслевая организация, пропагандирующая передовые практики обеспечения безопасности систем. Он публикует документ «Critical Controls» (*https://www.cisecurity.org/critical-controls.cfm*) – перечень 20 основных рекомендаций по обеспечению информационной безопасности организации, а также эталонные правила CIS – набор единодушно одобренных рекомендаций по укреплению ОС Unix/Linux и Windows, мобильных устройств, сетевых устройств, облачных платформ и распространенных программных пакетов. Эти рекомендации построены с учетом требований, изложенных в различных нормативно-справочных документах.

Контрольные списки CIS распространяются бесплатно в формате PDF. Одно руководство может насчитывать несколько сотен страниц, на ко-

торых подробно объясняется, что делать и почему. Для членов CIS они доступны также в формате XML, чтобы их можно было использовать совместно с автоматизированными инструментами, поддерживающими спецификацию SCAP XCCDF (*https://scap.nist.gov/specifications/xccdf/*).

Руководства по укреплению типа спецификаций CIS – это, скорее, идеал, к которому нужно стремиться, а не контрольные списки, которые следует в точности выполнять. Необязательно реализовывать все описанные в них действия, да это и не всегда возможно, потому что после удаления пакета, ограничения доступа к файлам или отзыва полномочий пользователя какие-то программы могут перестать работать. Но, имея подобное руководство, вы, по крайней мере, понимаете, чем рискуете, идя на компромисс.

Проблемы, возникающие при укреплении

Укрепление системы – не такая работа, которую раз выполнил и забыл. Требования к укреплению постоянно изменяются, поскольку выпускаются новые версии программ, содержащие новые функции, и одновременно обнаруживаются новые виды атак и выпускаются новые или обновляются старые нормативно-правовые документы.

Поскольку многие требования выпускаются по инициативе правительства и регулирующих органов, которые неизбежно обрастают всяческими комитетами и экспертными комиссиями, то процесс публикации утвержденных рекомендаций бюрократический, непрозрачный и очень медленный. Могут уйти месяцы или годы, чтобы достигнуть соглашения о том, что включать и что не включать в спецификации укрепления, а затем опубликовать их в виде, который можно было бы использовать. Неприглядная реальность заключается в том, что многие официальные руководства такого рода относятся к версиям ПО, которое уже устарело и содержит известные уязвимости.

По такой вот печальной иронии, вы можете найти ясные и утвержденные инструкции о том, как укрепить ПО, давно уже замененное чем-то более новым, что могло бы быть безопаснее в эксплуатации, если бы вы только знали, как его правильно настроить. Это значит, что часто вам придется укреплять новое ПО, полагаясь на собственные силы и следуя основополагающему принципу – сократить поверхность атаки.

Как уже отмечалось, многие укрепительные действия могут приводить к сбоям – от ошибок во время выполнения при определенных условиях до полной невозможности запустить приложение. Изменения

следует вносить итеративно и небольшими шагами, проверяя каждый раз, не сломалось ли что-то.

Поиск компромисса между укреплением и функциональностью является скорее искусством, чем наукой. Ситуация осложняется еще и тем, что унаследованные системы уже не поддерживаются поставщиком или автором.

Но даже при использовании совсем новых систем, поддерживаемых поставщиком, в процессе укрепления вы можете оказаться в полном одиночестве. Например, одному из нас недавно довелось укреплять корпоративную систему видеоконференций. Потребовалось бесчисленное количество писем и телефонных звонков, чтобы вытянуть из поставщика достоверную информацию о назначении различных сетевых портов, которые, согласно документации, должны были оставаться открытыми. В итоге получилось, что клиент должен был объяснять поставщику, какие порты из длинного перечня, приведенного в документации, действительно необходимы для работы. Увы, такую ситуацию не назовешь из ряда вон выходящей, поэтому будьте готовы заняться исследованиями, чтобы сократить поверхность атаки до минимума, даже имея дело с коммерческим продуктом.

Осложняет укрепление и размытость границы между операционной системой и онлайновыми службами. Сегодня все основные операционные системы общего назначения содержат функции, которым для работы необходимы онлайновые службы и API. В зависимости от среды такой перенос ОС в облако может вызвать серьезную обеспокоенность по поводу безопасности и конфиденциальности. А определить границы доверия вокруг оконечных точек становится все труднее.

Как это ни печально, обновление системы и приложений также может подорвать все ваши усилия по укреплению, поскольку после некоторых обновлений конфигурационные параметры принимают новые значения или возвращаются к значениям по умолчанию. Поэтому стратегия укрепления должна предусматривать тестирование и квалификацию всех обновлений, применяемых к укрепленным системам.

Укрепление требует обширных технических познаний и внимания к деталям, поэтому стоит дорого, и провести его правильно нелегко. Но именно детали имеют значение. Противник может воспользоваться мелкими ошибками в конфигурации, чтобы проникнуть в систему, а автоматизированные сканеры (входящие в арсенал любого атакующего) многие из таких ошибок находят без труда. Необходимо тщательно следить за тем, чтобы не упустить что-то важное.

Автоматизированное сканирование на соответствие нормативным требованиям

Большинство из нас давно уже уяснило, что невозможно все сделать вручную: создателям систем, как и атакующим, нужны хорошие инструменты.

Сканирование на соответствие нормативным требованиям и сканирование на уязвимость – разные вещи. Сканеры соответствия проверяют соблюдение предопределенных правил и рекомендаций (правильные практики). Сканеры уязвимостей ищут известные уязвимости, например учетные данные по умолчанию и отсутствующие исправления (неправильные практики). Разумеется, имеет место пересечение, потому что наличие известных уязвимостей – результат деятельности людей, не соблюдающих правильные практики.

Сканеры типа Nessus и Nexpose ищут уязвимости, а также проверяют соблюдение определенных рекомендаций, применяя различные политики или подключаемые модули. Для Red Hat Linux и только для этой ОС OpenSCAP ищет как нарушения нормативных требований, так и уязвимости.

Другие сканеры, например из проекта OpenVAS, ищут только уязвимости. OpenVAS – результат разветвления Nessus, произошедшего более 10 лет назад, когда Nessus стал коммерческим проектом. Он проверяет систему на тысячи известных уязвимостей и эксплойтов из своей базы данных. Правда, этот сканер заработал неважную репутацию, поскольку часто поднимает ложную тревогу, так что будьте готовы проверять результаты, а не принимать их за истину в последней инстанции.

Существует несколько автоматизированных инструментов аудита, которые умеют сканировать конфигурацию инфраструктуры и сообщать, если она не отвечает рекомендациям по укреплению.

- Члены CIS могут скачать и использовать инструмент CIS-CAT, который проверяет системы на соответствие эталонным правилам CIS.
- OpenSCAP (*https://www.open-scap.org/*) проверяет определенные дистрибутивы Linux и другое ПО на соблюдение политик укрепления, основанных на рекомендациях PCI DSS, STIG и USGCB, а также может автоматически исправить найденные недочеты.

- Lynis (*https://cisofy.com/lynis/*) – сканер с открытым исходным кодом для систем Linux и Unix, который проверяет конфигурацию на соответствие спецификациям укрепления CIS, NIST и NSA, а также рекомендациям конкретных поставщиков и общепринятым передовым практикам.
- В Интернете можно найти свободные средства проверки конкретных систем, например osx-config-check (*https://github.com/kristovatlas/osx-config-check*) и Secure-Host-Baseline (*https://github.com/iadgov/Secure-Host-Baseline*).
- Для нескольких коммерческих сканеров уязвимостей, например Nessus, Nexpose и Qualys, имеются модули проверки на соответствие нормативным требованиям, проверяющие соблюдение эталонных правил CIS для различных ОС, баз данных и сетевого оборудования.

Если вы не производите сканирование своих систем регулярно, в составе конвейеров сборки и развертывания, и не устраняете проблемы сразу, как только их обнаружил сканер, значит, ваши системы небезопасны.

Подходы к построению укрепленных систем

Для создания изначально укрепленных конфигураций системы также можно применить автоматизацию. Существует две стратегии.

Золотой образ

Подготовить базовый укрепленный шаблон, или «золотой образ», на основе которого будут затем создаваться системы. Скачайте стандартный дистрибутив операционной системы, установите его на «голую» машину, поставьте необходимые пакеты и исправления, а затем выполните все шаги укрепления, какие считаете нужными. Протестируйте этот образ и используйте в производственных системах.

Такие конфигурации рабочей системы должны считаться постоянными: после установки их уже нельзя изменять. Если необходимо применить обновления или исправления либо изменить конфигурацию исполняющей среды, то следует создать новый базовый образ и на его основе перестроить все рабочие машины заново. В Amazon и Netflix управление развертыванием производится именно так, поскольку гарантирует, что даже при таком гигантском масштабе контроль не будет утрачен.

Для подготовки образов машин можно использовать такие инструменты, как Aminator (*https://github.com/Netflix/aminator*) от Netflix или Packer (*https://www.packer.io/*) от HashiCorp. Packer позволяет сконфигурировать один системный образ и использовать один и тот же шаблон на нескольких платформах: Docker, VMware и облачные платформы типа EC2, Azure или Google Compute Engine. Таким образом, вы можете разместить среды разработки, тестирования и эксплуатации на разных платформах, и при этом будет гарантирована полная синхронизация.

Автоматизированное управление конфигурацией

Взять минимальный образ ОС, загрузить с его помощью каждое устройство, а затем построить исполняющую среду, следуя шагам, запрограммированным в средстве управления конфигурацией: Ansible, Chef, Puppet, SaltStack и т. п. Инструкции по установке пакетов и применению исправлений, конфигурированию исполняющей среды, включая укрепление, записываются в репозиторий наряду со всем остальным кодом. Их можно протестировать перед применением.

Большинство таких инструментов автоматически синхронизирует конфигурации всех управляемых систем в соответствии с записанными правилами: примерно каждые 30 минут производится сравнение всех параметров, все отклонения фиксируются в отчете и автоматически исправляются.

Однако этот подход надежен, только если все изменения конфигурации прописаны в коде и вносятся единообразно. Если программист или администратор базы данных самостоятельно, а не под контролем инструмента управления конфигурацией вносит изменения в конфигурационные файлы, пакеты или учетные записи пользователей, то такие изменения не будут учтены, и синхронизация на них не распространяется. Со временем конфигурации начинают «разъезжаться», в результате чего несогласованности и уязвимости производственных систем могут остаться незамеченными и неисправленными.

Оба подхода делают процесс подготовки и конфигурирования системы быстрее, надежнее и прозрачнее. Оба позволяют реагировать на проблемы, быстро и уверенно применяя исправления. А при необходимости скомпрометированную инфраструктуру можно разобрать, вычистить и полностью перестроить.

Автоматизированные шаблоны укрепления

С помощью современных инструментов управления конфигурацией, например Chef и Puppet, можно оформить рекомендации по укреплению прямо в виде кода.

Один из лучших примеров – DevSec (*https://github.com/dev-sec*), комплект шаблонов укрепления с открытым исходным кодом, первоначально разработанный в компании Deutsche Telekom, а теперь поддерживаемый добровольцами из многих организаций.

В этом каркасе реализованы практические шаги по укреплению распространенных дистрибутивов Linux и таких компонентов исполняющей среды, как ssh, Apache, nginx, mysql и Postgres. Предоставляется полный комплект шаблонов укрепления для Chef и Puppet, а также несколько сценариев для Ansible. Все шаблоны состоят из конфигурируемых правил, которые вы можете расширить или модифицировать по собственному усмотрению.

Правила укрепления основаны на общепризнанных передовых практиках, включая внутренние стандарты Deutsche Telekom, руководство по укреплению прикладной криптографии на сайте BetterCrypto.org (*https://bettercrypto.org/static/applied-crypto-hardening.pdf*), в котором объясняется, как безопасно пользоваться криптографическими средствами, а также другие руководства по укреплению.

Такие спецификации являются самодокументированными (по крайней мере, для технически подкованных специалистов) и тестопригодными. Чтобы проверить правильность применения правил, можно воспользоваться такими инструментами автоматизированного тестирования, как Serverspec (см. главу 12).

В состав DevSec входят автоматизированные тесты соответствия нормативным требованиям, написанные на предметно-ориентированном языке InSpec (*https://github.com/chef/inspec*), который сам написан на Ruby. Эти тесты проверяют, что правила укрепления для Chef, Puppet и Ansible следуют одним и тем же рекомендациям. В проект DevSec включен также написанный на InSpec профиль соответствия для эталонных правил CIS для Docker (*https://github.com/dev-sec/cis-docker-benchmark*) и эталонного теста SSL (*https://github.com/dev-sec/ssl-benchmark*).

InSpec работает, как Serverspec, – сравнивает конфигурацию машины с ожидаемой и выдает ошибку в случае несовпадения. Его можно использовать для управляемой тестами проверки соответствия нормативным требованиям. Для этого нужно написать утверждения, справедливость которых зависит от того, выполнены ли шаги укрепления,

а затем использовать эти тесты для проверки того, что новые системы сконфигурированы правильно.

Тесты InSpec специально проектируются так, чтобы их могли использовать и инженеры, и аудиторы соответствия нормативным требованиям. Они пишутся на обычном английском языке, и каждый сценарий можно снабдить аннотацией, чтобы его назначение было понятно аудитору. Для каждого правила тестирования можно определить серьезность, или уровень риска, и добавить описания, сопоставимые с контрольными списками проверки соответствия или нормативно-правовыми требованиями. Такой подход позволяет изучить весь код тестов вместе с аудитором, а затем продемонстрировать результаты его выполнения.

Вот два примера тестов соответствия, взятых из InSpec GitHub:

```
Only accept requests on secure ports - This test ensures that a web server is
only listening on well-secured ports.

describe port(80) do
  it { should_not be_listening }
end

describe port(443) do
  it { should be_listening }
  its('protocols') {should include 'tcp'}
end

Use approved strong ciphers - This test ensures that only
enterprise-compliant ciphers are used for SSH servers.

describe sshd_config do
  its('Ciphers') ↵
  { should eq('chacha20-poly1305@openssh.com,aes256-ctr,aes192-ctr,aes128-ctr') }
end
```

InSpec также лежит в основе продукта Compliance Server, входящего в состав Chef. Разработчики Chef работают над переводом автоматизированных профилей укрепления типа CIS с языка SCAP XML на InSpec, чтобы можно было автоматически проверять всю инфраструктуру на соблюдение этих правил.

Сеть как код

При традиционных подходах к разработке систем безопасность сети и приложений обеспечивается независимо. Разработчикам иногда необходимо открыть какой-то порт на брандмауэре или изменить пра-

вило маршрутизации. Возможно, они должны знать, как работать с прокси-серверами в демилитаризованной зоне. Но в остальном эти два мира почти не соприкасаются.

Все меняется, когда приложения перемещаются в облако, где разработчикам предоставлено больше сетевых правил и средств управления и где разработчики больше не могут полагаться на то, что средства охраны периметра, например брандмауэры, защитят их приложения.

Сетевые устройства – брандмауэры, системы обнаружения и предотвращения вторжений, маршрутизаторы и переключатели – по-прежнему настраиваются и конфигурируются (и укрепляются) вручную с помощью специально написанных скриптов или консолей устройств. Но за последние несколько лет производители сетевого оборудования добавили REST API и другие программируемые интерфейсы для поддержки программно определяемых сетей, а в инструменты типа Ansible, Chef и Puppet была добавлена поддержка программного конфигурирования сетевых устройств, включая коммутаторы и брандмауэры таких производителей, как Cisco Systems, Juniper Networks, F5 Networks и Arista Networks.

Благодаря всему этому инструментарию идея «сети как кода» изменила мир так же, как идея «инфраструктуры как кода» в отношении управления серверами, системами хранения и облачными платформами. И мы имеем те же преимущества:

- конфигурациям сетевых устройств можно присваивать номера версий и хранить их в системе управления исходным кодом, предоставляя контрольный журнал для управления изменениями и криминалистической экспертизы;
- правила конфигурирования сетевых устройств можно записывать с помощью языков высокого уровня и стандартных шаблонов, что делает внесение изменений проще и прозрачнее;
- изменения конфигурации сети можно инспектировать и тестировать заранее, применяя описанные выше инструменты и приемы, а не полагаться на команды ping и traceroute постфактум;
- изменения можно автоматически применять к различным устройствам, чтобы обеспечить согласованность;
- изменения конфигурации сети можно координировать с изменениями приложений и прочими инфраструктурными изменениями и развертывать совместно, что уменьшает риски и несовместимости.

Во многих организациях группа эксплуатации сети – это отдельное подразделение, возможно, появившееся раньше, чем группа эксплуатации, а это значит, что внедрение описанных выше идей потребует таких же культурных и технических изменений, которые имеют место при внедрении новых методов управления серверной инфраструктурой в командах, переходящих на DevOps. Но в награду конфигурирование и управление сетью станут более открытыми для сотрудничества, проще для понимания и безопаснее для изменения. И все это хорошо с точки зрения безопасности.

Мониторинг и обнаружение вторжений

В современной жизни никуда не деться от того, что стоит подключить любой компьютер или сетевое устройство к Интернету, как его порты сразу же начнут сканировать. Системы обнаружения должны отличать фоновый шум, связанный с попытками ботов просканировать периметр системы, от успешного взлома, а также различать обычное поведение и атаку – на что это похоже: на медленное и постепенное просачивание данных или на периодическое обновление биржевого тикера?

В крупных организациях мониторингом безопасности обычно занимается специальный центр безопасной эксплуатации (security operations center – SOC), укомплектованный аналитиками, которые просеивают сведения об атаках, предупреждения о вторжении и описания новых угроз. В организациях поменьше мониторинг безопасности иногда поручают стороннему поставщику услуг управления безопасностью (managed security services provider – MSSP) типа SecureWorks или Symantec, а иногда и вовсе ничего не делают. Такой вид мониторинга в основном осуществляется на сетевом уровне, т. е. сетевой трафик обследуется на наличие известных сигнатур и необычных паттернов.

Инженеры-эксплуатационники наблюдают за серверами, сетями, системами хранения, базами данных и работающими приложениями с помощью таких инструментов, как Nagios, или служб типа Zabbix или New Relic, которые проверяют состояние системы и ищут признаки замедления или проблем в ходе работы: аппаратные сбои, переполнение диска, аварийные остановки служб.

В ходе мониторинга приложений проверяется состояние приложения и собираются данные о его работе. Сколько сегодня было выполнено транзакций? Откуда поступает большая часть трафика? Сколько денег мы сегодня заработали? Сколько новых пользователей заходило в нашу систему? И так далее.

Мониторинг с целью организации обратной связи

Команды, практикующие гибкие методики, а особенно DevOps и бережливые технологии, применяют информацию о работе системы и ее использовании для принятия основанных на данных проектных решений о том, какие функции наиболее востребованы и в каких частях системы наблюдаются проблемы с надежностью или качеством. Они получают обратную связь от процессов тестирования и эксплуатации, показывающую, на что обратить внимание и что улучшить. Это существенно отличается от многих традиционных подходов к безопасности, когда цели безопасности определяются заранее и доводятся до персонала в виде приказов, подлежащих неукоснительному выполнению.

Команды DevOps могут быть помешаны на показателях так же, как гибкие команды помешаны на тестах. Технологии мониторинга, например комбинация statsd+carbon+graphite, применяемая в Etsy (*https://github.com/etsy/statsd*), Prometheus (*https://prometheus.io/*), Graylog (*https://www.graylog.org/*), Elastic Stack (*https://github.com/elastic*), или облачные платформы мониторинга типа Datadog или Stackify позволяют с относительно малыми затратами создавать, собирать, коррелировать, визуализировать и анализировать показатели работы.

Компании-лидеры в применении DevOps – Etsy, Netflix и PayPal – отслеживают тысячи показателей работы своих систем и собирают сотни тысяч их значений ежесекундно. Если кто-то думает, что некоторая информация может представлять интерес, он превращает ее в показатель. Затем этот показатель отслеживается и представляется в виде графика, чтобы понять, выделяется ли он чем-то на фоне шума. В Etsy говорят: «Если что-то двигается, построй график», эта компания впервые призналась, что «приносит дары на алтарь графика».

Использование мониторинга приложений в интересах безопасности

Те же инструменты и паттерны можно использовать, чтобы обнаруживать атаки и лучше понять, как защитить систему.

Появление во время выполнения ошибок HTTP из диапазонов 400/500 может быть признаком ошибок сети или других эксплуатационных проблем – или признаком того, что противник пытается изучить приложение, чтобы взломать его. Ошибки SQL на уровне базы данных могут быть вызваны дефектами в программе или проблемами самой базы, но также и попытками внедрения SQL. Нехватка памяти может быть обусловлена неправильной конфигурацией или ошибкой в про-

грамме, но также и атакой с переполнением буфера. Ошибка при входе в систему – либо результат забывчивости пользователя, либо признак бота, перебирающего пароли.

Разработчики могут помочь группам эксплуатации и безопасности, обеспечив возможность заглянуть внутрь приложений, т. е. добавить датчики, которые позволяют получить и проанализировать интересующую эксплуатационников и безопасников информацию, а затем предпринять необходимые действия.

Разработчики могут также помочь в установлении эталона «нормального» поведения системы. Поскольку они хорошо понимают правила приложения и то, как его предполагалось использовать, то могут заметить аномалии там, где другие увидят лишь шум в журналах или на графиках. Разработчик может подсказать, как установить пороговые значения для уведомлений. Он подпрыгнет, увидев «невозможное» исключение, поскольку сам писал утверждение, заверяющее, что такого не может быть. Например, если некоторая проверка на стороне сервера не проходит и разработчик знает, что точно такая же проверка реализована на стороне клиента, то вывод может быть только один: кто-то атакует систему, применяя перехватывающий прокси.

Средства уведомления и аналитики

Существует ряд «крутых» каркасов и инструментов уведомления, с помощью которых можно реализовать уведомления о работе и безопасности приложения в режиме реального времени.

Если для мониторинга вы используете комбинацию Elasticsearch-Logstash-Kibana (ELK), то рекомендуем поинтересоваться программой Elastalert компании Yelp, построенной поверх Elastic Search.

Система управления уведомлениями Etsy 411 также основана на ELK, она позволяет настроить правила уведомления и процессы обработки уведомлений.

Компания AirBnB разработала систему StreamAlert с открытым исходным кодом, комплексный бессерверный каркас анализа данных в реальном времени.

Эту информацию можно использовать не только для обнаружения противника – будем надеяться, до того как он зашел слишком далеко, – но и для упорядочения по важности работ, касающихся безопасности. Зейн Лэки (Zane Lackey), работающий в компании Signal Sciences

и бывший глава группы обеспечения безопасности в Etsy, называет это *защитой в ответ на атаку*. Наблюдайте за тем, что делает противник, изучайте методы его работы и идентифицируйте предпринимаемые им атаки. Затем примите меры для защиты своей системы от этих атак. Уязвимость, найденная сканером кода или в процессе моделирования угроз, может быть важна. Но уязвимость, которую противник активно пытается эксплуатировать в рабочей системе, – и вы это видите – является критической. Тут надо немедленно бросить все силы на тестирование и исправление.

Системы обнаружения вторжений следует устанавливать в изолированных сетях и устройствах, иначе в случае успешной компрометации машины противник сможет отключить саму систему обнаружения (хотя хорошая система безопасности заметит, что машина прекратила проверку). Особенно хороши в этом отношении средства прослушивания сетей и параллельного мониторинга.

Следует также с самого начала осознать, что использование внутри системы протокола SSL/TLS затрудняет перехват трафика не только противнику, но и вашей собственной группе безопасности и эксплуатации сети, которая пытается делать то же самое – правда, с другими целями. Если вы переходите от сети, не защищенной SSL, к сети, где на внутренних линиях связи применяется шифрование, то разработайте соответствующую новую архитектуру, в которой предусмотрены места, нуждающиеся в мониторинге.

Чтобы защититься от инсайдерских атак, настройте инструменты и службы мониторинга, так чтобы они администрировались отдельно от производственных систем, чтобы администраторы локальных систем не могли ни вмешаться в работу устройств или политик безопасности, ни отключить протоколирование. Следует обращать особое внимание на сотрудников, которые могут переходить из группы безопасности в группу администрирования и обратно.

Аудит и протоколирование

Аудит и протоколирование незаменимы для мониторинга безопасности и важны также для проверки соответствия нормативным требованиям, выявления мошенничества, неотрицаемости (противодействия попыткам пользователя отрицать ответственность за некоторое действие) и обвинения в атаке (привлечения атакующего к суду).

Аудит – это сохранение информации о действиях, произведенных системными принципалами: что было сделано, когда и в каком порядке.

Данные аудита предназначены для анализа безопасности и соответствия нормативным требованиям, а также для расследования случаев мошенничества. Эти данные должны быть полны, чтобы их можно было использовать в качестве доказательства, поэтому код аудита следует проверить на отсутствие пробелов в покрытии, а собранные записи должны быть пронумерованы, чтобы легко можно было обнаружить отсутствующие.

При проектировании системы аудита следует соблюдать баланс между требованием сохранять достаточно информации, чтобы можно было предъявить ясный и полный контрольный журнал, и транзакционными издержками и риском вызвать перегрузку систем хранения и анализа данных аудита, особенно если вы подаете информацию на вход SIEM или другой платформы анализа данных безопасности.

Но руководящий принцип остается в силе: если можете себе позволить, попытайтесь аудировать все, что делает пользователь: каждую команду или транзакцию и особенно все действия администраторов и эксплуатационников. Учитывая, что стоимость хранения неуклонно снижается, решение о том, сколько данных аудита хранить, следует основывать на требованиях регуляторов к конфиденциальности и срокам хранения данных, а не на бюджетных ограничениях.

В распределенных системах, например в микросервисных архитектурах, журналы аудита от разных систем в идеале должны объединяться, чтобы аудиторам было проще свести информацию из разных источников в единое связное целое.

Протоколирование на уровне приложения – более общий механизм. С точки зрения группы эксплуатации, протоколирование дает возможность следить за тем, что происходит в системе, и обнаруживать ошибки и исключения. Для разработчика протоколирование – подспорье в диагностике и отладке ошибок и последняя надежда, когда что-то ломается или кто-то задает вопрос: «Как это случилось?» Журналы могут также быть источником информации для бизнес-аналитики и для систем оповещения. А для группы безопасности журналы неоценимы, когда возникает необходимость в мониторинге событий, реагировании на инциденты или криминалистической экспертизе.

При проектировании системы протоколирования одно из важных решений – для чего предназначен журнал: для чтения людьми или разбора программой. Что важнее: понятность человеку и, как следствие, многословность или более эффективная структура, благодаря которой данные будет проще интерпретировать инструментам оповещения, системе обнаружения вторжений и информационным панелям?

При любом подходе следите за тем, чтобы протоколирование выполнялось единообразно в пределах одной системы и, по возможности, одинаково во всех системах. Состав полей в каждой записи должен быть одним и тем же, должна быть отражена, по крайней мере, следующая информация:

- кто (идентификатор пользователя, IP-адрес источника, идентификатор запроса);
- где (узел, идентификатор и версия службы – информация о версии особенно важна, когда код часто изменяется);
- тип события (INFO, WARN, ERROR, SECURITY);
- когда (синхронизированное время).

Команды должны согласовать, какая библиотека протоколирования будет использоваться по умолчанию, например Apache Logging Services (*https://logging.apache.org/*), куда входят расширяемые библиотеки для Java, .NET, C++ и PHP.

Как уже отмечалось в главе 5, некоторые нормативно-правовые документы, в т. ч. PCI DSS, четко определяют, какую информацию следует протоколировать, а какую – не следует и в течение какого времени хранить журналы.

Как минимум, необходимо протоколировать:

- события запуска и остановки системы;
- все обращения к функциям безопасности (аутентификация, управление сеансами, контроль доступа, криптографические функции), а также все ошибки и исключения, возникавшие в этих функциях;
- ошибки контроля данных на сервере;
- исключения и сбои во время выполнения;
- события управления журналами (включая ротацию), а также любые ошибки и исключения в процессе протоколирования.

Дополнительные сведения о том, что и как протоколировать, а что не протоколировать, изложенные с точки зрения приложения, см. в шпаргалке по протоколированию OWASP (Logging Cheat Sheet). Рекомендуем также прочитать книгу Dr. Anton Chuvakin, Gunnar Peterson «How to do Application Logging Right».

Обращайтесь аккуратно с секретными данными. Пароли, маркеры аутентификации, идентификаторы сеансов никогда не следует записы-

вать в журналы. Если в журналы записываются персональные данные или иная конфиденциальная информация, то к файлам журналов и к системе их резервного копирования должны применяться правила защиты данных, оговоренные в таких документах, как PCI DSS, GLBA или HIPAA.

Еще один важный вопрос – управление журналами.

Для протоколирования следует использовать безопасную централизованную систему, так чтобы в случае компрометации одного хоста журналы нельзя было уничтожить или модифицировать и чтобы противник не мог прочитать журналы и узнать о слабых местах и упущениях в вашей системе мониторинга. Консолидировать журналы различных приложений и узлов позволяют такие средства перемещения журналов, как rsyslog (*http://www.rsyslog.com/*), logstash (*https://github.com/elastic/logstash*) или fluentd (*http://www.fluentd.org/*), а также облачные службы протоколирования типа loggly (*https://www.loggly.com/*) или papertrail (*https://www.papertrail.io/uk/*).

Следует правильно организовать ротацию и хранение журналов, чтобы журналы можно было использовать для расследования проблем в процессе эксплуатации и для криминалистической экспертизы в случае взлома. Нормативные требования диктуют хранение в течение нескольких месяцев или даже лет, а не часов или дней.

Проактивное и реактивное обнаружение

Невозможно найти все проблемы безопасности в режиме, близком к реальному времени, применяя системы мгновенного обнаружения или онлайнового анализа (что бы ни говорили на этот счет многочисленные поставщики!). Для поиска иголки в стоге фонового шума часто приходится просеивать большие объемы данных, чтобы установить корреляции и выявить связи. Особенно это относится к задачам вроде обнаружения мошенничества, когда нужно прослеживать длительные промежутки времени и строить модели поведения, в которых учитывается множество переменных.

Говоря об обнаружении, многие имеют в виду только возможность выявлять события по мере наступления и реагировать на них или возможность определять и блокировать конкретные атаки во время работы системы. Но как можно, видя проблему, сказать, что это: изолированный инцидент или нечто такое, что происходило уже достаточно долго, прежде чем вы наконец заметили?

Многие взломы обнаруживаются только спустя несколько недель или месяцев после компрометации. Смогут ли ваши системы мониторинга

и обнаружения помочь оглянуться назад и определить, что произошло, когда, кто это сделал и что нужно предпринять для исправления?

При организации мониторинга необходимо также выбирать компромисс между стоимостью и временем. Бесполезно и неэффективно затоплять эксплуатационников или безопасников тревожными сигналами в попытке обнаружить проблемы немедленно. Лучше потратить дополнительное время, чтобы агрегировать и отфильтровать события, это сэкономит людям время и предотвратит или, по крайней мере, снизит вероятность апатии от постоянных тревог.

Обнаружение ошибок во время выполнения

Чем больше изменений вносится в системы, тем больше вероятность ошибок. Именно поэтому гибкие разработчики пишут автономные тесты: так создается страховочная сетка для вылавливания ошибок. Гибкие команды, а особенно команды DevOps, стремящиеся развертывать систему в тестовой и производственной среде чаще, должны делать то же самое в применении к эксплуатации, т. е. писать и прогонять тесты времени выполнения, которые проверяют, что приложение развернуто и сконфигурировано корректно и работает безопасно.

Компания Netflix с ее знаменитой «армией обезьян» (*https://github.com/Netflix/SimianArmy*) показала, как это можно сделать. Это набор инструментов, которые автоматически и непрерывно работают в производственной среде, проверяя, что все системы сконфигурированы и работают правильно:

- обезьяна безопасности (*https://github.com/Netflix/security_monkey*) проверяет наличие небезопасных политик и сохраняет историю изменения политик;
- обезьяна послушания (*https://github.com/Netflix/SimianArmy/wiki/Conformity-Home*) сравнивает конфигурацию работающего экземпляра с предопределенными правилами и уведомляет владельца (и группу безопасности) о любых нарушениях;
- обезьяна хаоса (*https://github.com/netflix/chaosmonkey*) и ее старшие сестры – горилла хаоса и Кинг-Конг хаоса знамениты тем, что случайным образом провоцируют отказы в производственной системе, чтобы проверить корректность ее восстановления.

Обезьяны Netflix первоначально предназначались для работы в облаке Amazon AWS, но сегодня некоторые из них (обезьяна хаоса, а сравнительно недавно и обезьяна безопасности) работают и на других

платформах. Их можно обобщить, реализовав свои правила, а можно на основе тех же идей создать собственный набор средств проверки во время выполнения, используя каркас автоматизированного тестирования типа Gauntlt (мы рассматривали его в главе 12), или InSpec, или даже более простые каркасы тестирования типа JUnit или BATS (*https://github.com/sstephenson/bats*).

Все эти тесты и проверки могут стать частью вашей программы непрерывной проверки на соответствие нормативным требованиям, которая докажет аудиторам, что соблюдение политики безопасности постоянно контролируется. Такая программа не только предотвращает ошибки по неосторожности, но и помогает обнаруживать идущие атаки и даже останавливать противника.

Безопасный SSL/TLS

Правильная настройка протокола SSL/TLS – вещь, которую обязаны понимать люди, занимающиеся сборкой или эксплуатацией систем, и тем не менее почти все делают это неправильно.

Айван Ристич (Ivan Ristic) из компании Qualys SSL Labs (*https://www.ssllabs.com/*) написал подробные и актуальные рекомендации по правильной настройке SSL/TLS (*http://bit.ly/github-configure-ssl-tls*).

SSL Labs предлагает также бесплатную службу, которой можно воспользоваться, чтобы сравнить конфигурацию вашего сайта с рекомендациями и получить отчет. Большая жирная красная буква F должна заставить прочитать рекомендации и разобраться, как делать правильно.

Проверка правильности настройки SSL должна стать стандартной частью ваших тестов и эксплуатационных проверок. Именно поэтому и Gauntlt, и BDD-Security включают тесты с использованием автономной программы sslyze (*https://github.com/nabla-c0d3/sslyze*). Это необходимо делать постоянно: криптографические алгоритмы, лежащие в основе SSL и TLS, всегда служат мишенью для атаки, и состояние, которое еще на прошлой неделе получило оценку A, сегодня, возможно, заслужит лишь D.

У среды, находящейся в состоянии постоянного и непредсказуемого коловращения (часто употребляют термин «инженерия хаоса»), есть одно преимущество, на которое часто не обращают внимания: для противника она представляет движущуюся мишень, что сильно затрудняет

описание среды эксплуатации и не дает занять удобную позицию. То и другое все-таки возможно, но стоимость атаки возрастает.

При написании тестов нужно решить, что делать, если утверждение оказывается ложным:

- изолировать машину и провести расследование;
- немедленно остановить службу или экземпляр системы;
- попытаться выполнить автовосстановление, если соответствующие действия просты и понятны: включить протоколирование или правило брандмауэра либо деактивировать небезопасный параметр по умолчанию;
- уведомить разработчиков, эксплуатационников или безопасников.

Оборона во время выполнения

Помимо мониторинга атак и проверки правильности конфигурации системы, у вас может возникнуть желание – или необходимость – принять дополнительные меры защиты во время выполнения.

Традиционные системы предотвращения вторжений и основанные на сигнатурах брандмауэры веб-приложений (Web Application Firewall – WAF), расположенные где-то перед приложением, не предназначены для такого быстрого изменения приложения и технологий, как практикуется в гибких методиках и DevOps. Особенно это относится к системам, работающим в облаке, где не существует четкого периметра сети, защищаемого брандмауэром, и где разработчики могут непрерывно развертывать изменения на сотнях или тысячах эфемерных экземпляров в публичных, частных и гибридных средах.

Обеспечение безопасности в облаке

С осознанием рисков, которые несут новые подходы к эксплуатации, появился ряд стартапов в области безопасности, предлагающих различные службы защиты облачных приложений, включая автоматизированный анализ атак, централизованное управление учетными записями и применение политик, непрерывный мониторинг целостности файлов и обнаружение вторжений, автоматизированное сканирование на уязвимости и микросегментацию:

- Alert Logic (*https://www.alertlogic.com/*);
- CloudPassage Halo (*https://www.cloudpassage.com/products/*);
- Dome9 SecOps (*https://dome9.com/*);
- Evident.io (*http://evident.io/*);

- Illumio (*https://www.illumio.com/home*);
- Palerra LORIC (*http://palerra.com/platform/*) (приобретена Oracle и теперь называется Oracle CASB Cloud Service);
- tCell (*https://www.tcell.io/*);
- Threat Stack (*https://www.threatstack.com/*).

Все эти службы подключаются к облачной платформе через ее API и применяют собственные средства аналитики и обнаружения угроз, заменяющие «черные ящики» безопасности на предприятиях вчерашнего дня.

Другие стартапы, например Signal Sciences (*https://www.signalsciences.com/*), предлагают более интеллектуальные брандмауэры веб-приложений следующего поколения (NGWAF), которые можно развернуть вместе с облачными приложениями. В них, чтобы идентифицировать и блокировать полезную нагрузку атаки, применяется прозрачное обнаружение аномалий, языковое разложение, непрерывный анализ данных об атаке и машинное обучение вместо правил на основе сигнатур.

Хотим предостеречь против безоглядной веры в машинное обучение! В настоящее время большие данные и машинное обучение у всех на слуху; производители не могут противиться соблазну включить их в свои продукты – вместе с громкими заявлениями о том, что это и есть та панацея, которую так долго ждала отрасль безопасности. Но, как и все уверения производителей, это нужно принимать с изрядной долей скептицизма и интересоваться деталями: какие именно методы машинного обучения применяются и какими данными подтверждается их релевантность стоящей перед вами задаче. Ложные тревоги часто губят идею на корню, а время, необходимое для обучения моделей на реальной производственной среде, стало непреодолимым препятствием не для одной великой «готовой панацеи» в области безопасности.

RASP

Еще один вид защитной технологии – *самозащита приложения во время выполнения* (runtime application self-protection – RASP). В этом случае исполняющая среда приложения оснащается средствами для обнаружения проблем с безопасностью по мере их возникновения. Как и брандмауэр уровня приложения, RASP может автоматически выявлять и блокировать атаки в момент их проведения. И точно так же RASP можно использовать для защиты унаследованных или сторонних приложений, для которых нет исходного кода.

Но, в отличие от брандмауэра, RASP не является средством обороны периметра. RASP предполагает непосредственное оснащение и мониторинг кода исполняющей среды и способен выявлять и блокировать атаки прямо в точке их выполнения. Средства RASP видят код приложения и контекст исполнения, они инспектируют переменные и предложения программы, применяя анализ потока данных, потока управления и методы лексического анализа, чтобы обнаружить атаку в процессе выполнения кода. Они не просто сообщают о проблемах, которые могли бы произойти, а перехватывают проблемы, которые происходят прямо сейчас. Это означает, что средства RASP, вообще говоря, порождают гораздо меньше ложноположительных (и ложноотрицательных) результатов, чем брандмауэры уровня приложения и инструменты статического анализа кода.

Большинство этих инструментов умеют перехватывать атаки путем внедрения SQL и некоторые виды XSS-атак, а также включают защиту от конкретных уязвимостей, например Heartbleed, применяя правила, основанные на сигнатурах. RASP можно использовать как средство обороны во время выполнения, включив режим блокировки, или для автоматического протоколирования попыток внедрения и аудита унаследованного кода, а также для лучшего понимания работающего приложения и атак против него.

На сегодняшний день существует совсем немного RASP-решений. В основном их предлагают небольшие стартапы, и ориентированы они на приложения, работающие в среде Java JVM или .NET CLR, хотя начинают появляться также решения для таких платформ, как Node.js, PHP, Python и Ruby:

- Contrast Security (*https://www.contrastsecurity.com/rasp*);
- HPE Application Defender (*http://bit.ly/hpe-application-defender*);
- Immunio (*https://www.immun.io/*);
- Prevoty (*https://www.prevoty.com/*);
- Waratek (*http://www.waratek.com/*).

Убедить команду использовать RASP может оказаться нелегко – а группу эксплуатации еще труднее. Вам придется уговорить коллег довериться способности решения точно находить и блокировать атаки и согласиться на сопряженные с этим накладные расходы во время выполнения. Кроме того, RASP приносит с собой новые точки отказа и эксплуатационные проблемы: кто должен отвечать за настройку, контролировать правильность работы и обрабатывать результаты?

OWASP AppSensor: запускаем собственный RASP

Проект OWASP AppSensor предлагает набор паттернов и пример кода на Java в помощь тем, кто желает реализовать обнаружение вторжений и реакцию на них на уровне приложения.

AppSensor строит карту типичных точек обнаружения в веб-приложениях: точки входа, куда можно добавить проверки ситуаций, которые не должны происходить в ходе нормальной работы. Затем определяются варианты реакции на такие ситуации: когда просто протоколировать, а когда блокировать атаку и как это делать. Демонстрируется, как автоматически обнаруживать многие распространенные атаки и как защищаться от них.

Но RASP может предложить заманчивое «латание по-быстрому» для команд, испытывающих давление, особенно пытающихся поддерживать небезопасный унаследованный или сторонний код или использующих брандмауэры веб-приложений для защиты своих приложений и вынужденных преодолевать ограничения этой технологии.

Взлом и сбой: то же, да не совсем

И группы эксплуатации, и группы реагирования на инциденты безопасности должны действовать быстро и эффективно, когда случается неприятность. Они должны знать, как и когда обращаться к высшей инстанции и как взаимодействовать с заинтересованными сторонами. И еще они должны учиться на собственном опыте, чтобы в следующий раз действовать лучше.

Но приоритеты различны. Если имеет место операционный сбой или серьезная проблема с производительностью, то задача группы эксплуатации – как можно быстрее восстановить работоспособность службы. С другой стороны, группа реагирования на инциденты безопасности должна быть уверена, что понимает масштаб атаки и знает, как ограничить ее последствия. Кроме того, она должна сделать снимки образов исполняющей среды и собрать журналы для криминалистической экспертизы – и только потом вносить какие-то изменения и восстанавливать работоспособность.

Интересы эксплуатации и безопасности могут вступать в конфликт, например в случае DDOS-атаки. Организация должна решить, что важнее или более срочно: восстановить службу или найти источник атаки и подавить его?

Реакция на инциденты: подготовка к взлому

Даже сделав все, о чем шла речь выше, нужно быть готовым к действиям в условиях отказа системы или взлома. Создавайте памятки, деревья вызова, лестницы эскалации, чтобы все знали, кого звать на помощь. Заранее расписывайте роли, чтобы, когда нагрянет беда и придется работать в условиях стресса, все знали, что и как делать, и не поддавались панике.

Тренируйтесь: учения и команда красных

Планирования и росписи сценариев недостаточно. Единственный способ обрести уверенность в том, что вы сможете успешно отреагировать на инцидент, – отработать действия на практике.

Учения

В Amazon, Google, Etsy и других компаниях, ведущих бизнес в Интернете, регулярно проводятся учения, когда отрабатываются действия в случае реального крупномасштабного сбоя, например выхода из строя всего центра обработки данных. Смысл в том, чтобы проверить практическую пригодность процедур отработки отказа и готовность команды к аварийной ситуации.

В этих учениях могут принимать участие (например, в Google) сотни инженеров, работающих круглосуточно в течение нескольких дней, чтобы протестировать процедуры аварийного восстановления и оценить, как стресс и усталость могут повлиять на способность организации справляться с реальными авариями.

В Etsy игровые дни проводятся в производственной среде и затрагивают даже ключевые функции, в т. ч. обработку платежей. Естественно, напрашивается вопрос: «Почему бы не имитировать аварию в тестовой или промежуточной среде?» Etsy отвечает, что, во-первых, даже небольшие различия в среде порождают неуверенность, а во-вторых, риск потерпеть неудачу при восстановлении тестовой среды не влечет за собой никаких последствий, что может привнести скрытые предположения при проектировании отказоустойчивости и процедуры восстановления. Цель учений – уменьшить неуверенность, а не увеличить ее.

Эти учения тщательно тестируются и планируются заранее. Команда проводит мозговые штурмы, чтобы придумать различные сценарии отказа и подготовиться к ним, отрабатывает их сначала на тестах и устраняет возникшие проблемы. А затем, когда приходит время воплотить сценарий в производственной среде, разработчики и операторы вни-

мательно наблюдают в полной готовности вмешаться и заняться восстановлением, особенно если случится что-то неожиданное.

Можно регулярно устраивать и менее грандиозные учения. В потоке Tabletop Scenarios (*https://twitter.com/badthingsdaily*) в Твиттере можно найти темы бесед «что, если» для мозгового штурма во время перерыва на кофе.

Команда красных против команды синих

Из учений можно позаимствовать много идей о том, как проверить отказоустойчивость системы и готовность команды DevOps к устранению системных сбоев. Эти идеи можно применить к отработке сценариев атак со стороны *команды красных*.

В таких организациях, как Microsoft, Intuit, Salesforce, и нескольких крупных банках, есть постоянные команды красных, которые непрерывно атакуют реальные производственные системы. В других организациях периодически устраивают не анонсированные заранее нападения команды красных, составленной из штатных сотрудников или привлеченных консультантов, с целью проверить готовность групп безопасности и эксплуатации (см. главу 12).

Идея противоборства красных и синих взята из военных игр «захвати флаг». Команда красных – небольшая группа атакующих – пытается взломать систему (не причиняя реального ущерба), а команда синих (разработчики, эксплуатационники и безопасники) стремится обнаружить и остановить их. Команда красных испытывает реальные примеры атак, чтобы организация могла узнать, как в действительности выглядит атака и как с ней бороться.

Некоторые учения длятся всего несколько часов, тогда как другие могут продолжаться дни или недели, имитируя продвинутую и постоянную угрозу.

Успех команды красных оценивается по количеству найденных серьезных проблем, по скорости их эксплуатации (среднее время до срабатывания эксплойта) и по тому, как долго ей удалось оставаться необнаруженной.

Команде синих может быть известно о том, что атака запланирована, и о том, какие системы будут мишенями, но детали атак ей не сообщают. Успех команды синих измеряют в терминах среднего времени обнаружения и среднего времени восстановления: сколько времени потребовалось, чтобы обнаружить и идентифицировать атаку, и сколько – чтобы остановить или ограничить ее и восстановить работоспособность систем.

Метод команд красных дает возможность увидеть, как система и группа эксплуатации будут вести себя в условиях атаки. Вы узнаете, как выглядят атаки, и обучите сотрудников распознавать атаки и реагировать на них. Если проводить такие учения регулярно, то ответные действия станут более слаженными и быстрыми. Кроме того, вы получаете шанс изменить мышление людей, побудив их рассматривать атаки не как нечто абстрактное и гипотетическое, а как вполне осязаемое и требующее срочных действий.

Со временем команда синих приобретает опыт распознавания атак и защиты от них, поэтому команде красных приходится прилагать больше усилий, копать глубже, проявлять изобретательность. Чем острее конкуренция между командами, тем более защищенной становится система и более высоким – общий уровень безопасности.

К примеру, Intuit проводит учения с участием команды красных в первый день недели (они называют это «красными понедельниками» – см. *https://www.acast.com/eyeonsecurity/red-blue-and-intuit*). Команда красных в течение недели намечает системы-мишени и планирует атаку. Каждую пятницу объявляется, какие системы будут атакованы. Команды синих, защищающие эти системы, часто посвящают выходные подготовке – поиску и устранению уязвимостей, – чтобы затруднить работу команде красных. По завершении учений команды собираются вместе, чтобы обсудить результаты и наметить план действий. Затем все начинается сначала.

Большинство организаций не в состоянии организовать нечто подобное своими силами. Для этого необходимы солидные знания и решимость. Как уже было сказано в главе 12, для проведения игр с участием команды красных можно обратиться за помощью извне.

Посмертный анализ без поисков виновного: обучение на инцидентах безопасности

Учения и игры с командой красных – хорошая возможность для обучения. Но еще важнее почерпнуть как можно больше нового и полезного, если в производственной системе действительно что-то случается: операционный сбой или взлом. В таком случае нужно собрать всю команду и проанализировать посмертные данные, чтобы понять, что случилось, почему и как предотвратить такие проблемы в будущем.

Посмертный анализ заимствует некоторые идеи гибкого ретроспективного анализа, когда команда на общем собрании обсуждает, что сделано, что было хорошо и как можно сделать еще лучше. В процессе посмертного анализа команда обсуждает имевшие место факты: что и

когда произошло, как люди реагировали и что случилось потом. В рассмотрение включаются дата и время события, доступная на тот момент информация, решения, принятые на основе этой информации, и результаты этих решений.

Спокойно и объективно обсудив факты и вскрывшиеся проблемы, члены команды могут узнать много нового о системе и о самих себе, а значит, понять, что нужно изменить.

Существует несколько источников информации, на основе которых можно составить картину происшедшего в процессе посмертного анализа. Это журналы, электронная почта, хранимые данные об активности в чатах и отчеты о дефектах. Компания Etsy разработала Morgue (*https://github.com/etsy/morgue*), онлайновый инструмент посмертного анализа с открытым исходным кодом и подключаемые модули к нему для получения информации из таких источников, как IRC и Jira, а также журналов и снимков мониторинга. Этот инструмент позволяет формировать посмертные отчеты.

Факты – вещь конкретная, понятная и не вызывающая опасений. Выложив факты на стол, команда может сначала задать вопросы: почему произошла ошибка, почему были приняты те, а не иные решения, а затем изучить альтернативы и понять, как можно действовать лучше.

- Как улучшить проект системы, сделав ее проще и безопаснее?
- Какие проблемы можно выловить в процессе тестирования и инспекции?
- Как помочь людям выявлять проблемы на более ранней стадии?
- Как облегчить реагирование на проблемы, упростить процесс принятия решений и снизить уровень стресса – за счет более качественной информации и инструментов, путем обучения или составления сценариев?

Чтобы все это работало, лица, принимающие участие в посмертном анализе, должны твердо знать, что его цель – чему-то научиться, а не найти козла отпущения. Они не должны бояться делиться информацией, быть честными и ничего не скрывать, критически осмыслять случившееся, не опасаясь стать мишенью для критики и упреков. В главе 15 мы еще поговорим о том, как создать в коллективе атмосферу доверия и безбоязненности, и обсудим вопросы доверия и готовность учиться в процессе посмертного анализа.

Эксплуатационные отказы во многих отношениях проще взломов. Они бросаются в глаза, их проще понять и принять решение о том, как быть. Инженеры видят цепочку событий или, по крайней мере, понимают, где имеются упущения и что нужно сделать для улучшения процедур или самой системы.

Чтобы разобраться со взломом, нужно больше времени, а цепочка причин и следствий не всегда очевидна. Возможно, придется привлечь экспертов-криминалистов, они смогут просеять журналы и восстановить события достаточно полно, чтобы команда поняла, в чем состояла уязвимость и как ее эксплуатировали. К тому же часто существует длительный промежуток времени между моментом взлома и моментом его обнаружения. Опытный противник часто старается уничтожить или исказить свидетельства, необходимые для реконструкции взлома, потому-то так важно защищать и архивировать журналы.

Защита сборочного конвейера

Автоматизация процессов сборки, интеграции и тестирования, начиная с записи в репозиторий, – одна из основ жизненного цикла гибкой разработки. Однако она несет с собой и новую ответственность – теперь весь этот конвейер надо защищать!

Учитывая, какие задачи делегируются сборочному конвейеру и какие полномочия необходимы для их выполнения, нетрудно понять, что он оказывается ключевым элементом инфраструктуры, а значит, желанной мишенью для атаки.

Эта притягательность многократно возрастает, если вы практикуете непрерывное развертывание, когда каждое изменение автоматически попадает в производственную систему после тестирования.

Автоматизация сборки и развертывания увеличивает поверхность атаки производственной системы, теперь она включает также среду и комплекты инструментов сборки. Если репозитории, сборочные серверы или системы управления конфигурацией скомпрометированы, то ситуация очень быстро становится крайне серьезной.

Если в результате компрометации получен доступ на чтение, то объектом кражи могут стать данные, исходный код и различные секреты, в т. ч. пароли и ключи API. Если же получен доступ на запись или выполнение, то сброшены уже все маски: открывается возможность внедрить в приложения черные ходы или вредоносный код, перенаправить или перехватить рабочий трафик и, наконец, полностью уничтожить производственные системы.

Даже если скомпрометирована только тестовая система, этого может хватить противнику, чтобы внедриться в автоматизированный конвейер и причинить ущерб.

Утратив контроль над сборочным конвейером, вы утратите также возможность реагировать на атаку, поскольку не сможете развернуть заплату или срочное исправление.

Защищать конвейер нужно не только от внешних противников, но и от компрометации инсайдерами. Для этого все изменения должны иметь автора, быть прозрачными и прослеживаемыми от начала до конца, чтобы замысливший вред и при этом информированный инсайдер не мог обойти средства контроля и внести изменения, оставшись незамеченным.

Постройте модель угроз сборочному конвейеру. Ищите слабые места в настройке и средствах контроля, а также пробелы в аудите и протоколировании. Затем предпримите следующие действия, чтобы обезопасить среду управления конфигурацией и сборочный конвейер:

1. Укрепите исполняющую среду управления конфигурацией, сборки и тестирования.
2. Выясните, какие действия производятся в облаке, и примите меры для их контроля.
3. Укрепите комплекты инструментов для сборки, непрерывной интеграции и непрерывной поставки.
4. Ограничьте доступ к инструментам управления конфигурацией.
5. Защитите ключи и другие секретные данные.
6. Ограничьте доступ к репозиториям исходных и двоичных файлов.
7. Защитите платформы чатов (особенно если практикуете методику ChatOps).
8. Регулярно просматривайте журналы управления конфигурацией, сборки и тестирования.
9. Пользуйтесь серверами-фениксами для создания подчиненных узлов сборки и тестирования; создавайте такие среды с нуля всякий раз, как в них возникает надобность, и уничтожайте по завершении работы.
10. Наблюдайте за сборочными конвейерами так же, как за производственной системой.

Как видим, много всего. Рассмотрим эти шаги по отдельности.

Укрепление инфраструктуры сборки

Укрепляйте системы, на которых размещены репозитории исходного кода и артефактов сборки, серверы непрерывной интеграции и поставки, а также инструменты управления конфигурацией, сборки и развертывания. Обращайтесь с ними так же, как с самыми ценными производственными системами, содержащими самые конфиденциальные данные.

Проанализируйте правила брандмауэра и сегментацию сети, обращая внимание на то, чтобы эти системы, равно как и системы для разработки и тестирования, случайно не оказались подключены к открытому Интернету и чтобы среды разработки и эксплуатации были строго разделены. Пользуйтесь контейнерами и виртуальными машинами для обеспечения дополнительной изоляции во время выполнения.

Выяснение того, что происходит в облаке

Масштабирование сред сборки и тестирования с помощью облачных служб выглядит легко и заманчиво. Но вы должны четко понимать – и контролировать, – какие части конвейеров сборки и тестирования находятся на территории предприятия, а какие – в облаке. Использование облачных служб имеет много преимуществ, но может породить проблемы доверия и значительно увеличить поверхность атаки, о которой вам придется заботиться.

Размещение репозиториев кода в облаке с помощью служб GitHub, Gitlab или BitBucket идеально подходит для проектов с открытым исходным кодом (разумеется), а также для стартапов и небольших команд. Эти службы предоставляют много полезного бесплатно или за очень скромные деньги. Но одновременно они являются желанными мишенями для противника, поэтому увеличивается риск, что ваш код и все, что в нем хранится (например, секреты), может быть скомпрометирован в результате целенаправленной атаки или широкомасштабного взлома.

Облачные репозитории кода типа GitHub постоянно находятся под угрозой, потому что атакующие знают, где и что искать. Они в курсе, что разработчики бывают беспечны и не всегда используют сильные пароли, отвергают многофакторную аутентификацию и другие средства защиты и что иногда по ошибке хранят закрытый код в публичных репозиториях. Эти упущения, а также ошибки кодирования или эксплуатации, допущенные поставщиками служб, за последние годы стали причиной многих резонансных взломов.

Если ваши разработчики собираются хранить закрытый код в облаке, то предпримите следующие действия для его защиты.

1. Заставьте разработчиков применять строгую аутентификацию (включая и многофакторную).
2. Убедитесь, что частные репозитории действительно частные.
3. Следите за своими репозиториями на GitHub с помощью инструментов типа GitMonitor.
4. Перед тем как записать код в репозиторий, просканируйте или проинспектируйте его на предмет отсутствия учетных данных.

Потенциальные последствия не ограничиваются репозиториями организации, а затрагивают и личные репозитории разработчиков. Склонность разработчиков делиться со всем миром своими *файлами с точкой* стала исходным пунктом для компрометации многих внутренних сетей, имеющих выход в Интернет.

Если вы размещаете в облаке такие службы непрерывной сборки и интеграции, как Travis CI или Codeship, проверьте, правильно ли настроен доступ, и убедитесь, что понимаете, как устроены их политики безопасности и конфиденциальности.

Наконец, как и для любого решения на платформе SaaS, настоятельно необходимо строго контролировать доступ к этим системам и отзывать права доступа у сотрудников, переходящих на другую работу или покидающих компанию. Если вы в значительной мере опираетесь на облачные службы, то подумайте об использовании единой точки входа (single-sign-on – SSO) для доступа ко всем приложениям с помощью одного идентификатора. Когда нужно поддерживать несколько учетных записей в разных системах, при увольнении сотрудника легко забыть о какой-то из них и оставить доступ открытым. С учетом того, что доступ к облачным приложениям глобальный (так и задумано), требование об использовании многофакторной аутентификации для защиты от кражи учетных данных становится обязательным.

Укрепление инструментов непрерывной интеграции и поставки

Укрепите комплекты инструментов сборки. Большинство таких инструментов устроено так, чтобы разработчики могли просто и быстро настроить и запустить их, но это означает, что по умолчанию они небезопасны – и сделать их безопасными трудно, если вообще возможно.

Хорошим примером может служить Jenkins, один из самых популярных инструментов автоматизированной сборки. Хотя в последнюю версию добавлены некоторые средства обеспечения безопасности, большая их часть по умолчанию выключена, в т. ч. и базовые средства аутентификации и контроля доступа. Вот что об этом написано на сайте[1]:

> По умолчанию Jenkins не производит никаких проверок, относящихся к безопасности. Это означает, что Jenkins может запускать процессы и получать доступ к локальным файлам, доступным всем тем, кто имеет доступ к веб-интерфейсу Jenkins, и не только.

Потратьте время, чтобы понять модель авторизации, применяемую в инструменте, и каким образом можно ограничить доступ. Настройте отношения доверия между мастерами сборки и серверами и включите прочие имеющиеся механизмы защиты. Затем логически разделите сборочные конвейеры разных команд. Тогда если один будет скомпрометирован, то хотя бы другие не пострадают.

Ограничив доступ к инструментам и включив средства обеспечения безопасности, вы должны также вовремя устанавливать обновления и исправления самих инструментов и всех необходимых подключаемых модулей. На заверения о безопасности этих инструментов, даже самых популярных, не всегда можно полагаться. Следите за выпуском исправлений и устанавливайте их сразу по мере появления. Но сначала обязательно протестируйте исправление, чтобы убедиться в его стабильности и отсутствии зависимостей: инструменты сборки со временем становятся сильно «заточенными» под пользователя и хрупкими.

Инструменты непрерывной интеграции – лучшие друзья хакера

«Работать с плохо сконфигурированным инструментом непрерывной интеграции – все равно, что предоставить готовую сеть ботов, доступную любому желающему».

Познакомьтесь с презентацией исследователя безопасности Адриана Миттала на конференции Black Hat 2015, в которой он рассматривает серьезные уязвимости, обнаруженные в различных инструментах непрерывной интеграции и поставки, включая Jenkins, Go и TeamCity (Nikhil Mittal «Continuous Intrusion: Why CI tools are an attacker's best friend», презентация на конференции Black Hat Europe 2015 (*https://www.blackhat.com/docs/eu-15/materials/eu-15-Mittal-Continuous-Intrusion-Why-CI-Tools-Are-An-Attackers-Best-Friend.pdf*)).

[1] Jenkins «Securing Jenkins», 15 апреля 2016 (*https://wiki.jenkins-ci.org/display/JENKINS/Securing+Jenkins*).

Ограничение доступа к диспетчерам конфигурации

Если вы пользуетесь такими инструментами управления конфигурацией, как Chef или Puppet, то должны ограничить доступ к ним. Любой, имеющий доступ к этим инструментам, может заводить новые учетные записи, изменять права доступа к файлам и политики аудита, устанавливать скомпрометированное ПО и изменять правила брандмауэра. Это все равно, что предоставить кому-то доступ с правами суперпользователя ко всем машинам с управляемой конфигурацией. Кому нужен пароль root, если кто-то другой будет любезно набирать за вас все команды?

Настройте инструменты безопасно, предоставьте доступ к ним только небольшой группе доверенных лиц – и аудируйте все, что они делают. См., например, статью на сайте Learn Chef Rally «How to be a secure Chef» (*https://learn.chef.io/modules/securing-chef/be-a-secure-chef#/*).

Защита ключей и секретов

При работе с конвейером непрерывной поставки необходимы ключи и другие учетные данные для автоматической подготовки серверов и развертывания кода. Кроме того, самой системе нужны учетные данные для запуска и исполнения. Убедитесь, что эти секреты не зашиты в скрипты, незашифрованные конфигурационные файлы или в код. О том, как безопасно обращаться с секретами, мы расскажем в следующем разделе.

Ограничение доступа к репозиториям

Ограничьте доступ к репозиториям исходных и двоичных файлов и аудируйте доступ к ним. Запретите неаутентифицированный, анонимный или совместный доступ к репозиториям и установите правила контроля доступа.

В репозиториях исходного кода хранится код вашего приложения, тесты и тестовые данные, конфигурационные рецепты и, если вы совсем уж беспечны, учетные данные и прочие вещи, которыми вы не хотели бы делиться с противником. Доступ для чтения к репозиториям исходного кода должен быть предоставлен только членам вашей и других команд, которые с этим кодом работают.

Всякий имеющий доступ к репозиториям кода для записи может записать вредоносное изменение. Проверьте, что этот доступ контролируется и что все операции записи находятся под непрерывным наблюдением.

Репозитории двоичного кода содержат кеш сторонних библиотек, скачанных из публичных репозиториев или с сайтов производителей,

а также последние результаты сборки вашего кода. Всякий, имеющий доступ к этим репозиториям для записи, может поместить вредоносный код в вашу среду сборки – и в конечном итоге в производственную систему.

Следует проверять сигнатуры сторонних компонентов, помещаемых в кеш для внутреннего использования, – как сразу после скачивания, так и в момент, когда компоненты участвуют в сборке. Кроме того, настройте шаги сборки, так чтобы двоичные файлы и другие артефакты сборки подписывались – во избежание манипуляций.

Безопасный чат

Если вы пользуетесь средствами организации коллективных чатов, например Slack, Mattermost или Hubot (*https://github.com/github/hubot*) на GitHub для помощи в автоматизации сборки, тестирования, выпуска и развертывания, то появляется новый набор рисков и вопросов.

Коллективные чаты типа Slack и HipChat предоставляют разработчикам и эксплуатационникам простой и естественный способ обмениваться информацией. Чат-боты могут автоматически отслеживать и сообщать о состоянии системы, наблюдать за конвейерами сборки и развертывания и посылать информацию в комнаты или каналы. Их также можно использовать для автоматической настройки и управления сборкой, тестированием, выпуском, развертыванием и другими эксплуатационными операциями, для чего достаточно ввести простые команды в процессе чата.

Какие комнаты являются каналами и какими именно: публичными или частными? Кто имеет доступ к этим каналам? Открыты ли какие-то из них заказчикам, третьим сторонам или всем? Какая информация в них циркулирует?

Для каких действий настроены боты? Кто имеет к ним доступ? Где работают боты? Где находятся скрипты?

Вы должны предпринять необходимые меры, чтобы обезопасить и ограничить доступ к соответствующим комплектам инструментов: самой программе чата или платформе для организации чатов, ботам, скриптам и подключаемым модулям, используемым членами команды. Обращайтесь со скриптами автоматизации чатов как с любым другим эксплуатационным кодом: не забывайте инспектировать его и храните в репозитории.

Многие средства коллективной работы теперь находятся в облаке, а это значит, что информация, которую постят члены команды, размещается где-то вне организации. Вы должны проанализировать средства

безопасности, применяемые поставщиком услуги, а также его политики безопасности и конфиденциальности. Не забывайте следить за относящимися к безопасности объявлениями поставщика и убедитесь, что готовы к сбоям и взломам.

И проверьте, что команды понимают, какую информацию можно постить в комнаты и каналы, а какую нельзя. Включение паролей и другой конфиденциальной информации в сообщения должно быть запрещено.

Контролируйте и аудируйте доступ к чатам, включая многофакторную аутентификацию, если она поддерживается. Если команда достаточно велика, то, возможно, придется настроить схемы разрешений для работы с ботами, воспользовавшись чем-то вроде программы hubot-auth (*https://github.com/hubot-scripts/hubot-auth*).

Помните, что ботам нужны учетные данные – для самой системы чатов и для других систем и инструментов, с которыми боты взаимодействуют. Эти секреты необходимо защищать, а как – будет описано ниже в этой главе.

Просмотр журналов

Журналы системы сборки должны быть частью тех же технологических процессов, что журналы производственных систем. Периодически просматривайте журналы инструментов – вы должны быть уверены в их полноте и в том, что сумеете проследить изменение от момента записи в репозиторий до развертывания. Журналы должны быть неизменяемыми, чтобы их нельзя было ни стереть, ни подделать. И не забывайте регулярно ротировать и архивировать журналы.

Использование серверов-фениксов для сборки и тестирования

Применяйте средства автоматизированного управления конфигурацией: Chef, Puppet, Docker (особенно рекомендуется), Vagrant и Terraform, чтобы автоматически поднять, настроить, «пропатчить» и уничтожить серверы сборки и тестирования по мере необходимости.

Старайтесь, чтобы машины для сборки и тестирования были одноразовыми, эфемерными «серверами-фениксами» (*https://martinfowler.com/bliki/ PhoenixServer.html*), которые существуют только на протяжении тестового прогона. Это уменьшает поверхность атаки, а заодно дает возможность регулярно проверять процессы управления конфигурацией, вселяя дополнительную уверенность при развертывании в укрепленной производственной среде.

Не дайте тестовым данным навлечь на вас беду

Поскольку создать хорошие синтетические тестовые данные трудно, во многих компаниях поступают проще: берут реальные данные из производственной системы или их подмножество и маскируют некоторые поля, стирая в них персональные данные. Если сделать это неправильно, то возможна утечка данных или другие нарушения законов о неприкосновенности частной жизни и нормативно-правовых требований.

Необходимо строго контролировать обращение со снимками системы и внедрить надежные (и тщательно проинспектированные) методики, гарантирующие, что персональная информация: имена, адреса, телефоны, адреса электронной почты, а также пароли, другие учетные данные и прочие секретные сведения – удалена и заменена случайными данными или иным способом сделана бесполезной, – только потом данные можно использовать для тестирования.

Все шаги этой процедуры должны аудироваться, тогда вы сможете доказать клиентам и аудиторам, что информация была защищена.

Мониторинг систем сборки и тестирования

Эти системы должны подвергаться мониторингу как часть производственной среды. Эксплуатационную безопасность, в т. ч. сканирование на уязвимости, системы обнаружения и предотвращения вторжений, мониторинг исполняющей среды, нужно распространить на инструменты и инфраструктуру, в которой они используются. Применяйте средства проверки целостности файлов, чтобы обнаруживать неожиданные или неавторизованные изменения конфигурации и данных.

Шшш... секреты должны храниться в секрете

Сохранение секретности секретов – проблема в любой системе. Приложению нужны ключи, пароли и идентификаторы пользователей, строки соединения, ключи AWS, ключи для подписывания кода, маркеры API и другие секретные данные, которые необходимо защищать. А инженерам-эксплуатационникам и администраторам – и их инструментам – нужен доступ к этим секретам.

С автоматизацией задач конфигурирования, тестирования, развертывания и управления системами проблема обращения с секретами

только усложняется. Невозможно раскрыть секреты лишь небольшой кучке людей. И хранить их в скриптах, незашифрованных конфигурационных файлах – или в исходном коде – тоже нельзя.

Хранить секреты в коде – идея хуже не придумаешь. Код широко доступен, особенно если вы пользуетесь распределенной системой управления версиями типа Git, в которой у каждого разработчика имеется собственная копия кода, т. е. каждый разработчик имеет доступ ко всем системным секретам. А если понадобится изменить секреты, то придется вносить изменения в код и повторно развертывать систему. А код имеет тенденцию просачиваться наружу, выставляя секреты на всеобщее обозрение.

Невозможно сохранить секреты в секрете на GitHub

Было несколько случаев, когда люди, работающие в широко известных организациях, были замечены в отправке паролей, ключей и прочих секретов в публичные репозитории на GitHub, по которым производится поиск (Dan Goodin «PSA: Don't upload your important passwords to GitHub», Ars Technica, 1/24/2013 (*http://bit.ly/goodin-passwords-to-github*)).

В частности, речь идет о людях, которые без всякого злого умысла закачали в публичный репозиторий на GitHub код Slackbot и случайно включили свои частные маркеры Slack API, которыми можно было воспользоваться для подслушивания разговоров в Slack или для действий от лица других пользователей.

А недавно прогремел случай, когда компания Uber признала, что противник скомпрометировал базу данных водителей, воспользовавшись ключом, который был случайно отправлен на GitHub (Dan Goodin, «In major goof, Uber stored sensitive database key on public GitHub page», Ars Technica, 3/2/2015 (*http://bit.ly/goodin-database-key-on-github*)).

Регулярно сканируйте GitHub с помощью программ Gitrob или Truffle Hog, которые ищут в публичных репозиториях файлы вашей организации, содержащие секретные данные.

Для прозрачного шифрования секретов и других конфигурационных данных и конфиденциального кода в момент записи в репозиторий можно пользоваться такими программами, как git-secret (*https://github.com/sobolevn/git-secret*) или BlackBox (*https://github.com/StackExchange/blackbox*) от StackExchange либо git-crypt (*https://github.com/AGWA/git-crypt*).

Но все равно риски остаются. Что, если кто-то забудет зашифровать файл, содержащий конфиденциальную информацию?

Необходимо сканировать или инспектировать код с целью убедиться, что учетные данные не попадают в репозиторий, либо, воспользовавшись точками подключения перед записью, добавить проверку на пароли и ключи. Проект Talisman (*https://github.com/thoughtworks/talisman*) с открытым исходным кодом от компании ThoughtWorks, Git Hound (*https://github.com/ezekg/git-hound*) и git-secrets (*https://github.com/awslabs/git-secrets*) – все эти инструменты позволяют автоматически искать секреты в коде и запрещают записывать такой код в репозиторий.

Доступ к секретам необходим серверам непрерывной интеграции и непрерывной поставки. А также скриптам развертывания и средствам автоматизации выпуска. И инструментам управления конфигурацией – Ansible, Chef и Puppet – доступ к секретам нужен, чтобы создавать другие секреты. Для управления секретами можно использовать следующие программы:

- Ansible Vault (*http://docs.ansible.com/ansible/playbooks_vault.html*);
- Chef Vault (*https://github.com/chef/chef-vault*);
- hiera-eyaml для Puppet (*https://github.com/TomPoulton/hiera-eyaml*).

Но это решает только часть проблемы.

Гораздо правильнее было бы использовать универсальный диспетчер секретов, применимый ко всем инструментам и всем вашим приложениям. Диспетчеры секретов выполняют следующие функции:

1) защищенное хранение и шифрование паролей, ключей и прочих учетных данных в месте хранения;
2) ограничение и аудит доступа к секретам, обязательная аутентификация и детальные правила контроля доступа;
3) предоставление API для безопасного доступа;
4) обработка отказов таким образом, что секреты всегда доступны пользователям системы.

Ниже перечислено несколько диспетчеров секретов с открытым исходным кодом:

1) Keywhiz (*https://github.com/square/keywhiz*) компании Square;
2) Knox (*https://github.com/pinterest/knox*) – хранитель секретов, используемый в Pinterest;
3) Confidant (*https://github.com/lyft/confidant*) компании Lyft, используется на платформе AWS;

4) CredStash (*https://github.com/fugue/credstash*) – простой хранитель секретов для AWS;
5) Vault (*https://www.vaultproject.io/*) компании HashiCorp – пожалуй, самый полный и готовый к реальной эксплуатации из диспетчеров секретов с открытым исходным кодом.

Сухой остаток

Надежная и безопасная эксплуатация системы ставит непростые проблемы, а разнообразие инструментов и новых технических методов их решения настолько велико, что может сбить с толку.

С чего же начать? Как получить наибольшую отдачу на вложенный капитал?

- Люди – вот главный компонент сборочного конвейера, наряду с технологиями их нужно рассматривать как часть поверхности атаки.
- В быстро изменяющейся среде сканирование на уязвимости должно производиться чуть ли не непрерывно. Сканирование раз в год или раз в квартал ничего не даст.
- Укрепление безопасности должно встраиваться в процессы подготовки и конфигурирования системы, а не добавляться задним числом.
- Автоматизация подготовки системы и управления конфигурацией с помощью таких программируемых средств, как Ansible или Chef, контейнеров типа Docker, программ создания образов типа Packer и технологий облачных шаблонов, должна лечь в основу стратегии безопасной эксплуатации.
- Эти технологии позволят гарантировать, что каждая система настроена правильно и единообразно – в среде разработки, тестирования и эксплуатации. Одни и те же изменения можно быстро вносить в сотни и даже тысячи систем, причем эта техника безопасна, тестопригодна и обеспечивает полную прозрачность и прослеживаемость всех изменений. Вы можете автоматически определить и контролировать политики укрепления и соответствия нормативным требованиям.
- Реализовав управление конфигурацией в виде кода и используя одни и те же инструменты и сборочные конвейеры в средах разработки и эксплуатации, вы сможете применять одни и те же средства контроля безопасности ко всем изменениям, включая

инспекцию кода, статический анализ кода и автоматизированное тестирование.

- Автоматизированный сборочный конвейер представляет заманчивую мишень для атаки. Его следует рассматривать как часть производственной среды и обращаться с ним так же, как с самыми ценными в отношении безопасности системами.
- Секреты должны храниться в секрете. Закрытые ключи, маркеры API и прочие учетные данные необходимы инструментам и зачастую должны использоваться по разные стороны границы доверия. Не храните секреты в скриптах, конфигурационных файлах или исходном коде. Поручите их заботам безопасного диспетчера секретов.
- Если информация о безопасности включена в циклы обратной связи мониторинга приложений, то вопросы безопасности становятся более наглядными для разработчиков и эксплуатационников.
- Готовьтесь к инцидентам заранее – это касается серьезных проблем эксплуатации и безопасности. Беда обязательно произойдет. Позаботьтесь о том, чтобы эксплуатационники и разработчики понимали, что им делать в экстренной ситуации.
- Практика, практика и еще раз практика. Регулярно проводите учения, игры с участием команд красных и синих и другие упражнения, чтобы отработать различные проблемы и накачать организационные мускулы.
- Посмертный анализ, т. е. разбор уже случившихся эксплуатационных отказов и инцидентов безопасности, дает команде возможность учиться на собственных ошибках. Если он организован правильно – открыто, искренне, не ставя целью найти виноватых, то послужит выстраиванию доверительных отношений между командами.

Глава 14

Соответствие нормативным требованиям

Соответствие нормативным требованиям – важная опора безопасности. В некоторых организациях это стержень программы безопасности, определяющий, как следует вносить изменения, как и когда проводить инспекции и выполнять тестирование, какие уязвимости исправлять, а какие нет и как должна быть построена совместная работа разработчиков, эксплуатационников и безопасников.

Такие нормативно-правовые требования, как PCI-DSS, HIPAA и HITECH, SOX, GLBA, SEC и MiFID, 23 NYCRR 500, правила безопасности FDA для проверки программного обеспечения, FERC и NERC, FERPA, FedRAMP, FFIEC и FISMA, COBIT и HiTRUST, ISO/IEC 27001, а также стандарты NIST и CIS, которым должны следовать организации, отвечающие этим требованиям, – все они определяют правила, рекомендации и ограничения на проектирование и гарантии системы, обучение и информирование персонала, управление рисками и уязвимостями, контроль изменений и управление релизами, аудит и хранение данных, мониторинг сети и системы и деятельность ИТ-департамента.

Если вы работаете в регулируемой среде, то должны понимать, как соответствие нормативным требованиям отражается на программе безопасности и как учесть эти требования при разработке и эксплуатации. Ранее в этой книге мы побуждали вас думать как противник, уделять повышенное внимание угрозам и уязвимостям и помнить о криминалистической экспертизе. В этой главе мы попробуем думать как аудитор, смотреть на систему с его точки зрения, акцентируя внимание на рисках, средствах контроля и доказательствах.

PHI и PII

В этой и других главах речь идет о различных категориях персональных или частных данных. Некоторые типы данных настолько важны, что для них имеются специальные акронимы, в т. ч. PHI и PII.

PHI

Personal (или Protected) Health Information: любая информация, относящаяся к состоянию физического или душевного здоровья человека, включая состояние больного, оказание медицинской помощи или оплату медицинских услуг.

PII

Personally Identifiable Information (Идентифицирующие персональные данные): любая информация, используемая для установления личности человека, включая имя, дату рождения, биометрические данные, номер социального страхования, IP-адрес и данные о кредитных картах.

Соответствие нормативным требованиям и безопасность

Но прежде чем перейти к тому, что соответствие нормативным требованиям значит на практике, нам кажется важным сказать несколько слов о различиях между соответствием нормативным требованиям и безопасностью.

Безопасность – это процесс разработки и внедрения физических, технических и административных средств контроля для защиты конфиденциальности, целостности и доступности информации. Это принятие мер, гарантирующих, что к системе и данным имеют доступ только лица, имеющие на это право, что хранимые данные не подвергаются несанкционированным изменениям и что заказчики могут использовать их в тот момент и таким образом, как им необходимо.

Соответствие нормативным требованиям – это немного другое. Подразумевается, что мы понимаем, что такое безопасность, готовы делать всё для ее обеспечения и соблюдать требования, – только тогда нам разрешено работать. Безопасность – это политики и процессы, призванные защитить всю информацию и системы. Соответствие нормативным требованиям – демонстрация того, что мы умеем делать это

правильно, т. е. соблюдаем условия, при которых разрешено работать в определенных областях и хранить определенные виды информации.

Стандарты и нормативно-правовые требования, такие как PCI, HIPAA, GLBA, законы о неприкосновенности частной жизни, например действующие в штате Калифорния, а также в Германии и Италии, закон PIPEDA (о защите персональных данных и электронных документах) в Канаде, новый Общий регламент ЕС по защите персональных данных (GDPR), – все эти документы определяют законодательные требования и конкретные обязанности по защите персональной и частной информации, регулируют вопрос о том, где эта информация может, а где не может храниться и что надлежит делать в случае невыполнения этих обязанностей.

Соответствие нормативным требованиям преследует следующие цели.

1. Подтвердить, что вы понимаете, что такое ответственный подход к работе. Для этого определяется минимальный набор требований к безопасности и конфиденциальности, направленных на соблюдение общепризнанных стандартов или передовых практик управления рисками, управления изменениями, разработки ПО, управления уязвимостями, управления удостоверениями и доступом, шифрования данных и функционирования ИТ-департамента.
2. Потребовать от вас доказательств ответственной работы: что сверху донизу установлены и своевременно модифицируются определенные политики, что отслеживаются все изменения, что регулярно производятся сканирование и тестирование на уязвимость, что периодически выполняются оценка и аудит с целью проверки эффективности средств контроля.
3. Наказать вашу организацию за несоблюдение требований посредством наложения штрафов и других санкций. Размер штрафа может достигать десятков и сотен тысяч долларов в месяц для банков, не соблюдающих требования стандарта PCI DSS, и нескольких миллионов долларов для медицинских учреждений в случае кражи данных (по закону HIPAA[1]).

В этой главе мы рассмотрим, как соответствие нормативным требованиям влияет на следующие аспекты:

- проектирование и управление требованиями (требования о предписанном аудите, отчетности и прослеживаемости);

[1] Закон об ответственности и переносе данных о страховании здоровья граждан.

- тестирование и инспекция (внутренний контроль и оценка, валидация и верификация);
- передача изменений ПО в производственную систему (управление изменениями и релизами);
- надзор за разработчиками, особенно за их вмешательством в производственную систему (разделение обязанностей);
- защита данных (конфиденциальность, контроль доступа, шифрование и сроки хранения данных);
- документация (как свести объем необходимой бумажной работы к минимуму).

Заявление об отказе от ответственности

В этой книге мы рассматриваем различные системы соответствия нормативным требованиям, в т. ч. стандарт PCI DSS, стремясь показать, как они функционируют и что означают на техническом уровне. И описываем, как эффективно и не жертвуя продуктивностью обеспечить соответствие в гибкой среде.

Но мы не претендуем на исчерпывающее изложение какого-то конкретного нормативного документа. Мы не даем юридических консультаций. Вы должны сами разобраться, к чему вас обязывают нормативные требования, на какие системы, данные и виды деятельности они распространяются, а затем в сотрудничестве с лицами, отвечающими за соответствие нормативным требованиям, безопасность и управление рисками, а также с аудиторами и руководством выработать решение, которое удовлетворяло бы все стороны.

Для вашей организации обязанность соответствовать нормативным требованиям может вытекать из законодательства напрямую или, если вы поставщик услуг, по настоянию клиентов. Некоторые организации подпадают под действие сразу нескольких законов с перекрывающимися и различающимися требованиями. Например, публичная компания, предоставляющая онлайновые медицинские услуги, может подпадать под действие законов SOX[2], HIPAA, PCI DSS (если клиенты могут расплачиваться кредитными или дебетовыми картами) и различных законов о конфиденциальности данных. Вам придется найти решение, удовлетворяющее всем требованиям этих законов одновременно и независимо.

[2] Закон Сарбейнса-Оксли.

Различные подходы к законодательному регулированию

В некоторых нормативных требованиях довольно четко указано, что именно вы должны сделать. В других же дело обстоит похуже. Дело в том, что есть два принципиально разных подхода к законодательному регулированию.

Предписывающий: основанный на правилах

Набор очень конкретных предписаний, указывающих, что вы должны или можете делать, а что не должны или не можете. В таких нормативно-правовых документах описаны определенные средства и процедуры контроля, перечислены риски и предписано, что вы – и аудиторы – обязаны делать, когда и с какой частотой и какие доказательства следует хранить. Как правило, такие требования можно оформить в виде контрольных списков.

Примерами требований на основе правил являются законы PCI DSS, FedRAMP, FISMA и различные нормы в сфере технической безопасности.

Описательный: на основе результатов

В таких законах описывается управление безопасностью или рисками либо операционные цели или возлагаемые на организацию правовые обязанности, но ничего не говорится о том, что нужно сделать для их выполнения. Обычно это несколько детальных правил, касающихся обязательной отчетности и оценок, но во всем остальном организация свободна в выборе подхода, при условии что средства контроля признаны «адекватными», «эффективными» и «разумными».

Теоретически это открывает возможность подойти к удовлетворению нормативно-правовых требований более рационально и инновационно. Но одновременно это означает, что факт соответствия требованиям труднее установить со всей определенностью. У аудиторов также больше пространства для маневра в решении о том, являются ли ваши программы и средства контроля достаточными или удовлетворительными, а размер штрафа зависит от того, насколько неадекватными, неэффективными или неразумными они признаны. Вам придется защищать свой подход и ясно продемонстрировать, как именно он обеспечивает выполнение нормативно-правовых требований.

Именно поэтому так много организаций опираются в работе на тяжеловесные системы управления и контроля типа COBIT[3] (*http://www.isaca.org/cobit/pages/default.aspx*) и ITIL[4] (*https://en.wikipedia.org/wiki/ITIL*). Следование стандартизованной, общепринятой и полной модели управления и контроля увеличивает затраты и задержки, но снижает риск признания вашей программы неадекватной аудиторами или следователями. А это, в свою очередь, снижает штрафы и ответственность в случае какого-либо происшествия.

Примерами законов на основе результатов являются HIPAA, FDA QSR[5], SOX 404 и система надзора за целостностью и соблюдением требований SEC[6].

Рассмотрим два примера, иллюстрирующих фундаментальные различия между этими подходами к законодательному регулированию.

PCI DSS: подход на основе правил

Стандарт безопасности данных в индустрии платежных карт (PCI DSS – *https://www.pcisecuritystandards.org/*) – это межотраслевая стандартная практика, которую вынуждены принимать во внимание многие команды, поскольку она относится к системам, где прямо или косвенно (с помощью сторонней службы) обрабатываются платежи, осуществляемые по кредитной или дебетовой карте.

Вместо расплывчатых юридических фраз о демонстрации надлежащего отношения в PCI DSS изложены определенные и в большинстве своем конкретные требования и обязанности: вы должны делать одно, не должны делать другое и должны доказывать это вот такими способами. Стандарт достаточно четко описывает, какими рисками следует управлять и как именно, какие данные следует защищать и как, все виды тестирования и инспекций и сроки их проведения.

PCI DSS состоит из ряда руководств, дополнительных документов, извещений, контрольных списков и часто задаваемых вопросов (FAQ); но основная часть закона относительно невелика. Хотя удовольствия это чтение не доставляет (с юридическими документами всегда так), следовать стандарту нетрудно. Имеется даже краткое справочное руко-

[3] Control Objectives for Information and Related Technologies – задачи управления для информационных и смежных технологий.

[4] IT Infrastructure Library – библиотека инфраструктуры информационных технологий.

[5] Регламент системы контроля качества Управления по надзору за пищевыми продуктами и медикаментами.

[6] Securities and Exchange Commission – Комиссия по ценным бумагам и биржам.

водство (https://www.pcisecuritystandards.org/documents/PCIDSS_QRGv3_2.pdf) примерно на 40 страницах, в котором очень доходчиво в общих чертах объяснено, что вам надлежит делать.

В программе безопасности, совместимой с PCI, должно быть 12 разделов. Хотя можно спорить – и таки спорят – об эффективности и полноте отдельных средств контроля, PCI DSS предлагает вполне приемлемую модель программы безопасности, применимую даже в случаях, не имеющих никакого отношения к обработке данных кредитных карт. В ней предвосхищается много рисков и проблем, которые должны быть учтены в любой онлайновой системе, где хранятся важные данные, и описывается структурированный подход к управлению этими рисками. Нужно просто заменить слова «данные держателя карты» теми секретными или конфиденциальным данными, которыми вы должны управлять.

Проект безопасной сети

Использовать брандмауэры и сегментацию сети. Периодически пересматривать конфигурации и следить за тем, чтобы все изменения конфигурации были протестированы. Построить карту всех соединений и потоков данных.

Укрепление конфигурации

Выявить все участвующие системы и убедиться, что они безопасно сконфигурированы в соответствии с «принятым в отрасли определением» (например, в смысле Центра интернет-безопасности – CIS), обращая особое внимание на изменение заводских учетных данных.

Защита секретных данных (данных держателя карты) в процессе хранения

Ограничить доступ к системе хранения. Использовать одностороннее хеширование, криптостойкое шифрование, преобразование в маркеры, маскирование или усечение. Конкретные указания о том, какие данные можно, а какие нельзя хранить.

Безопасная передача данных

Использовать стойкое шифрование и криптографические протоколы при передаче секретных данных по открытым сетям.

Защита систем от вредоносного ПО

Использовать антивирусные программы.

Разработка и сопровождение безопасных систем и приложений

Этот вопрос будет подробно рассмотрен ниже.

Ограничение доступа к данным держателя карты

Разрешать доступ только лицам, которым это нужно для выполнения функциональных обязанностей.

Требования к идентификации и аутентификации пользователей

Включая описание условий, при которых необходима многофакторная аутентификация.

Физическая безопасность

Ограничение физического доступа к центрам обработки данных, участкам работы с секретными данными, резервным носителям. Учет каждого факта доступа.

Учет и мониторинг всех фактов доступа к сети и данным держателей карт

Аудит, протоколирование. Конкретные требования к аудиту операций, доступу к защищенным данным и срокам хранения контрольных журналов.

Регулярное сканирование и тестирование безопасности

Сканирование на наличие беспроводных точек доступа и уязвимостей внутренних и внешних сетей, тестирование на проникновение, обнаружение и предотвращение вторжений и контроль с целью обнаружения изменений.

Оформление документов

Политики контроля, оценки рисков и информационной безопасности, доказывающие, что руководство и все остальные понимают требования и то, как они соблюдаются. Все операционные процедуры и политики для каждого из вышеизложенных требований должны быть «документированы, известны всем заинтересованным сторонам и применяться на практике».

Сюда входит и регулярное обучение разработчиков и эксплуатационников основам безопасности и соответствия нормативным требованиям.

Соответствующая стандарту организация должна выполнять все эти требования и регулярно проходить освидетельствование.

Раздел 6 непосредственно применим к разработке программного обеспечения. В нем рассматриваются сбор требований, проектирование, кодирование, тестирование, инспекция и внедрение.

6.1: управление уязвимостями

Выявлять уязвимости, определять их приоритеты на основе рисков и устранять. Вопрос об управлении уязвимостями мы подробно рассматривали в главе 6.

6.2: применение исправлений

Следить за применением всех выпущенных исправлений стороннего ПО. Критические исправления должны быть установлены в течение месяца с момента выпуска.

6.3: безопасное кодирование в соответствии с «лучшими отраслевыми практиками»

Сюда входят конкретные требования, например: удаление всех тестовых учетных записей и учетных данных до передачи приложений в эксплуатацию, инспекция внесенных в код изменений на наличие уязвимостей (вручную или с применением автоматизированных средств), одобрение результатов инспекции руководством до выпуска изменений.

6.4: управление изменениями

Четкое разделение обязанностей между группой эксплуатации и группой разработки и тестирования, доказательства проведения тестирования безопасности, анализа последствий и планирования отката, документально подтвержденное одобрение изменения уполномоченными лицами.

6.5: обучение команды разработчиков безопасному кодированию

Не реже раза в год с выдачей наставления по безопасному кодированию.

6.6: защита веб-приложений

Проводить оценку уязвимости приложений не реже раза в год или после выпуска существенных изменений. Вместо этого разрешается защищать систему посредством внедрения оборонительного решения, блокирующего атаки на этапе выполнения (например, брандмауэра на уровне веб-приложений).

6.7: оформление документов

Для доказательства всего вышеперечисленного.

Это «всё необходимое» для соответствия стандарту PCI с точки зрения разработчика, если не считать корректной реализации средств контроля данных держателя кредитной карты из раздела 3 и ограничений доступа к этим данным из раздела 7. Но, конечно же, дьявол кроется в деталях.

Надзор за целостностью и соблюдением требований: подход на основе результатов

Сравним это с Reg SCI (Systems Compliance and Integrity – надзор за целостностью и соблюдением требований) – одним из многочисленных законодательных актов, регулирующих деятельность фондовых бирж и других финансовых организаций в США. Он направлен на обеспечение функциональных возможностей, целостности, отказоустойчивости, доступности и безопасности финансовых систем.

Документ Reg SCI состоит из 743 страниц, написанных на юридическом жаргоне и содержащих в основном описание того, как составлялся проект этого закона. Суть закона изложена примерно на 20 страницах, начинающихся где-то со страницы 700. Требуется, чтобы организация разработала и поддерживала «оформленные в письменном виде разумно составленные политики и процедуры», которые гарантируют, что разработка и эксплуатация систем будут вестись безопасно и что системы удовлетворяют всем требованиям законодательства.

Эти политики и процедуры должны распространяться на проектирование, разработку и тестирование системы, планирование и тестирование вычислительных мощностей, планирование и тестирование непрерывности функционирования, управление изменениями, проектирование и управление сетью, эксплуатацию и мониторинг системы, реагирование на инциденты, физическую и информационную безопасность, а также на передачу любой из вышеперечисленных обязанностей на сторону. В Reg SCI также сформулированы обязанности по отчетности перед SEC и по ежегодному проведению аудита.

Политики и процедуры можно рассматривать как «разумно составленные», если они согласуются с «практиками информационных технологий, широко доступными ИТ-специалистам в финансовом секторе и выпущенными признанной организацией», например правительством США.

Список общепризнанных передовых практик включает (почти для любого сектора) стандарт NIST SP 800-53r4 (*http://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-53r4.pdf*), представляющий собой 462-страничный документ, в котором описано 218 конкретных мер по контролю над управлением рисками, безопасностью и конфиденциальностью и их применение к федеральным информационным системам США (на основе профиля риска системы). Меры, относящиеся к разработке ПО, описаны в различных разделах, например: создание системы (в основном), управление конфигурацией, оценка рисков, целостность системы и информации, информирование и обучение, конфиденциальность, оценка безопаснос-

ти, авторизация и т. д. В каждой из этих мер имеются ссылки на другие меры, или другие документы NIST, либо иные публикации и стандарты.

В общем, эффект получается ошеломительным. Решение о том, что нужно делать обязательно, что желательно, что допустимо и как все это доказывать, возлагается на организацию и ее аудиторов.

В документе NIST SP 800-53r4 не сказано, что запрещается создавать и поставлять системы с применением гибких и бережливых методик, но сам по себе он дает ярчайший пример всего того, против чего эти методики восстают: правовые политики и горы бумаг, призванные осчастливить бюрократа, но не переводимые на язык конкретных, поддающихся измерению требований, которым можно удовлетворить.

SEC ясно дает понять, что список стандартов следует рассматривать как директивные документы. Ваши средства контроля и программы будут оцениваться на соответствие этим стандартам. Если вы решитесь не следовать им, то на вас ляжет обязанность доказывать, что ваша система средств контроля «разумно спроектирована».

Какой подход лучше?

Хотя подход Reg SCI, основанный на результатах, оставляет организации некоторую гибкость и свободу принимать решения о том, как удовлетворить требованиям, предписывающие контрольные списки PCI гораздо яснее: вы знаете, что должны делать и по каким параметрам вас будут оценивать. Если вы не выполнили требования PCI, то, вероятно, имеете четкое представление, почему так произошло и что с этим делать. С другой стороны, трудно заставить аудиторов и оценщиков оторвать взгляд от этих детальных списков и помочь вам понять, действительно ли ваша программа делает организацию безопасной.

Управление рисками и соответствие нормативным требованиям

Системы правового регулирования и контроля строятся на основе рисков: их назначение – помочь организации и, что еще важнее, клиентам и обществу защититься от рисков безопасности и конфиденциальности.

Управление рисками при гибкой разработке рассматривалось в главе 7. А здесь мы рассмотрим, как управление рисками влияет на решения, связанные с соблюдением нормативных требований и применением средств контроля, с точки зрения соответствия этим требованиям.

Регулирующие органы требуют, чтобы в организации существовала активная программа управления рисками, нацеленная на риски, относящиеся к безопасности и конфиденциальности, и связанные с ними эксплуатационные и технические риски. Тем самым гарантируется, что вы не просто пытаетесь выполнить базовые пункты контрольного списка, а что руководство, равно как и все сотрудники, работающие над проектами и в отделе эксплуатации, осознанно стремится заблаговременно выявлять быстро изменяющиеся риски и угрозы, осмыслять их и принимать надлежащие меры.

Аудиторов будет интересовать формальное заявление о политике, в котором объясняется, почему управление рисками представляет важность для организации, и приводится персональное распределение обязанностей по управлению рисками. Их также будут интересовать свидетельства того, как программа управления рисками проводится в жизнь, они захотят убедиться, что средства контроля и процедуры эксплуатации и безопасности регулярно подвергаются анализу и пересматриваются с учетом изменяющихся рисков.

Например, стандарт PCI DSS 12.1.2 требует, чтобы организация не реже раза в год выполняла формальную оценку рисков с целью анализа угроз и уязвимостей, а также изменений в среде. Результатом должна быть уверенность в том, что действующие средства контроля и программы эффективно справляются с этими рисками.

Программа управления рисками должна включать следующие ежедневные действия:

- использование информации об угрозах для назначения приоритетов применению исправлений, тестированию, инспекциям и обучению основам безопасности – мы рассматривали этот вопрос в главе 8;
- использование преимуществ гибких инспекций и ретроспективного анализа, включение в рассмотрение новых рисков – безопасности, эксплуатационных, соответствия нормативным требованиям – и мер борьбы с ними;
- проведение посмертного анализа после сбоев и взломов, а также по результатам тестирования на проникновение и других оценок с целью выявления слабых мест в средствах контроля и возможности для улучшения;
- информирование членов команды о типичных рисках и о действиях по их предотвращению, например рисков, перечисленных в списке OWASP Top 10.

Список рисков OWASP Top 10

Открытый проект обеспечения безопасности веб-приложений (Open Web Application Security Project – OWASP), сообщество специалистов по безопасности приложений, регулярно публикует список 10 главных рисков, с которыми сталкиваются веб-приложения: OWASP Top 10.

Для каждого риска приводятся примеры уязвимостей и атак, а также инструкция о том, как проверить наличие проблемы и что сделать для ее предотвращения.

Список OWASP Top 10 часто упоминается в правовых документах, например PCI DSS, как важнейший инструмент управления рисками. Ожидается, что в программу обучения разработчиков основам безопасности войдут риски из этого списка. Средства сканирования на безопасность отмечают риски из списка OWASP Top 10, а тестировщики проникновения обязательно включают их в отчеты об обнаруженных уязвимостях.

Прослеживаемость изменений

Фундаментальное требование в системах соответствия нормативным требованиям и контроля – способность доказать, что все изменения надлежащим образом авторизованы и учтены и что внесение неавторизованных изменений эффективно предотвращается. Это подразумевает прослеживание изменений от момента запроса до момента поставки и доказательство того, что все необходимые тесты и проверки были выполнены:

- Когда было произведено изменение?
- Кем было произведено изменение?
- Кто дал разрешение?
- Кто инспектировал?
- Производилось ли тестирование изменения?
- Прошли ли тесты?
- Как и когда изменение было развернуто в производственной среде?

Все это может сопровождаться кучей бумажной писанины. Но, как мы объясним ниже в этой главе, можно обойтись вообще без писанины, положившись на существующие рабочие процессы и комплекты инструментов.

Конфиденциальность данных

HIPAA, PCI DSS и GLBA[7] – примеры правовых норм, относящихся к защите персональной или конфиденциальной информации: соответственно сведений о состоянии здоровья, данных о держателе кредитной карты и персональных финансовых данных клиентов.

Эти нормы, а также вышеупомянутые правительственные законы о конфиденциальности определяют, какая информация должна быть классифицирована как конфиденциальная или секретная, содержат правила и ограничения, распространяющиеся на защиту этих данных, и описывают, что необходимо для доказательства того, что они действительно защищены (включая заверения, аудит и хранение данных в течение регламентированного срока).

Не пытайтесь ничего делать без консультации с юристом

Мы предлагаем лишь общие рекомендации. Проконсультируйтесь со своим уполномоченным по конфиденциальности или соответствию нормативным требованиям либо с юрисконсультом, чтобы понять, какие обязанности возлагаются на вашу организацию в части обеспечения конфиденциальности данных.

Перечислим основные действия по обеспечению конфиденциальности данных.

1. С самого начала ознакомить владельца продукта и команду с ясными, утвержденными инструкциями и правилами. При необходимости организовать обучение, чтобы все отчетливо понимали риски конфиденциальности данных, соответствующие правила и ограничения.
   Проверьте, что пункт о соответствии нормативным требованиям включен в принятое командой определение готовности. Должно быть предусмотрено документирование операций или иные доказательства, которые можно предъявить аудитору.
2. Создать матрицу всей конфиденциальной или секретной информации, в которой указано, кто владеет данными и кому разрешен к ним доступ (создание, изменение, чтение, удаление). Сделать эту матрицу легкодоступной команде и поддерживать ее в актуальном состоянии.

[7] Закон Грэма-Лича-Блайли о финансовой модернизации.

3. Понять и получить подтверждение своего понимания ограничений на местонахождение данных, особенно если система работает в облаке. Некоторые законы о конфиденциальности требуют, чтобы к защищенным данным не было общего доступа или чтобы они хранились в определенной юрисдикции.
4. Графически изобразить, как защищенные данные собираются, хранятся, обновляются и удаляются, как к ним осуществляется общий доступ и как организованы ссылки на них. Использовать для этой цели такие же диаграммы потоков данных, как при моделировании угроз. Включить временные и рабочие копии, кеши, отчеты, электронные таблицы и резервные копии.
5. Гарантировать, что защищенные данные шифруются при хранении и передаче с применением общепризнанного стандартного алгоритма шифрования, маскируются, заменяются псевдонимами или безопасным маркером.
6. Написать истории для учета следующих требований: согласие на обработку конфиденциальных данных (которое подразумевается по умолчанию, но может быть отозвано), право на забвение, предоставление уведомления, а также обязательный аудит, протоколирование, хранение данных и отчетность перед регулирующими органами.
7. Для любой истории, в которой речь идет о сборе, хранении, обновлении, удалении, общем доступе и ссылках на защищенные данные, внимательно сопоставить критерий приемки с нормативными требованиями и пометить эти истории для дополнительной правовой экспертизы или проверки на соответствие нормативным требованиям; только после этого истории можно считать готовыми.
8. Регулярно проводить сканирование контента, баз данных и файлов, включая журналы, тестовые данные и рабочие каталоги в производственной системе на наличие защищенных данных, которые не зашифрованы, не замаскированы и не заменены маркерами.
9. Проводить сканирование кода на наличие ссылок на защищенные данные и вызовов функций, реализующих криптографические операции, маскирование или преобразование в маркеры. Уведомлять об изменениях такого кода, чтобы их можно было подвергнуть инспекции.

10. Написать истории, относящиеся к соответствию нормативным требованиям, для аудита, внешней инспекции, тестирования на проникновение и других регламентных проверок, чтобы их можно было запланировать и включить в график.
11. Разработать план реагирования на инциденты, включая взлом и разглашение персональных данных, и регулярно проверять способность к действиям по этому плану (как описано в главе 13).
12. Сохранять доказательства того, что все это делается надлежащим образом, включая комментарии к инспекциям, приемочные тесты, архивные записи об управлении изменениями, журналы доступа к системе и результаты аудита.
13. Регулярно проводить с командой совещания по вопросу последовательного соблюдения всех этих инструкций и совершения предписанных практических действий.

Шпаргалки OWASP по криптографии

Чтобы помочь вам правильно применять криптографические функции, специалисты из OWASP составили следующие шпаргалки.

1. О том, как безопасно хранить данные, см. «Шпаргалку по криптостойкому хранению» (*https://www.owasp.org/index.php/Cryptographic_Storage_Cheat_Sheet*).
2. Поскольку безопасное обращение с паролями – отдельная проблема, существует специальная «Шпаргалка по хранению паролей» (*https://www.owasp.org/index.php/Password_Storage_Cheat_Sheet*).
3. О безопасной передаче данных см. «Шпаргалку по защите на транспортном уровне» (*https://www.owasp.org/index.php/Transport_Layer_Protection_Cheat_Sheet*).

Повторим еще раз: не нужно реализовывать собственный алгоритм шифрования. Если у вас есть вопросы по криптографии, обратитесь к специалисту.

Как соответствовать нормативным требованиям, сохраняя гибкость

Как можно адаптировать гибкую методику или DevOps к ограничениям, действующим в строго регулируемых средах?

Действуйте инициативно. Не стоит подходить к соответствию нормативным требованиям пассивно, ждать, когда кто-нибудь разъяснит вам, что и как делать, или, хуже того, сообщит, что вы не сделали чего-то, что должны были сделать. Попытайтесь понять цели и задачи правовых норм, выпишите четко определенные твердые правила, которые в них содержатся, и подумайте, как встроить соблюдение нормативных требований в свои рабочие процессы и особенно в автоматизацию.

Поразмыслите, как задействовать хорошо зарекомендовавшие себя технические практики и автоматизацию, чтобы и соблюсти нормативные требования, и удовлетворить собственные потребности. Если вы сможете гарантировать – и доказать, – что любое изменение проходит по автоматизированному конвейеру после инспекции и записи в репозиторий, то у вас будет сильная позиция в разговоре с аудиторами, а также действенный инструмент для расследования и даже предотвращения эксплуатационных проблем.

Соответствие нормативным требованиям иногда требует твердого «нет»

На протяжении всей книги мы подчеркиванием, как важно, чтобы группа безопасности искала пути содействовать, а не мешать работе: делать все, чтобы разработчики и эксплуатационники могли заниматься своим делом безопасно, а не создавать им препятствия.

Но иногда во имя соблюдения нормативных требований приходится твердо заявлять «Нет, этого мы делать не можем» и отвергать изменение или запрещать развертывание. После каждого такого случая созовите совещание с командой и руководством, чтобы найти пути предотвращения подобной ситуации в будущем, встроив соответствие нормативным требованиям в проект или в рабочие процессы команды.

Истории о соответствии и соответствие в историях

Обязательное тестирование соответствия нормативным требованиям, инспекции и другие пункты контрольных списков должны быть включены в принятое командой определение готовности историй, спринтов и релизов:

- какие инспекции необходимо провести и кто это должен делать;
- какое тестирование следует выполнить;
- какую документацию нужно обновить;

- какие необходимо представить доказательства того, что все это сделано.

В главе 5 мы говорили о том, как писать *истории, касающиеся безопасности*, которые помогали бы реализации требований к безопасности или средств контроля безопасности. Точно так же может возникнуть необходимость в написании *историй, касающихся соответствия нормативным требованиям*, где явно описывались бы действия, которые следует выполнить и доказать. Эти истории отделены от критериев соответствия, которые могут быть частью конкретных пользовательских историй.

Истории, касающиеся соответствия, могут играть роль напоминаний о средствах контроля, которые должны быть разработаны и внедрены заранее (включая обязательное обучение членов команды), а также о таких оценках безопасности, как аудит и тесты проникновения.

Применяйте такие инструменты, как osquery (*https://osquery.io/*) и InSpec (*https://www.inspec.io/*), для написания онлайновых тестов соответствия нормативным требованиям и для обеспечения прослеживаемости до конкретных правовых норм, областей применимости правил или средств контроля.

Больше кода, меньше писанины

Гибкие и бережливые команды стремятся минимизировать потери времени, сведя к минимуму документацию: работающий код – лучшая документация.

Но когда речь заходит о соответствии нормативным требованиям, совсем без бумажной писанины не обойтись.

И какой же минимальный объем документации вы обязаны представить для соблюдения требований регулирующих и контролирующих органов? И что можно позаимствовать из уже создаваемых артефактов?

Необходимо иметь документированные политики и инструкции по безопасности и управлению рисками, документы по постановке учета и отчетности и ясное распределение обязанностей. И правовую защиту типа соглашений о конфиденциальности и неразглашении. Все это необходимо раздать сотрудникам, получить обратно в подписанном виде, подшить и регулярно пересматривать и обновлять.

Необходимо также хранить требования в машиночитаемом виде, чтобы их можно было подвергнуть аудиту и проследить изменения: нельзя просто записывать истории на карточках или клейких бумажках, а затем выбрасывать по мере выполнения.

Но большинство процедур соответствия нормативным требованиям и детальных контрольных списков можно – и должно – перенести из документов в рабочие процессы разработчиков и эксплуатационников, а там, где возможно, – в код. Их следует подкрепить правилами автоматизированных конвейеров сборки и развертывания, автоматизированных тестов и проверок и воспользоваться создаваемыми при этом контрольными журналами.

Не пишите, если не собираетесь исполнять

Если что-то записано в политике, то все должны быть готовы это выполнять – и доказывать, что выполнили надлежащим образом. Не копируйте слепо шаблон политики, найденный в Интернете (например, на странице *https://www.sans.org/security-resources/policies*), не убедившись, что он подходит вашей организации. Не вписывайте в политику пункт просто потому, что вам понравилось, как он звучит. Если организация не соблюдает собственные политики, то аудиторы смогут интерпретировать это как доказательство существенной неэффективности контроля, особенно если ваша среда регулируется правовыми нормами на основе результатов. За это сотрудников можно уволить, а на руководителей возложить персональную ответственность.

К политикам следует относиться серьезно. Считайте их обещаниями – а вы ведь всегда держите свои обещания, правда?

Для этого понадобится свести руководство, службу надзора за соответствием нормативным требованиям, службу внутреннего аудита, отдел руководства программой (PMO) и группу безопасности с разработчиками и эксплуатационниками.

Все эти заинтересованные стороны должны в самом начале совместно определить правила соответствия и рабочие процессы контроля, и любые их изменения должны быть формально утверждены руководством и задокументированы, например на совещании консультативного комитета по изменениям. Разработчики и эксплуатационники должны проработать процедуры и контрольные списки со службой надзора за соответствием нормативным требованиям, группой безопасности и отделом руководства программой, выделить ключевые средства контроля и согласовать простые способы их автоматизации. Руководство должно понимать, как различные риски – эксплуатационные, безопасности и прочие – будут управляться и контролироваться с помощью этих автоматизированных рабочих процессов, тестов и конвейеров.

Использование комплекта инструментов DevOps Audit Defense

Многие приведенные в этой главе идеи касательно автоматизации соответствия нормативным требованиям основаны на комплекте инструментов DevOps Audit Defense Toolkit (*https://itrevolution.com/devops-audit-defense-toolkit/*) – свободном, принадлежащем сообществу каркасе, который был написан специалистами по соответствию нормативным требованиям и ИТ-контролю Джеймсом Делючиа IV (James DeLuccia IV), Джеффом Гэллимором (Jeff Gallimore), Джином Кимом (Gene Kim) и Байроном Миллером (Byron Miller).

В этом комплекте инструментов гибкие методики и DevOps, включая непрерывную поставку и непрерывное развертывание, рассматриваются через призму соответствия нормативным требованиям с учетом реального опыта работы в регулируемых средах. Он написан в форме анализа ситуации с соответствием нормативным требованиям в воображаемой организации («Parts Unlimited», той же компании, что была описана в романе Джина Кима «Проект Феникс» (*http://itrevolution.com/books/phoenix-project-devops-book/*), посвященном DevOps), на которую распространяются нормы SOX, PCI DSS, SOC 2 и FedRAMP (Федеральная программа управления рисками и авторизацией). История начинается с изложения типичных эксплуатационных рисков и стратегий контроля, а затем обсуждается, как внедрить требуемые средства контроля, воспользовавшись преимуществами применяемых в гибких методиках и DevOps практик, рабочих процессов и автоматизации.

Комплект инструментов показывает, как объяснить и защитить перед аудиторами гибкие методики и DevOps и такие практики, как непрерывное развертывание. Его можно использовать как дорожную карту для реализации собственной программы соответствия, адаптированную к профилю рисков, нормативным требованиям, деловой культуре и степени зрелости процессов, характерным для вашей организации.

Рассмотрим, как решить в виде кода некоторые конкретные проблемы соответствия нормативным требованиям.

Прослеживаемость и гарантии непрерывной поставки

В главе 11 мы объяснили, как работает автоматизированный сборочный конвейер и как применять инструменты и рабочие процессы непрерывной интеграции и поставки для быстрого и безопасного тестирования и развертывания изменений.

Теперь мы воспользуемся всем этим для соблюдения нормативных требований – мы будем прослеживать каждое изменение от момен-

та его запроса до момента поставки с целью показать, что изменения обрабатываются единообразно и надлежащим образом. Перечислим шаги, которые гибкая команда может предпринять, чтобы доказать прослеживаемость и гарантии в регулируемой среде.

1. Мы уже видели, что гибкая команда не отталкивается от детального технического задания, а записывает требования в виде простых и конкретных *пользовательских историй* на каталожных карточках или липких бумажках, которые выбрасываются, когда история реализована. Однако, чтобы соответствовать нормативным требованиям, необходимо хранить сведения о каждой функции и каждом изменении: кто запросил, кто утвердил и согласованные критерии приемки (т. е. условия, при которых заказчик будет удовлетворен).
   Конечно, можно было бы записать все это, например, в электронную таблицу, но к услугам современных команд имеются инструменты управления проектом или трекеры историй, например Rally Version One или Jira, которые ясно показывают, чем команда занимается сейчас и что она уже сделала с точки зрения аудитора.
2. Команда совместно с группой соответствия нормативным требованиям согласует определение готовности, включив в него факты, необходимые для доказательства соответствия историй, спринтов и релизов.
3. Все рабочие материалы – код приложения, конфигурационные рецепты и шаблоны, тесты, политики, документация (естественно, за исключением секретов) – записываются в систему управления версиями с отсылкой к конкретному требованию, запросу изменения или извещению об ошибке (по идентификатору истории, номеру заявки или еще какому-то идентификатору, который можно указать в комментарии при записи в репозиторий). В результате формируется подробная история всего, связанного с каждым изменением.
4. Фильтры записи в репозиторий автоматически сканируют код на наличие секретов и небезопасных вызовов, перед тем как объединить его с главной ветвью.
5. Все изменения кода и конфигурации инспектируются (до записи в репозиторий), и результаты инспекции становятся видны команде. Для этой цели применяются запросы на включение Git или инструменты коллективной инспекции кода типа Gerrit или Review Board.

Инспекторы сверяются с контрольными списками, проверяя, что код отвечает принятым в команде стандартам и наставлениям, и обращая внимание на небезопасное кодирование. Руководство периодически проводит аудит, чтобы подтвердить, что инспекции проводятся надлежащим образом и что программисты не просто проштамповывают работу коллег.

6. Каждое изменение (кода приложения и конфигурации) проходит стадию тестирования: разработка через тестирование, статический анализ кода и другие виды сканирования, а также автоматизированное приемочное тестирование описаны в главе 11. Автоматически измеряется тестовое покрытие. В случае серьезных проблем сборка прерывается.
   Поскольку тесты хранятся в репозитории исходного кода, их можно проинспектировать, сопоставить с критериями приемки каждой истории и убедиться, что требования были реализованы правильно.
7. Код (в т. ч. и код зависимостей) и инфраструктура регулярно сканируются на наличие уязвимостей, поскольку это часть автоматизированного конвейера. Сведения о найденных уязвимостях фиксируются и направляются в журнал пожеланий команды для исправления.
8. Изменения автоматически подвергаются приемочному тестированию, затем переносятся в промежуточную технологическую среду и, если все тесты и проверки прошли, поступают в производственную среду, так что можно увидеть, когда изменение попало в каждую среду и как это происходило.
9. Системы регулярно проверяются на наличие неавторизованных изменений с помощью программ обнаружения изменений (например, Tripwire или OSSEC).

Все это прекрасно с точки зрения эксплуатации и поддержки. В любой момент можно сказать, когда были внесены изменения и исправлены ли уязвимости, а если что-то пошло не так, то можно точно проследить, что именно было изменено, и быстро исправить ошибку. Это хорошо и с точки зрения контроля, потому что можно проследить каждое изменение и удостовериться в том, что все изменения вносятся единообразно и ответственно. И с точки зрения соответствия нормативным требованиям это тоже замечательно, потому что все сделанное можно доказать аудиторам.

Управление изменениями при непрерывной поставке

Управление изменениями имеет фундаментальное значение для соблюдения нормативно-правовых требований и в системах контроля типа ITIL и COBIT. Перечислим, что имеется в виду:

- обязательная авторизация всех изменений – SOX, инсайдерские угрозы, мошенничество, Reg SCI;
- минимизация эксплуатационного риска, связанного с изменением, – изменения должны быть понятны, протестированы и безопасны;
- обязательная аудитопригодность изменений, что зависит от прослеживаемости.

В большинстве систем контроля управление изменениями осуществляется бюрократическим способом с огромным объемом бумажных документов, заблаговременным планированием и формальным утверждением на совещании консультативного комитета по изменениям, где оцениваются риски и последствия изменения, определяется степень готовности и согласует график внедрения.

Такой подход вступает в принципиальное противоречие с гибкой и бережливой разработкой, а особенно с методикой DevOps и непрерывным развертыванием, в основе которых лежит идея о частых итеративных изменениях, включая A/B-тестирование в производственной системе для получения обратной связи. «Частые» может означать несколько изменений еженедельно или даже ежедневно, а в организациях типа Amazon – вообще несколько раз в секунду.

И как же быть с управлением изменениями, если они выкатываются каждые несколько секунд?

А вот как – с самого начала устранить риск, прогоняя каждое изменение через ту самую батарею тестов и проверок, которая была описана выше. И оптимизировать, выполняя изменения небольшими порциями.

Управление изменениями в ITIL основано на нечастых высокорисковых изменениях – «большом взрыве», тогда как в гибких методиках и DevOps изменения в основном небольшие, с низким риском. Можно считать, что это стандартные или рутинные изменения, которые заранее одобрены руководством и не требуют созыва полномасштабного совещания комитета.

Таким же образом можно вносить и многие более крупные изменения, применяя *запуск втемную* (dark launching). Эта практика, популяризированная Flickr и Facebook, состоит в том, что изменения в

коде активируются *флагом функциональности* (feature flag): переключателем, который включается только после одобрения. А тем временем команда может продолжать работу и тестировать изменения постепенно, выпуская релизы, так что это не влияет на эксплуатацию. В некоторых случаях новый код работает в режиме эмуляции для сбора данных об использовании и производительности либо развертывается только для небольшой группы пользователей с целью бета-тестирования – пока все не будут уверены, что новая функциональность готова к эксплуатации.

Темная сторона запуска втемную

Флаги функциональности и запуск втемную несут с собой потенциальные эксплуатационные риски и риски безопасности, которые следует понимать и учитывать:

- хотя темные функции скрыты от пользователей, пока не активированы, код все же может быть доступен атакующему. Добавление этих функций увеличивает поверхность атаки приложения, что особенно опасно ввиду того, что код еще только разрабатывается и может содержать ошибки, допускающие эксплуатацию;
- пока темные функции не активированы, код может быть более сложным, более трудным для понимания, для внесения изменения и для тестирования, т. к. приходится покрывать тестами больше путей, а иногда и комбинаций путей, если несколько функций перекрываются. Чтобы минимизировать эти риски, флаги функциональности не должны существовать долго. После развертывания функциональности флаг следует убрать, а код почистить и подвергнуть рефакторингу.

Прежде чем включать функцию, члены команды могут провести анализ эксплуатационной готовности или «предсмертную» инспекцию, чтобы оценить возможные сценарии отказа и свою способность к реагированию на них, а также заранее проинформировать все заинтересованные стороны о грядущем изменении.

Связанный с изменением риск можно также свести к минимуму с помощью автоматизации, благодаря которой изменения тестируются и развертываются единообразно, с повторяемым результатом. Любые изменения – кода и конфигурации – следует производить с помощью тех же конвейеров сборки и поставки, что используются при тестирова-

нии. Это даст возможность воспользоваться встроенными средствами контроля, гарантировать, что все шаги отрепетированы и проверены, и обеспечить полную прослеживаемость и прозрачность. Все знают, какие были внесены изменения, когда, кем, как они тестировались, как и когда были развернуты.

К моменту, когда вы будете готовы к развертыванию в производственной среде, изменение уже прошло стадии разработки, приемочного тестирования и побывало в промежуточной среде – и всегда выполнялись одни и те же шаги.

При такой модели внесение изменений становится рутинным и предсказуемым процессом.

Разделение обязанностей

Одна из проблем управления изменениями, особенно в средах DevOps, где программисты могут автоматически отправлять изменения в производственную систему, применяя непрерывное развертывание, – это *разделение обязанностей*.

Речь идет о том, что ни один человек не может полностью контролировать изменение от начала до конца и что изменения нельзя вносить без тестирования и утверждения. Это ограничение призвано предотвратить атаки и мошенничество со стороны инсайдеров-злоумышленников, а также не дать честным сотрудникам срезать углы и обходить проверки и противовесы, призванные защитить систему и организацию от рисков безопасности и эксплуатационных рисков.

Разделение обязанностей выделено отдельной строкой в системах контроля ITIL и COBIT и подразумевается в других системах и правовых нормах, например ISO/IEC 27001, SOC 1 и SOC 2, SOX 404, PCI DSS, NIST 800-53, Reg SCI и т. д. Оно тесно связано как с управлением изменениями, так и с обеспечением конфиденциальности данных (благодаря ограничению количества лиц, имеющих доступ к реальным данным).

Аудиторы привычно обращают внимание на доказательство разделения обязанностей, например на сегментацию сети между системами разработки и эксплуатации, на должностные инструкции, из которых должно быть видно, что роли разработчиков, тестировщиков, эксплуатационников и аналитиков службы поддержки исполняют разные люди с разграничением доступа к различным системам и командам.

Наиболее очевидная реализация этого принципа – каскадная модель технологической зрелости (Waterfall/CMMI), в которой требуется документированная передача обязанностей между ролями:

- бизнес-аналитики определяют требования, которые утверждает владелец компании;
- проектировщики получают требования и создают спецификации;
- разработчики пишут код, реализующий спецификации, и передают его для тестирования;
- независимые тестировщики проверяют, что код соответствует спецификациям;
- эксплуатационники упаковывают код для выпуска релиза и ждут, когда менеджер по изменениям завершит анализ последствий и оценку рисков и включит развертывание изменения в график;
- все эти шаги документируются и учитываются для ознакомления и визирования руководством.

Аудиторам такой подход очень нравится. Вы только взгляните на четкую, документированную передачу обязанностей, на экспертизу и утверждение, на двойные проверки, в ходе которых можно выловить все ошибки и должностные преступления.

Но взгляните также и на неоправданные задержки, на накладные расходы, на многочисленные возможности для недопонимания. Вот почему почти никто уже не создает и не поставляет системы таким образом.

В комплекте инструментов DevOps Audit Toolkit проводится мысль, что все эти передачи не нужны, чтобы предотвратить мошенничество и инсайдерские атаки и гарантировать авторизацию, тестирование и учет всех изменений. Можно даже предоставить разработчикам право отправлять изменения прямо в производственную среду при выполнении следующих условий.

- Команда гарантирует, что все изменения отвечают определению готовности, т. е. прошли приемочные тесты, определенные для каждой истории, и получили положительное заключение от владельца продукта, который представляет интересы бизнеса и руководства.
- Дружественная оценка (или парное программирование) являются залогом того, что ни один разработчик или эксплуатационник не сможет внести изменение без того, чтобы еще хотя бы один человек в организации не ознакомился с ним. Можно даже настоять на том, чтобы команда назначала инспекторов случайным образом, дабы предотвратить сговор.
- Любое изменение – кода или конфигурации – проходит через автоматизированные конвейеры сборки и развертывания, это гарантирует, что все они будут протестированы и учтены.

- Разработчикам можно дать доступ для чтения к журналам производственной системы, чтобы они могли помочь в поиске и устранении неполадок. Но любые исправления должны проходить через автоматизированные сборочные конвейеры или автоматически же откатываться в случае ошибки.
- Все изменения, прошедшие конвейер, прозрачны: они протоколируются, публикуются на информационных панелях, в комнатах чатов и т. д.
- Журналы доступа в производственные системы регулярно просматриваются руководством или группой соответствия нормативным требованиям.
- Распределение прав доступа регулярно пересматривается. Это включает доступ к различным средам, к репозиториям, к конвейерам и инструментам управления конфигурацией.
- Автоматизированные средства обнаружения изменений (Tripwire, OSSEC, AIDE и UpGuard) применяются, чтобы уведомить о неавторизованных изменениях в среде сборки и в производственных системах. Если вы производите развертывание несколько раз в месяц или даже несколько раз в неделю, то никаких осложнений не возникает. Но если изменения вносятся несколько раз в день, то нужно тщательно отфильтровывать утвержденные автоматизированные изменения и показывать исключения. Важно, чтобы уведомления посылались для анализа лицам, не входящим в технические команды, – кому-то из группы безопасности или соответствия нормативным требованиям или из руководства, чтобы избежать конфликта интересов.

С авторитетным и объективным взглядом аудитора на разделение обязанностей в среде DevOps можно познакомиться в статье Дугласа Барбина «Auditing DevOps – Developers with Access to Production» (*https://www.schellman.com/blog/2012/12/auditing-devops-developers-with-access-to-production/*).

Встраивание соответствия нормативным требованиям в корпоративную культуру

Чтобы встроить соответствие нормативным требованиям в корпоративную культуру, нужно время и настойчивость. Для этого требуется встречное движение – сверху вниз и снизу вверх.

Руководство должно понимать, что необходимо для соответствия нормативным требованиям, и доводить это до каждой команды. Необ-

ходимо также ясно демонстрировать серьезность намерений как в части готовности тратить на это деньги, так и в части принятия решений относительно приоритетов.

Для команды соответствие нормативным требованиям должно прямо вытекать из готовности работать ответственно и поставлять правильно функционирующее программное обеспечение. Успех будет скорее сопутствовать тем командам, которые уже выбрали в качестве цели нулевую терпимость к дефектам, и тем, которые практикуют правильные технические решения, в т. ч. непрерывную интеграцию.

Чем больше вы задействуете эти решения в работе, а особенно автоматизацию, тем проще будет соблюсти нормативные требования. В описанной выше модели многие нормативные требования можно удовлетворить, если усилить и без того хорошие инженерные практики, которые команды уже применяют или должны бы применять, и пользоваться контрольными журналами, формируемыми автоматизированными средствами.

Даже в четко структурированной предписывающей системе правовых норм типа PCI предоставьте техническим командам возможность выдвигать собственные идеи о том, как удовлетворить требованиям, шанс автоматизировать как можно большую часть работы и право голоса в том, что касается необходимой документации. Помогите им понять, где проходят красные линии, насколько высоко должна быть поднята планка и где можно проявить гибкость в интерпретации нормативных требований или регламентов.

Это не те проблемы, которых нужно избегать или обходить стороной. Эти проблемы следует решать результативно, эффективно и по возможности не нервируя людей, которым предстоит делать свою работу. К ним нужно подходить, как рекомендуют бережливые методики: нарисовать поток создания ценности, понять ограничения, выявить узкие места, где теряется эффективность, измерять, учиться на результатах и совершенствоваться.

Все это тяжкая ноша для стартапов и для команд, не имеющих решительной поддержки со стороны руководства. Но эта цель достижима – и во имя ее достижения стоит потрудиться.

Как доставить удовольствие аудитору

Чтобы аудитор был доволен, ему нужно представить доказательства соблюдения конкретных нормативных требований или достижения определенных в законе результатов.

У каждого свое представление о красоте – кому и кобыла невеста. Так и соответствие нормативным требованиям – субъективное мнение аудитора. Поначалу аудитор может не разобраться в принятом вами подходе, особенно если он привык рассматривать подробные политики и процедуры и настаивать на заполнении контрольных списков и электронных таблиц.

Вы должны будете объяснить, как работают конвейеры, рассмотреть вместе с аудитором средства контроля, код, репозитории, тесты, контрольные журналы и показать, как все это сочетается. Но квалифицированный аудитор, вероятно, сумеет оценить то, что вы делаете, и понять, что это хорошо как для вас, так и для него.

Если вы будете следовать описанному выше подходу, т. е. использовать автоматизированный сборочный конвейер как опору для соблюдения нормативных требований, то сможете доказать, что регулярно сканируете и инспектируете код и инфраструктуру на наличие уязвимостей и имеете возможность узнать, когда уязвимость была обнаружена и исправлена.

Вы можете предъявить полный контрольный журнал для каждого изменения: когда и почему изменение было запрошено, кто разрешил его внести, кто внес изменение и что именно было изменено, кто инспектировал изменение и что при этом нашел, как и когда изменение было протестировано и с какими результатами, когда оно было развернуто.

- Вы можете доказать, что изменения были проинспектированы, протестированы и внесены стандартизированным, повторяемым способом.
- Вы можете продемонстрировать, что в процессе инспекции, тестирования, сканирования и управления релизами соблюдались все политики и средства контроля соответствия нормативным требованиям.
- Вы можете продемонстрировать разделение обязанностей между разработчиками и эксплуатационниками.
- И если вы неукоснительно применяли все средства контроля, то сможете доказать, что все вышеперечисленное было сделано для каждого изменения.

Как частое выполнение шагов сборки и развертывания снижает эксплуатационные риски, так соблюдение нормативных требований при каждом изменении благодаря следованию одному и тому стандартизованному процессу и автоматизации снижает риск допустить несоответствие нормативным требованиям. Вы – и аудиторы – можете быть

уверены, что все изменения вносятся единообразно, что весь код проходит одни и те же тесты и проверки, что все учитывается одинаково – с начала и до конца.

Как одно из доказательств, будьте готовы предъявить аудиторам журналы ошибок тестирования и сборки, а также подтверждение их последующего исправления. Это убедит аудиторов в том, что ваши средства контроля действительно работают и обнаруживают ошибки.

Аудиторы могут удостовериться, что ваша программа управления последовательная, полная, повторяемая и аудитопригодная. Мы уже видим улыбки на их лицах.

Как быть, когда аудиторы недовольны

Не всегда аудиторы будут довольны, особенно если их цель – провести расследование после взлома.

Хотя во многих правовых нормах сформулированы требования к отчетности в случае взлома, не всегда ясна расстановка приоритетов, особенно если вы с головой погружены в расхлёбывание инцидента безопасности. Нужно понять и оценить множество факторов.

Насколько серьезен взлом? Разные агентства имеют разные точки зрения на то, что включать в отчет и когда. Что, если на вашу организацию распространяется несколько правовых юрисдикций и всем им нужно соответствовать?

Перед кем отчитываться в первую очередь? Перед юрисконсультом? Перед правлением? Перед экспертом-криминалистом? Перед правоохранительными органами? Перед партнерами и клиентами? Перед страховым агентом? Перед регулирующими органами? Перед правительственными агентствами? К кому обращаться? Какую информацию необходимо представить и насколько быстро?

Все это нужно продумать заранее и прописать в сценариях реагирования на инциденты.

Решив вопрос о раскрытии информации, вы должны подготовиться к последующему анализу и аудиторской проверке (или проверкам) с целью понять, что случилось, насколько всё плохо, кто несет ответственность и во что обойдется устранение последствий. Если произошел серьезный взлом и вы работаете в регулируемой среде, то организация по определению будет признана не соответствующей нормативным

требованиям – ведь если бы она соответствовала на 100%, то ничего плохого просто не могло бы произойти, верно? Следствию всего лишь предстоит доказать это, а поскольку задним умом всякий дурак крепок, то оно обязательно докажет. После взлома или обнаружения серьезного несоответствия нормативным требованиям все будут пытаться найти улики – выискивать ошибки, недоработки и козлов отпущения, пока не нароют достаточно материала для отчета.

Повторяемые автоматизированные рабочие процессы со встроенными контрольными журналами и доказательствами, хранящимися в системе управления версиями, хотя бы сделают всю эту кутерьму менее болезненной и дорогостоящей для вас и следователей и помогут продемонстрировать, что какие-то вещи (хорошо бы большинство) вы делали правильно.

Сертификация и аттестация

Наличие сертификата по стандарту типа ISO/IEC 27001, свидетельства о соответствии требованиям SOC[8] или аналогичного подтверждения квалификации может быть важным условием соответствия нормативным требованиям.

Наличие такого документа позволит убедить аудиторов, а также клиентов и владельцев – да и конкурентов – в том, что вы предприняли ответственные и разумные шаги для защиты своей организации и клиентов. Сертификация может также послужить аргументом при необходимости защищаться в суде в случае взлома или другой аварии.

Для получения сертификата придется потрудиться. Но он доказывает, что организация достигла определенного уровня зрелости, и неоспоримо свидетельствует о вашем искреннем стремлении делать всё правильно.

Непрерывное соответствие и взломы

В контексте соответствия нормативным требованиям мы говорим о сертификации и аттестации так, будто это конечная цель процесса. Но в некоторых отношениях это только начало. Цель заключается в том, чтобы организация в каждый момент времени соответствовала нормативным требованиям. Любое требование (по крайней мере, в предписывающих нормах на основе правил) по своему замыслу должно быть постоянным и присутствовать в повседневной деятельности. Потому-то

[8] Service Organization Control – Контроль сервисных организаций.

сертификацию или аттестацию необходимо периодически, например раз в год, подтверждать, доказывая, что средства контроля применяются неукоснительно.

Таким образом, соответствие нормативным требованиям уже не сводится к одномоментной проверке аудитопригодного требования, а становится непрерывным процессом. Применяя рекомендованные в этой главе средства автоматизации, ваша команда и организация смогут удостовериться в своем непрерывном соответствии и выявить проблемы на ранней стадии. Это необязательно предотвратит взлом, но когда аудиторы постучатся в дверь, у вас на руках будет подтверждение всех процессов и подходов вплоть до момента взлома, а не только на момент последней аттестации или сертификации.

Сертификация не означает, что вы в безопасности

Успешная сертификация или аттестация еще не означает, что ваша организация соответствует нормативным требованиям. Это означает, что вы выполнили ряд условий, которых вправе ожидать регулирующий орган, применяя общепризнанный и проверенный подход. Тем не менее регулятор может по какой-то причине уличить вас в несоответствии, хотя вы значительно снизили риски.

Сертификация не означает, что вы в безопасности. Можно назвать несколько организаций, которые имели сертификаты и прошли все аудиторские проверки, но все же пострадали от получивших широкую огласку взломов или серьезных аварий. Из того, что ваша квалификация подтверждена сертификатом, не следует, что можно прекратить учебу и совершенствование и почивать на лаврах.

Сухой остаток

Хотя соответствие нормативным требованиям не означает безопасность системы, для многих организаций это непременное условие работы, к тому же стремление соответствовать может стать важной частью подходов к безопасности и корпоративной культуры в целом.

- В правовых нормах сформулированы минимальные требования к средствам контроля безопасности, инспекциям и надзору за соблюдением. Существует два подхода: на основе детально прописанных правил, как PCI DSS, и на основе результатов, как SOX 404.
- Необходимость соответствовать нормативным требованиям налагает ограничения на свободу гибких команд «инспектировать и

адаптировать» способ своей работы, но у команды по-прежнему есть право голоса (и выбора) в том, как именно выполнять возложенные требованиями обязанности.

- Вместо контроля соблюдения требований посредством заполняемых вручную контрольных списков и разовых аудиторских проверок правила соответствия можно встроить в рабочие процессы и средства автоматизации, чтобы контроль производился непрерывно, а сохранение доказательств для аудиторов естественно происходило в процессе работы сборочного конвейера.
- Разделение обязанностей между разработчиками и эксплуатационниками беспокоит регулирующие органы, потому что граница между функциями этих подразделений размыта, особенно в средах, где практикуется методика DevOps. Для управления этими рисками необходимо тщательно продумывать свои действия и опираться на автоматизацию.

За соответствие нормативным требованиям несут ответственность все, даже если вы не кропаете код каждый день или в жизненном цикле разработки занимаетесь всего лишь поддержкой пользователей. Вы можете помочь в защите систем организации и соблюдении нормативных требований, соблюдая правила информационной безопасности, например: блокировать свое устройство, отходя от рабочего места, выбирать стойкие уникальные пароли для рабочих систем, не пользоваться сообща одной учетной записью или одним паролем.

И наконец, будьте бдительны и не молчите, если что-то покажется вам неправильным. Каждый должен вносить свой вклад в защиту систем и уведомлять о любых обнаруженных инцидентах и подозрениях в возможном несоблюдении нормативных требований. Если вы что-то заметили, сообщите об этом. Помните, что соответствовать требованиям нужно ежедневно и круглосуточно, последствия даже краткосрочного несоответствия могут быть очень серьезными. Каждый должен играть свою роль.

Глава 15

Культура безопасности

Значительная часть этой книги посвящена инструментам и приемам, но важно понимать, что слово «гибкий» относится прежде всего к людям. Если вы действительно хотите разработать эффективную и ориентированную на будущее программу безопасности, то не менее важно поговорить о человеческой стороне безопасности, о том, какое значение для ценностей гибких команд имеют эмоциональная чуткость, открытость, прозрачность и коллективизм.

Внимание только к техническим аспектам никогда не достигает цели, во многих отношениях это можно рассматривать как основную причину неудачи программ информационной безопасности, которые пытались реализовать в 1990-е и в начале 2000-х годов.

В каждой организации уже сложилась какая-то культура безопасности. Вопрос в том, хотите ли вы взять на себя ответственность за нее и оказывать на нее влияние или предпочитаете, чтобы она так и оставалась неконтролируемой и неподдерживаемой. В этой главе мы постараемся помочь вам составить объективный взгляд на текущее состояние культуры безопасности, а также предложить практические шаги к тому, как стать во главе и способствовать ее развитию.

Для построения сколько-нибудь значимой культуры безопасности необходимо активное участие всех членов коллектива, а не только группы безопасности, и если другие сотрудники имеют более полное представление о некоторых вызовах, обсуждаемых в этой главе, то это только улучшит общую атмосферу и взаимопонимание. И хотя реализация рассматриваемых здесь практических аспектов (или их элементов) – задача группы безопасности, важно, чтобы и гибкие команды, и безопасники прочитали и усвоили последующие разделы.

Важность культуры безопасности

Складывается ощущение, что термин *культура* в технических контекстах сильно перегружен и часто используется не к месту, поэтому мы рискуем растерять преимущества, проистекающие из ясного и краткого обсуждения культуры. Кроме того, выстраиванию продуманной *культуры безопасности* внутри организации обычно уделяют недостаточно внимания – во многих случаях это даже не считается обязанностью группы безопасности. Но если вы осознаете, что место культуры там, где встречаются технологии и люди, то должно быть ясно, что для успеха программы безопасности не стоит жалеть сил на внедрение всепроникающей культуры безопасности.

Определение «культуры»

Стараясь не слишком усложнять обсуждение, мы, говоря о *культуре безопасности*, возьмем за основу определение культуры, данное Карлом Вигерсом (Karl Wiegers) в книге «Creating a Software Engineering Culture» (издательство Dorset House):

> В понятие культуры входит набор общих ценностей, целей и принципов, которые определяют поведение, действия, приоритеты и решения группы людей, работающих во имя достижения общей цели.

В рамках этого определения общая цель культуры безопасности – добиться такого положения дел, когда каждый сотрудник организации понимает, что безопасность – это коллективная ответственность и для достижения успеха недостаточно усилий одной лишь группы безопасности. Ключ к достижению этой цели – сформировать такие принципы и такое поведение, которые понимают и принимают все члены организации. К сожалению, часто это делается никуда не годно, а в результате люди активно пытаются избежать всякого участия в процессах обеспечения безопасности. И такое поведение следует признать вполне рациональным, принимая во внимание, какие неудобства многие традиционные меры доставляют людям, которых безопасники стараются защитить.

Тяни, а не толкай

Выработка между разработчиками и безопасниками общего понимания того, что действия каждого непосредственно влияют на ситуацию с безопасностью во всей организации, – важный шаг на пути построения

культуры безопасности. Это побуждает каждого сотрудничать с группой безопасности, думая о будущем, что может изменить модель работы безопасников с другими подразделениями – вместо одностороннего *подталкивания* в нужную сторону хотя бы иногда сотрудники *тянут* сами.

Чем меньше группе безопасности приходится подталкивать процесс, тем реже она воспринимается как препятствие людьми, занятыми своими делами, не относящимися к обеспечению безопасности. Чем сильнее *подтягивание* в сторону группы безопасности, тем с большим основанием можно сказать, что внедряется концепция *безопасности как части повседневной работы*. Когда организация настроена на активное сотрудничество с группой безопасности, возникает ощущение, что функция безопасности – не в том, чтобы застопорить всякий прогресс, а в том, чтобы поддержать инновации, предлагаемые сотрудниками.

Выстраивание культуры безопасности

Культуру нельзя купить в магазине или создать в авральном порядке. В каждой организации есть корпоративная культура, отражающая ее ценности и нужды, и ассоциированная с ней культура безопасности должна подстраиваться под те же нужды. Культура безопасности, пользующаяся успехом в одной организации, необязательно переносится на другие, однако не исключено, что в ней можно выделить ряд элементов и адаптировать их к целевой организации. Вопросы, обсуждаемые в этой главе, следует рассматривать как отправные точки или идеи, от которых можно отталкиваться при построении собственной культуры безопасности, а не как правила, подлежащие неукоснительному выполнению.

Следует также отметить, что выстраивание культуры безопасности – задача, которая никогда не будет завершена: достигнутое необходимо постоянно пересматривать и совершенствовать, поскольку сама организация меняется и развивается. Кроме того, культура безопасности должна строиться с оглядкой на внешнее окружение, с пониманием ландшафта технологий и угроз, в котором функционирует организация; необходимо идти в ногу с технологиями, которые приняты и используются сотрудниками организации.

Основная цель культуры безопасности состоит в том, чтобы изменить или, по крайней мере, оказать влияние на техническую и общую корпоративную культуру организации, с тем чтобы в процессе принятия решений соображения безопасности принимались во внимание наряду с прочими факторами. Важная отличительная особенность культуры безопасности заключается в том, что это набор мер, способствующих таким изменениям, а не сами изменения.

Это не означает, что ваша задача – выстроить техническую культуру, в которой безопасность стоит на первом месте, требуется лишь, чтобы безопасность учитывалась наравне с другими требованиями и приоритетами на всем протяжении жизненного цикла ПО. Это кажется само собой разумеющимся, но опыт авторов свидетельствует, что безопасность зачастую рассматривается как второстепенный фактор, который можно игнорировать вовсе или признавать только на словах.

Итак, мы поговорили о технической культуре и культуре безопасности вообще, но как воплотить идеалы в жизнь конкретно в вашей организации? Как уже было сказано, не существует рецепта, который годился бы для всех: у каждой организации свои нормы, нужды и сложившаяся культура, и любая попытка внедрить культуру безопасности должна их учитывать. Нет, мы не хотим сказать, что не следует пытаться изменить существующие обычаи, плохо сочетающиеся с безопасностью, но в стремлении установить любую форму культуры безопасности следует отвергнуть идеализм и оставаться на позициях реализма и прагматизма.

А теперь рассмотрим реальный пример весьма успешного внедрения культуры безопасности и поговорим, какие из него можно извлечь идеи и уроки для себя.

Размышляя о культуре безопасности, полезно рассматривать свой подход к безопасности как бренд, определяющий взгляд на вашу работу со стороны. Бренд – это обещание, данное клиентам: он выражает, что клиенты могут ожидать и в чем ваше отличие от конкурентов. Аналогично четко определенная культура безопасности – обещание, данное сотрудникам организации о ваших обязательствах по обеспечению безопасности, о той поддержке, которую вы стремитесь оказывать, чтобы помогать, а не препятствовать решению проблем, и о том, какого восприятия группы безопасности вы ожидаете. Как и с брендом, для создания культуры безопасности нужно много времени, а подорвать доверие можно в одночасье.

Компании тратят много сил на формирование и защиту своего бренда, поскольку понимают, насколько он важен для долгосрочного успеха. Инвестируйте в культуру безопасности с тем же пылом, и со временем это обернется дивидендами и поможет построить светлое будущее.

Принципы эффективной безопасности

Для конкретики мы рассмотрим группу и программу безопасности в компании Etsy. Это отнюдь не идеал, который следует копировать, а полезный пример усилий, прилагавшихся на протяжении ряда лет, из ко-

торого можно извлечь уроки для себя. Изучив удачные находки и ошибки Etsy, мы сможем добиться результата за меньшее время.

Что такое Etsy и почему она нам интересна?

Etsy – крупнейшая в мире онлайновая торговая площадка для торговли изделиями ручной работы. Она была основана в 2005 году. После несколько неудачных экспериментов с технологиями Etsy радикально изменила способы разработки и поставки программного обеспечения, став новым лидером и привнеся новые методы организации производства. Бывший технический директор, а ныне генеральный директор, Чэд Дикерсон, и нынешний технический директор, Джон Эллспоу, раньше работали в компании Flickr, где внедрили новаторский, в высшей степени коллективистский подход, при котором разработчики и эксплуатационники тесно сотрудничали с целью ежедневного развертывания небольших изменений. Презентация Эллспоу на конференции Velocity в 2009 году «10 и более развертываний в день: кооперация разработчиков и эксплуатационников в Flickr» (*https://www.youtube.com/watch?v=LdOe18KhtT4*) считается одной из главных вех движения DevOps.

И сегодня Etsy остается лидером DevOps и, подобно многим интернет-компаниям, является технически ориентированной организацией, в которой инженеры несут основную ответственность за достижение коммерческих результатов, включая поддержку всех изменений, которые вносят.

На момент написания книги в Etsy работало свыше 300 программистов, которые развертывали новые изменения в производственных системах 50 и более раз на дню, а это означает, что для достижения успеха подход к безопасности должен радикально измениться. Etsy заслуженно считается пионером не только в области DevOps, но и в прогрессивных подходах к безопасности и культуре безопасности.

Хотя Etsy развивала философию безопасности и практические методы ее обеспечения в собственных интересах, многие ее решения могут оказаться полезными и для компаний, которые не практикуют непрерывную интеграцию и развертывание. В этой главе мы более пристально рассмотрим наработки Etsy и ее групп безопасности и обсудим, как их можно использовать для построения позитивной культуры безопасности (и DevOps).

В Etsy культура безопасности базируется на трех основных принципах. Программа безопасности оценивается, исходя из того, насколько хорошо она следует этим принципам или развивает их. Ничто не является незыблемым, все можно пересматривать и уточнять, поскольку Etsy продолжает развиваться и вбирает в себя внешние влияния.

Содействуй, а не блокируй

Если нужно назвать самый главный принцип безопасности в Etsy, то это будет принцип содействия.

> Эффективность группы безопасности должна измеряться тем, чему она содействовала, а не тем, чему препятствовала.
>
> – Рич Смит

Успех многих программ безопасности, групп и отдельных профессионалов измеряется в терминах того, чему они воспрепятствовали: появлению уязвимостей или действия, которые могли бы оказать негативное влияние на организацию, если бы группа безопасности вовремя это не пресекла. На первый взгляд, этот показатель кажется разумным – группа безопасности для того и существует, чтобы пресекать неприятности, но у такого взгляда на вещи есть интересный эффект – за сравнительно короткое время раскручивается отрицательная спираль неприязни к группе безопасности.

Если другие работники привыкнут считать группу безопасности источником помех, вредителями, которые только мешают делать *настоящую работу*, то очень быстро научатся избегать ее любой ценой. Результат очевиден – усиливающееся расхождение между реальным и мнимым положением дел с безопасностью в организации. Группа безопасности имеет очень ограниченное представление о вещах, где ее вмешательство могло бы оказаться полезным, поскольку все прочие работники изо всех сил стремятся избежать контактов с безопасниками, считая, что все их действия только выбивают из рабочей колеи.

Если организация рассматривает группу безопасности как препятствие, то будет искать обходные пути независимо от всяких процессов и политик: чем больше вы мешаете, тем меньше эффекта приносит ваша деятельность и тем больше действий остается от вас скрыто.

Но если перевернуть цель группы безопасности – вместо того чтобы не дать бестолковой команде безмозглых программистов и их менеджеров совершить убийственные ошибки в их дурацких проектах, бросить все силы на то, чтобы содействовать инновациям и искать пути безопасного воплощения даже самых сумасшедших идей, – то группа

безопасности становится интересным объектом, с которым можно сотрудничать и у которого можно просить совета.

Этот переворот во взгляде на мир – измерять не то, чему удалось помешать, а то, что удалось осуществить, – настолько фундаментален, что без ясного его осознания и принятия трудно говорить о том, что организация вообще понимает, что такое прогрессивный подход к безопасности.

Мы вовсе не хотим сказать, что группе безопасности вообще запрещено говорить «нет». Сказанное означает, что во многих случаях, когда «нет» было бы самым естественным ответом, слова «да» или «да, но» станут залогом долговременного сотрудничества.

Слова «да, и» означают, что группа безопасности готова оказать консультацию и всяческую поддержку, чтобы довести идею или проект до такого состояния, когда все задуманное воплотится в жизнь, но некоторые (или даже все) аспекты, ставящие под угрозу безопасность, будут устранены.

«Нет» – это туз в колоде группы безопасности, которым играть нужно редко: если использовать его слишком часто или как стандартную реакцию на любой проект, в котором требуется нестандартное мышление, чтобы найти разумный компромисс между целями бизнеса и безопасности, то окажется, что группа безопасности в основном возводит препятствия, а не стремится содействовать успеху. Нежелательный побочный эффект ответа «нет» состоит еще и в том, что группу безопасности исключают из процесса принятия решений и цикла обратной связи. Разработчики перестают работать с безопасниками над решением проблем, а просто мирятся с ними или не обращают на них внимания и ищут обходные пути. Такой отказ от сотрудничества и консультаций с группой безопасности наносит огромный ущерб организации в том, что касается творческого подхода к решению задач и, что самое главное, создания ценности.

Если члены группы безопасности не загораются от перспективы найти решение новой, трудной и комплексной проблемы, для которой нет однозначных ответов, то стоит подумать, а те ли люди отвечают у вас за безопасность. Ленивые безопасники отвечают «нет» по умолчанию, прикрывая свой тыл от любых негативных последствий, которые мог бы повлечь за собой проект, которому они оппонируют. Неэффективные группы безопасности стремятся сохранить профиль безопасности компании неизменным, чтобы не пришлось делать трудный выбор между безопасностью и инновациями. Ничто не заставит сторониться безопасников быстрее, чем ожидание, что проекту будут вставлять пал-

ки в колеса, потому что безопасники не желают изменений в профиле безопасности.

Не стоит и говорить, что в организациях с такими командами образуется пропасть между инженерами, стремящимися двигать компанию вперед, и экспертами по безопасности, задача которых – помочь в понимании рисков и нахождении компромисса при решении новаторских или повседневных задач.

К сожалению, безопасность редко бывает выбором между черным и белым. Во многих случаях существуют остаточные риски, и для достижения успеха следует сопоставлять их с выгодами рассматриваемого проекта. Осознанное принятие риска может оказаться правильным подходом при условии, что риски и потенциальные последствия хорошо поняты, оценены и соотнесены с плюсами, которые сулит продолжение проекта, несмотря на них. Осознанное принятие риска всегда лучше слепого игнорирования.

Анализ и оценивание рисков, связанных с проектом, – отличная возможность наладить созидательный диалог между группой безопасности и разработчиками. Это важно не только для определения приемлемого уровня риска, но и способствует зарождению взаимопонимания и может значительно снизить уровень неуверенности, зачастую испытываемой разработчиками, когда они задумываются о безопасности своего творения. Обсуждение того, какой уровень риска считать приемлемым, и выгод, которые можно получить, оставив этот риск, демонстрирует, что все находятся по одну сторону баррикады и работают совместно во имя выпуска самого лучшего продукта, причем безопасность – один из факторов, делающих продукт *лучшим*.

Стопроцентная безопасность – это зачастую иллюзия, питаемая принятой моделью угроз, стремление к ней редко стоит выставлять как причину зарубить проект на корню. По отношению к большинству коммерческих проблем безопасности и их решений правильнее употреблять слова *прагматическая безопасность*. Слишком часто приходится наблюдать, как у команды безопасности входит в привычку называть любое подозрение критической проблемой; это то же самое, что каждый раз кричать «волки», и является самым коротким путем к отторжению (или полному игнорированию) разработчиками любых предложений группы безопасности.

Очень часто встречаются неверные представления по этому поводу. Рассмотрим пример: злополучная попытка остановить запуск сайта маркетинговой кампании из-за отсутствия поддержки HTTPS. Сайт размещался на отдельных серверах и не в том домене, что основной

сайт организации, и содержал только статические публичные данные. И хотя это было нарушением принятой в организации политики, согласно которой все веб-активы должны быть защищены протоколом SSL, фактический риск был минимален, если не считать потенциальный репутационный ущерб вследствие владения доменом, не защищенным HTTPS. Маркетинговый отдел, являющийся владельцем проекта, был засажен за определение критичности риска. Как часто бывает с маркетинговыми компаниями, время играло не на стороне проекта, так что значительные усилия по наращиванию общественного и акционерного капитала были выброшены на ветер группой безопасности, которая бездумно отстаивала политику в отсутствие всякого риска. К счастью, ее попытки остановить проект в конечном итоге провалились, и сайт был благополучно запущен в срок.

Остерегайтесь также сторонних компаний, в т. ч. тестировщиков проникновения, которые тоже рады сделать из мухи слона, пытаясь найти критические уязвимости и ужасающие последствия для демонстрации собственной нужности. Гораздо чаще, чем хотелось бы, найденные недочеты представляются с точки зрения стопроцентной, а не прагматичной безопасности, и в задачи группы безопасности, получившей результаты внешней оценки, входит правильная расстановка приоритетов, поскольку эта группа гораздо лучше понимает контекст, в котором функционирует организация, и осознает, с какими рисками можно смириться.

Вы станете препятствием, только если узнаете обо всем в последнюю очередь. Группа безопасности должна быть включена в жизненный цикл разработки на ранних стадиях и вносить свой вклад в создание историй. Но проявлять инициативу должны не только разработчики – группа безопасности также должна стремиться к установлению ранних и частых контактов с разработчиками. Один из способов достичь этого – назначать степень срочности всем находкам и останавливать работу, только если нет разумных альтернатив, а не всякий раз, как обнаруживается какая-нибудь проблема, пусть даже незначительная. Иногда лучший подход, по крайней мере для некритичных проблем, – разрешить выпуск системы с дефектом, заручившись обещанием разработчиков устранить его в оговоренные сроки.

Неэффективная организация безопасности – пристанище для тех, кто жаждет избежать изменений любой ценой в надежде избежать наказания за потенциальную небезопасность в будущем. С другой стороны, при эффективной организации безопасности такое умонастроение искореняют и заменяют его другим – тот факт, что изменения могут при-

вести к проблемам с безопасностью, принимается как данность, но это считается лишь удачным поводом встретить проблемы лицом к лицу. И тогда роль группы безопасности как помощника в осуществлении проектов, превращающего потенциально разрушительные инновации в настолько безопасные, насколько возможно, будет признана и вознаграждена.

Понять, как вашу деятельность оценивает организация – как содействие или как препятствие, – первое, что нужно сделать, если вы серьезно настроены выстраивать прогрессивную культуру безопасности.

Прозрачная безопасность

> Группа безопасности, которая ни от кого не скрывает, что делает и почему, распространяет вокруг себя понимание.
>
> – Рич Смит

Безопасники часто совершают одну и ту же ошибку: начинают изображать из себя *секретных агентов*. Это означает секретность, разделение на группы и прочие действия, которые предпринимаются во имя безопасности, но зачастую только вредят конечной цели – сделать организацию более безопасной. А все потому, что безопасники хотят выглядеть какими-то особенными или крутыми ребятами.

Принцип прозрачности в культуре безопасности, на первый взгляд, может показаться противоречащим интуиции, однако прозрачность может принимать разные формы, и все становится на свои места, если рассматривать этот принцип как распространение правильных представлений о безопасности на более широкую группу лиц.

Смысл прозрачности можно пояснить на совсем простом примере: разъяснение, почему некоторая проблема так беспокоит группу безопасности, четко описав реальные последствия, которые она может повлечь. Показав разработчикам, как необработанные входные данные могут быть использованы во зло и как это может отразиться на компании, вы достигнете гораздо более прозрачного и полезного эффекта, чем если бы ткнули их в наставление по безопасному кодированию на конкретном языке и бросили несколько общих слов. Первый подход вызывает эмоциональную близость и способствует постепенному распространению знаний о безопасности. Второй, скорее всего, будет воспринят как бесполезное начетничество.

Если используется внешнее оценивание (тестирование на проникновение или непрерывная программа вознаграждения за найденные ошибки), обязательно сотрудничайте с командами разработчиков, ко-

торых это касается, на протяжении всего процесса – от обсуждения объема и содержания тестирования до его завершения, рассказывайте им обо всех найденных проблемах и привлекайте к обсуждению и принятию решений о том, как смягчить последствия и исправить дефекты.

Нет ничего более отталкивающего, чем группа безопасности, которая вместе с третьими лицами ищет дыры в системе, а затем по итогам каких-то тайных обсуждений выкатывает серию изменений и требований. Команда, чей проект оценивается, вряд ли обрадуется срочному перечню требований или недостатков, доведенному до нее в приказном порядке. Скорее, она хотела бы понимать, какие проблемы были обнаружены, как это произошло и каковы реальные возможные последствия. Мы уже говорили на эту тему в главе 12 и теперь не будем повторяться. Однако считаем важным рассмотреть культурные аспекты разъяснительных бесед о проблемах, которые могут быть найдены во время внешней оценки безопасности, отметив, что такие беседы следует проводить еще до того, как тестировщики проникновения увидели первую строку кода или отправили первый пакет.

Значимость оценки в немалой мере обусловлена максимальной прозрачностью для команд, чья работа подвергается экспертизе. Это не только шанс найти серьезные дефекты и изъяны, связанные с безопасностью, но для разработчиков еще и редкая и весьма ценная возможность поучиться у команды целеустремленных специалистов по безопасности. Есть простые, но, к сожалению, нечасто используемые решения, например открыть разработчику репозиторий ошибок, найденных во время внутренних или внешних оценок, со всеми неприглядными подробностями. Это может также способствовать созданию положительной обратной связи, когда разработчики сами пополняют репозиторий ошибок, если им удается найти что-то интересное для безопасников. Удивительно, сколько относящихся к безопасности дефектов можно найти, воспользовавшись коллективными знаниями команды разработчиков.

Еще одно возможное проявление принципа прозрачности – продемонстрировать открытость и пригласить разработчиков временно войти в группу безопасности и занять свое место в окопах. Это может быть частью испытательного срока, который новые разработчики проходят перед зачислением в штат, либо частью программы непрерывного обучения или периодического перехода из одной команды в другую: важно, чтобы лица, не работающие в группе безопасности, получили возможность увидеть, как выглядят ее трудовые будни. Знакомство с будничными деталями работы безопасников способно пробудить в раз-

работчиках чувство эмоциональной близости и понимания больше, чем что-либо другое. Часто это приводит к тому, что в организации появляется много людей, которые формально не входят в группу безопасности, но поддерживают с ней тесный контакт и продолжают расширять свои знания о безопасности и применять их к процессу разработки. В Etsy таких людей называют «проводниками безопасности» (security champions), они многое делают, для того чтобы группа безопасности оставалась доступной максимально большому числу команд.

Группа безопасности, которая по умолчанию демонстрирует открытость и лишь в редких случаях прибегает к ограничению доступности информации, сможет гораздо эффективнее распространять знания о том, что делает, и – главное – о том, почему она это делает.

Не ищите виноватых

> Инциденты безопасности неизбежно случаются, но, лишь не пытаясь повесить на кого-то все грехи, вы сможете понять истинные причины, извлечь уроки и улучшить положение дел.
>
> – Рич Смит

Хотя признавать это неприятно, в достаточно сложной системе никакие действия группы безопасности не смогут устранить все возможные причины инцидентов. Хуже того, те же самые ошибки могут возникнуть и в будущем.

Отказ от поиска виноватых в качестве одного из столпов вашего подхода к безопасности? Как-то странно видеть это в списке трех главных принципов – очевидно же, что обеспечение безопасности организации включает поиск проблем, являющихся причинами небезопасности, а чтобы повысить безопасность, нужно же найти, что (или кто) несет ответственность за эти проблемы. Правильно?

Неправильно!

Всякий работающий в современной организации знает, как сложно устроена в ней жизнь и что со временем сложность среды только возрастает, а не уменьшается. Если добавить сюда еще технологии и программное обеспечение, то мы очень быстро придем к ситуации, когда ни один отдельный человек не сможет описать все хитросплетения системы сверху донизу. И если объявляется, что такую величественную конструкцию нужно обезопасить, то сложность этой задачи оказывается столь непомерной, что какие-то углы неизбежно будут срезаться, поскольку иначе невозможно добиться никакого прогресса в отведенное время.

И один из вопросов, который настолько распространен, что обычно даже не обсуждается, – это вопрос вины. Поиск виноватых часто стоит на первом месте после обнаружения проблемы, – но это совершенно бесполезно. Важно понимать, что попытка выстроить культуру безопасности на вере в то, что назначение виноватого во всех бедах сможет сделать систему более безопасной в будущем, принципиально порочно.

Многие проблемы, на первый взгляд, возникают из-за того, что кто-то что-то решил или сделал или, наоборот, не сделал. Когда такая ситуация возникает, а она возникает обязательно, следующая мысль – как привлечь человека, который кажется виноватым, который допустил ошибку, к ответственности за халатность. Он стал причиной серьезного инцидента, из-за которого группа безопасности, а то и вся организация, целый день стояла на ушах, и должен понести за это достойную кару.

Меры есть разные – публичное шельмование, запрет даже приближаться к системе снова, понижение в должности и даже увольнение. И будучи настроены на такой лад, мы видим и еще одно преимущество: остальным будет неповадно, и они не сделают такой ошибки, опасаясь разделить ту же судьбу.

Если ступить на этот путь порицания, то мы выстроим культуру безопасности, основанную на запугивании, где люди просто слишком боятся последствий, чтобы делать ошибки.

Было бы справедливо сказать, что программной инженерии стоит многому поучиться у других инженерных дисциплин в вопросе вины и, в более широком смысле, безопасности. Известный авторитет в области инженерии безопасности, Эрик Холлнейгл (Erik Hollnagel), применял разработанные им подходы в самых разных отраслях, включая авиастроение, здравоохранение, атомную энергетику, и – к счастью для нас – в области безопасности программной инженерии. Точка зрения Холлнейгла заключается в том, что поиск коренной причины и последующее устранение этой ошибки в попытке все исправить и тем самым сделать систему безопаснее – подход, который хорошо работает для машин, но не для людей.

На самом деле Холлнейгл утверждает, что акцент на поиске и устранении причины настолько основательно повлиял на подход к оценке рисков, что коренные причины зачастую не столько отыскиваются, сколько ретроспективно выдумываются расследователями. Говоря о человеческом факторе, присутствующем в авариях, Холлнейгл отмечает, что если мы хотим снизить риски, то при проектировании системы должны учитывать неизбежное непостоянство качества работы челове-

ка. Холлнейглу приписывают следующую цитату, подводящую итог его взглядам на аварии с участием человека:

> Мы должны понять, что аварии происходят не потому, что человек поставил и проиграл.
> Аварии происходят, потому что человек полагает, что:
> …то, что вот-вот случится, невозможно,
> …или то, что вот-вот случится, никак не связано с тем, что он делает,
> …или что возможность получить желаемый результат стоит любого риска.

Если не существует неопровержимых доказательств того, что кто-то в организации сознательно пытается вызвать инцидент безопасности, то причину инцидента не следует сходу относить на ошибку человека, а затем тут же налагать соответствующее взыскание. Ключ к прогрессивной культуре безопасности – всеобщее понимание, что работники не стремятся подорвать безопасность, а пытаются делать свою работу способом, который, как они полагают, приведет к желаемому результату, однако происходит нечто неожиданное или неподконтрольное им – и вот вам инцидент. Вместо того чтобы тратить энергию на поиск коренной причины и виноватого, лучше направить усилия на то, чтобы понять, почему человек принял определенное решение и что можно улучшить в той системе, где он работает, чтобы он мог своевременно обнаружить, что вот-вот случится какая-то неприятность.

Если акцент ставится на том, чтобы найти виноватых и воздать полной мерой тем, на кого повесили ответственность, то получится, что люди, которые больше всех знают о происшедшем, не станут честно и откровенно рассказывать обо всем, что знают, из страха перед возможными последствиями. Если же, напротив, люди, находившиеся ближе всех к инциденту безопасности, уверены, что могут абсолютно честно рассказать все о том, что и почему они делали, и что эта информация послужит улучшению самой системы, то гораздо больше шансов, что вы сможете выявить критические недостатки системы и, быть может, модифицировать ее таким образом, чтобы в будущем работать с ней было безопаснее.

Механизм, разработанный для того, чтобы помочь в поиске причин решения и затем улучшить систему, так чтобы в будущем решения принимались более обоснованно, – это посмертный анализ без поисков виновного.

Приступать к процессу посмертного анализа следует, понимая, что случай, который вы расследуете, вполне может повториться. Предотвра-

щение будущих аварий – не самоцель, цель – извлечение уроков, именно на этом и должен быть сосредоточен посмертный анализ. Участники должны сознательно противиться естественному человеческому желанию найти единственное объяснение и единственный способ исправления ошибки. Часто этому помогает присутствие руководителя посмертного анализа, который кратко инструктирует других участников, в чем состоит их задача. Это должно быть слабо структурированной общей беседой, и не более того.

На бумаге все это выглядит гладко, но как обстоит дело на практике? По опыту работы одного из авторов в Etsy, посмертные анализы инцидентов безопасности собирали едва ли не самую многочисленную аудиторию и давали законную возможность исследовать коренные причины, что приводило к существенному улучшению безопасности систем. Посмертный анализ вдохновляет инженеров делать то, что они умеют делать лучше всего, – искать новаторские подходы к решению задач, а заодно устанавливает доверительные отношения между инженерами и группой безопасности. Большое число присутствующих на собрании также открывает отличную возможность проинформировать о безопасности широкую и разнородную аудиторию. Кто не захочет прийти на собрание, где разбираются детали резонансного инцидента, и послушать, как эксперты со всей организации совместно решают проблему у всех на глазах?

Наконец, стоит отметить, что посмертный анализ – полезный способ создать и распространить базу знаний о проблемах, известных различным командам. Если рассматривать коренные причины и повторяющиеся закономерности в целом, а не для каждого случая в отдельности, то и найденные решения будут масштабируемыми и системными.

В главе 13 мы отмечали, что хотя многое, касающееся посмертного анализа без назначения виноватых в программной инженерии, написано с точки зрения эксплуатации, вынесенные уроки в значительной мере применимы и к безопасности, а последние десять лет развития DevOps многому нас научили. Дополнительные сведения о посмертном анализе без назначения виноватых можно почерпнуть в следующих источниках:

- «What Etsy Does When Things Go Wrong: A 7-Step Guide» (*http://bit.ly/etsy-7-step-guide*);
- «Blameless PostMortems and a Just Culture» (*http://bit.ly/blameless-postmortems-just/*);
- «Etsy's Debriefing Facilitation Guide for Blameless Postmortems» (*http://bit.ly/etsy-debrief-guide*);
- «To Err Is Human: The ETTO Principle» (*http://bit.ly/etto-principle*).

Красивый и лаконичный пример: манифест безопасно работающего инженера

Три принципа эффективной безопасности – развернутый способ охватить многое из того, что организация, возможно, захочет включить в понятие культуры безопасности. Но во время презентации Криса Хаймса (Chris Hymes), посвященной подходу к безопасности в компании Riot Games, один из авторов услышал красивый и лаконичный способ облечь их в понятные и находящие отклик в душе слова.

Манифест безопасно работающего инженера:

1. Я буду паниковать корректно.
2. Я понимаю, что ноутбук может уничтожить всё.
3. Я буду противиться желанию создать никуда не годные пароли.
4. YOLO не является девизом инженеров.

Выраженный в шутливой форме, этот манифест – прекрасный пример призыва к огромной популяции инженеров ответственно относиться к своей безопасности, и в то же время он доносит важные и осмысленные советы без диктаторских замашек.

Крис честно признается, что его манифест основан на советах и взглядах других людей и в первую очередь на замечательной статье в блоге Райана Макджихана (Ryan McGeehan) «Политика информационной безопасности для стартапа» (*http://bit.ly/isp-for-startup*).

Размышляя о том, как выстроить культуру безопасности в своей организации, попробуйте написать адаптированный к условиям организации манифест, который можно было бы раздать командам. Это отличный способ приобрести сторонников своего видения безопасности.

Масштабировать безопасность, усиливать фланги

Практический подход к прогрессивной безопасности, в котором воплощаются многие описанные выше принципы, состоит в том, чтобы активно привлекать сотрудников, не являющихся членами группы безопасности, к процессу принятия решений, а не исключать их, навязывая безопасность как нечто чужеродное. Во многих случаях сами пользователи располагают куда большими знаниями о контексте происходящего, чем безопасники; включение их в процесс принятия ре-

шений и разрешения проблем поможет сделать реакцию на инциденты безопасности соразмерной угрозе и правильно расставить приоритеты.

Включить всех в цикл принятия решения и разрешения проблем можно по-разному, конкретный способ следует выбирать с учетом особенностей организации. Приведем несколько примеров:

- оповещать пользователей во внутреннем чате о некритических инцидентах, например о неудачных попытках входа в систему, о входе с незнакомого IP-адреса или о необычном трафике на внутреннем сервере, присовокупив просьбу кратко объяснить, что происходит. Во многих случаях объяснение вполне невинное, и никаких дальнейших действий не требуется. Если пользователь не знает, что произошло, или не отвечает в течение разумного времени, то группа безопасности может вмешаться и продолжить расследование. Отличное обсуждение такого подхода имеется в статье Райана Хубера (Ryan Huber) «Distributed Security Alerting» (*http://bit.ly/distributed-security-alerting*);
- уведомлять пользователей в момент входа в систему о том, что ОС, браузер или подключаемые модули устарели, и давать возможность обновиться немедленно (предпочтительно в один клик, если это возможно), но также – и это очень важно – позволять отложить обновление на потом. Сколько раз можно откладывать обновление, а также версии ПО, для которых откладывание возможно, определяется группой безопасности, но главное, чтобы у пользователя был выбор между оттенками серого, а не только между черным и белым. Например, если браузер лишь немного устарел, то вполне допустимо дать пользователю сделать то, что он хотел, а обновиться позже, вместо того чтобы блокировать доступ и заставлять перезагружать браузер в самый неподходящий момент;
- спросить пользователя по электронной почте или в чате, не устанавливал ли он только что новое программное обеспечение или не редактировал ли вручную конфигурацию, поскольку в его системе наблюдаются изменения, например: загружен новый модуля ядра, удален файл истории команд оболочки или смонтирован ram-диск. Хотя такие действия вызывают подозрения и могут быть признаком вредоносной активности, у программиста может быть много причин для внесения низкоуровневых изменений в конфигурацию своей системы;
- во внутренних тестах фишинга используйте в качестве основного показателя не количество щелчков мышью по ссылкам в фишин-

говом письме, а количество пользователей, сообщивших группе безопасности о том, что они получили нечто странное. В программах информирования о фишинге акцент часто ставится на стремлении свести количество переходов по ссылкам к нулю, что неправильно. Авторы этой книги могут засвидетельствовать на примере многих организаций, что задача добиться того, чтобы *каждый* сотрудник вел себя правильно и *никогда* не щелкал по ссылкам в фишинговом письме, благородна, но абсолютно нереалистична. Если условие выигрыша состоит в том, что каждый поступает правильно в каждом случае, то, вероятно, вы играете не в ту игру, поскольку выигрывать будете очень редко. Если все усилия направлены на то, чтобы убедить человека в том, что правильное действие – это уведомление группы безопасности о получении фишингового письма, то выиграть в такую игру шансов куда больше.

Это лишь несколько примеров, на самом деле вариантов гораздо больше, и все они направлены на то, чтобы включить пользователя вместе с его контекстом в процесс принятия решений. Это и группе безопасности позволит правильнее расставлять приоритеты в своей работе, и пользователи не будут воспринимать безопасность как палку, которая их обязательно ударит. Ко всему прочему такое взаимодействие дает отличную возможность доставить по нужному адресу крохи сведений о безопасности и способствовать формированию спаянного коллектива, сознающего свою ответственность за принятие относящихся к безопасности решений. В конечном счете образуется положительная обратная связь с пользователями, которые активно сотрудничают с группой безопасности в духе «заметил – сообщи» или заранее уведомляют о том, что собираются сделать нечто такое, что может вызвать срабатывание систем оповещения.

Кто – не менее важно, чем как

Все сказанное касается того, в чем должна состоять миссия эффективной группы безопасности. Ну, а следующий шаг – выполнение этой миссии. Для построения замечательной культуры нужны замечательные люди, а сотрудников группы безопасности, увы, слишком часто набирают, ориентируясь только на технические знания и не обращая внимания на человеческие качества. Но если мы верим, что для эффективной программы безопасности люди важны не меньше, чем технологии, то должны учитывать это при приеме на работу в группу безопасности.

Одна из причин, по которым безопасников избегают, состоит в том, что их личные качества и отношение к людям прямо противоречат задаче сделать безопасность близкой и понятной сотрудникам организации. Нередко безопасники выказывают снисходительное и покровительственное отношение к тем, кого они призваны поддерживать, а некоторые даже непритворно радуются, видя проблемы, с которыми столкнулись их коллеги.

Эта книга не об управлении коллективами людей, но стоит отметить, что руководитель, отвечающий за программу безопасности, должен обращать особое внимание на поведение членов группы безопасности, не согласующееся с более широкими целями построения культуры безопасности, и пресекать его в зародыше. Главное правило, которое следует иметь в виду при наборе сотрудников в группу безопасности, кажется очевидным, но, как показывает практика, слишком часто не соблюдается:

> Не нанимай козлов.

И тесно примыкающее к нему:

> Если случайно нанял козла или получил его по наследству, избавься от него как можно скорее.

Один-единственный козел в группе безопасности может сделать для подрыва усилий по реализации программы прогрессивной безопасности больше, чем всё остальное вместе взятое, поскольку безопасники – лицо этой программы, именно с ними другие сотрудники имеют дело ежедневно.

Этот вопрос гораздо более подробно обсуждается в замечательной книге Robert I. Sutton «The No Asshole Rule: Building a Civilized Workplace and Surviving One That Isn't» (издательство Business Plus). По мнению авторов, эту книгу обязан прочитать любой менеджер по безопасности, и она должна всегда быть под рукой на случай, если содержащиеся в ней ценные уроки начнут забываться.

Продвижение безопасности

У Кори Доктороу (Cory Doctorow) в книге «Information Doesn't Want to Be Free» (издательство McSweeney's), вышедшей в 2014 году, есть замечательный пассаж, в котором очень кратко изложена самая суть продвижения безопасности:

> Дружеская беседа – неизбежный продукт социализации. Именно в дружеской беседе устанавливаются доверительные отношения между людьми.

Продвижение безопасности – это не пицца, купленная в качестве своего рода взятки, чтобы удержать людей на презентации, посвященной проверке входных данных. Это искреннее стремление создавать такие неформальные ситуации, которые располагали бы к разговорам между безопасниками и другими сотрудниками. В таких разговорах обязательно завязываются доверительные отношения, а они, в свою очередь, создают благоприятный эмоциональный фон вокруг безопасности.

И скажем честно, эта эмоциональная близость действует в обе стороны; мало того что остальные начинают осознавать, сколь трудна работа группы безопасности, но и у безопасников раскрываются глаза на досадные мелочи, с которыми ежедневно приходится бороться коллегам из других отделов, желающим всего лишь выполнить свою работу. Создание условий, при которых могут познакомиться реальные люди, стоящие за логинами и адресами электронной почты, – ключ к тому, чтобы избежать возникновения ложных представлений друг о друге.

Продвижение безопасности дает группе безопасности шанс не поддаться искушению считать разработчиков ленивыми или просто безразличными к безопасности, а разработчикам – не пасть жертвой измышления, будто безопасники испытывают извращенное удовольствие, заставляя их делать дополнительную работу. Благодаря продвижению безопасности каждый предполагает, что коллегами движут наилучшие намерения, а это хорошо не только для безопасности организации, но и для многого другого.

Приведем несколько примеров продвижения безопасности, не сводящихся к примитивному обучению.

- Поход в кино на вечерний сеанс, где крутят какую-нибудь фантастику или фильм о *хакерах*. Дополнительная фишка – всем надеть роликовые коньки и соответствующий прикид!
- Приз, который группа безопасности периодически и с большой помпой вручает за какое-нибудь выдающееся достижение. Например, за наибольшее число сообщений о фишинговых письмах или исправленных дефектов, связанных с безопасностью.
- День охоты за ошибками, когда группа безопасности располагается в общем помещении и бомбит новый продукт или функцию, достигшие очередного этапа жизненного цикла, пытаясь найти дефекты и демонстрируя находки на большом экране. При этом любой может подойти и спросить, что они делают и к чему все это.
- Просто пригласить ребят из других подразделений в бар по соседству выпить пару рюмочек за счет безопасников.

Разумеется, любой пример следует адаптировать с учетом того, чем занимается организация и какие люди в ней работают.

И позаботьтесь о том, чтобы руководство одобрило концепцию продвижения безопасности и ассигновало на нее средства. И уверяем вас, отдача будет едва ли не самой высокой во всей программе безопасности.

Эргобезопасность

До сих пор мы в основном говорили о теоретических аспектах выстраивания и развития культуры безопасности. Но, поскольку культура лежит в основе совместной работы людей и формирования коллектива, не следует игнорировать и практические аспекты. Один из авторов книги (сознаемся, это был Рич) придумал для презентации термин *эргобезопасность* (securgonomics), составленный из двух слов: *эргономика* и *безопасность*.

Если эргономика определяется в словаре следующим образом:

> эргономика
> имя существительное, женский род
> наука об эффективности труда человека на рабочем месте, –

то определение эргобезопасности (если бы такое слово существовало) звучало бы так:

> эргобезопасность
> имя существительное, женский род
> наука об эффективности безопасного взаимодействия человека на рабочем месте

Хотя слово и придуманное, оно отражает вполне реальное наблюдение: группа безопасности часто совершает ошибку, отгораживаясь от людей, которых призвана поддерживать. Гибкий подход нацелен на разрушение стен, поскольку они мешают общению и сотрудничеству, и его необходимо распространить на реальный мир, где обитают безопасники и другие члены организации. Задача в том, чтобы понизить или вовсе устранить барьеры для взаимодействия с группой безопасности.

Офисное пространство должно быть устроено так, чтобы безопасников было видно, чтобы к ним было легко подойти и завязать разговор. Увы, нередко группа безопасности отгораживается стенами и запертыми дверьми, что создает реальное разделение на *мы* и *они*. Конечно, безопасники в своей повседневной работе имеют дело с секретными данными, но верно и то, что большую часть времени их работа не более секретна, чем работа коллег.

Безопасники часто примеряют на себя стереотипный образ секретных агентов, работающих над вещами, которые нельзя выносить за пределы группы. Иногда так оно и есть, но гораздо чаще это ложное (и никому не нужное) впечатление, создаваемое самими безопасниками. Команда, отгородившаяся от остальной организации, окажется в крайне невыгодном положении, попытавшись стать прозрачной командой помощников, которая стремится сделать так, чтобы все сотрудники искренне разделяли ответственность за безопасность.

Посмотрите, где сидит ваша группа (или группы) безопасности, и задайтесь такими вопросами:

- Помогает это или мешает свободно подойти и поговорить с членами группы?
- Видят ли люди безопасников постоянно в течение рабочего дня или, для того чтобы поговорить лично, должны специально куда-то идти?
- Видят ли новые сотрудники безопасников в первый день, когда совершают обзорную экскурсию по офису?
- Располагается ли группа безопасности рядом с теми командами, с которыми контактирует наиболее тесно и часто?
- Могут ли другие сотрудники понять, что группа безопасности находится в офисе? Видят ли они, что группа занята?

Если есть возможность, отведите группе безопасности помещение в таком месте, мимо которого все ходят, так чтобы для общения не нужно было специально подниматься на лифте. Если люди проходят мимо группы безопасности на пути к кофейному автомату или входя либо выходя из офиса, то мимолетные контакты неизбежны. Если не заставлять людей предпринимать специальные усилия, чтобы добраться до безопасников, то случайные разговоры будут завязываться чаще. Это хорошо для всех, и не в последнюю очередь потому, что способствует установлению доверительных отношений между безопасниками и всеми остальными.

Местонахождение группы безопасности имеет значение. Но, даже получив самое лучшее место, легкодоступное всем желающим, группа не должна почивать на лаврах: необходимо работать над тем, чтобы у людей появилось желание общаться с ними. Здесь не надо ничего сверхъестественного, достаточно просто обменяться стикерами, завести библиотеку по безопасности, где любой может одолжить книгу, или поставить банку с леденцами. В Etsy конфетки от группы безопасности стали самостоятельным явлением: вся организация с интересом реша-

ет, какие конфеты заказать в следующий раз, и может узнать, сколько конфет осталось, задав вопрос внутреннему чат-боту *irccat*.

Как-то даже создали специальный проект – поставили веб-камеру над банками с конфетами, чтобы можно было удаленно посмотреть, что есть в наличии. Пусть затея смешная и несерьезная, но отдача от этого простого акта раздачи конфет, чтобы поощрить неформальное общение с группой безопасности, многократно превзошла финансовые затраты на него.

И, завершая разговор о физическом расположении групп безопасности, хотим дать совет: не требуйте, чтобы сотрудники не отходили от своих рабочих столов. Не менее важен другой метод работы – когда сотрудники могут свободно перемещаться по офису и подсаживаться к тем, с кем они работают.

Сотрудникам организации должно казаться естественным, что инженер-безопасник садится рядом с разработчиком и вместе с ним занимается диагностикой ошибки или объясняет, как устранить уязвимость. Это простое действие наглядно демонстрирует, что группа стремится оказывать содействие и поддержку, хочет понять нужды людей и готова способствовать успеху в их начинаниях, не жертвуя безопасностью.

Информационные панели

Во многих командах имеются информационные панели, которые выводятся на большой экран и показывают общие тенденции и текущие события. Понятно, что группа безопасности – не исключение. Однако стоит отметить, какую роль информационные панели могут сыграть в культуре безопасности, а также их двойное назначение. Если вы будете лучше понимать, достижению каких целей могут способствовать информационные панели, то сможете принять более правильные решения о том, какие панели лучше всего подходят для вашей среды.

Информационные панели можно приблизительно разбить на четыре категории:

1) ситуационные панели;
2) сводные панели;
3) показушные панели;
4) аварийные панели.

На *ситуационных панелях* отображаются данные, интересные прежде всего самой группе безопасности. Здесь показано текущее состояние среды или системы, а также столько исторических и контекстных данных, сколько помещается. Это позволяет понять, происходит ли что-то

необычное или нет. Основная цель ситуационной панели состоит в том, чтобы группа безопасности с одного взгляда могла определить, ведет ли себя система ожидаемым образом. Отметим, что ситуационная панель показывает только, что что-то не в порядке, и нуждается в расследовании, но не говорит ничего определенного о возможной атаке. Это простой способ распознать симптомы. А дальше группа должна вмешаться и установить причину.

На ситуационной панели может, например, отображаться «количество удачных и неудачных попыток входа в систему за период» (рис. 15.1), «количество SMS, вызовов и push-кодов двухфакторной аутентификации» или «количество зафиксированных попыток межсайтового скриптинга за период» (рис. 15.2). С точки зрения культуры, достоинство ситуационных панелей – в том, что, глядя на них, люди испытывают уверенность в том, что «безопасность систем всегда находится под контролем и группа готова к действиям». Они также могут подвигнуть на контакт с группой безопасности, особенно если отображаются графики и заметные флуктуации настораживают. Видя, что графики сбесились, человек часто любопытствует, что стало причиной и что делается в ответ.

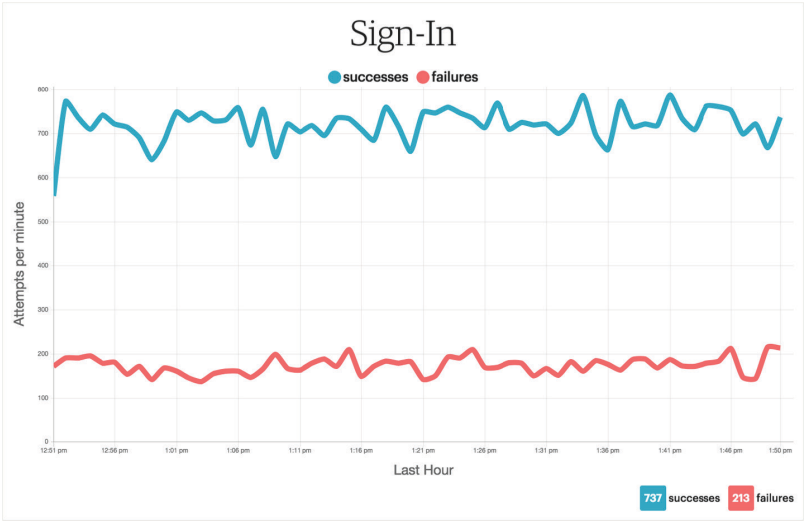

Рис. 15.1. Пример ситуационной панели, на которой отображается количество успешных и неудачных попыток входа в веб-приложение. Соотношение этих показателей может свидетельствовать о различных ситуациях, требующих внимания со стороны группы безопасности. Например, увеличение количества неудачных попыток, по сравнению с удачными, может быть признаком атаки путем подбора пароля. Уменьшение количества удачных попыток может свидетельствовать о проблеме в системе аутентификации, когда законные пользователи не могут войти в систему

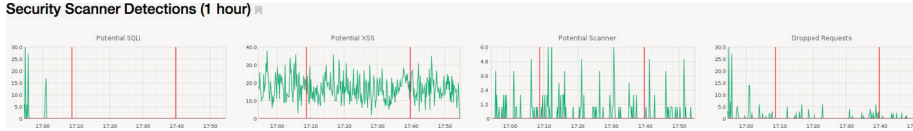

Рис. 15.2. Еще один пример ситуационной панели – отображается количество атак или обнаруженных сканером уязвимостей по типам. Пики и продолжительные отклонения от нормального поведения должны побудить группу безопасности провести расследование

Если на ситуационной панели отображается информация, идентифицирующая пользователя, например имя пользователя, то следует заменить ее псевдонимом, чтобы ненароком не возложить на кого-то вину и не выставить на посмешище. Например, такое маскирование данных уместно, если на панели показываются недавние входы через VPN и место, из которого произведен вход. Даже подмененный идентификатор пользователя полезен, когда нужно показать события, требующие внимания группы безопасности, не раскрывая истинных имен, ассоциированных с событием. В процессе дальнейшего расследования можно установить, какому реальному пользователю соответствует псевдоним.

На *сводной панели* отображаются суммарные данные за период или справки для руководства. Например, можно было бы показать «общее количество сообщений о фишинговых письмах за последние четыре недели», «сумму в долларах, выплаченную в рамках программы вознаграждения за найденные ошибки в текущем году» или «количество открытых внешних портов, обнаруженных при последнем сканировании». Это сводные данные за прошедший период, зачастую вообще без информации о трендах или с минимальной информацией, например изменение неделя к неделе.

Из четырех типов информационных панелей сводные панели, пожалуй, наименее полезны для группы безопасности, поскольку не несут информации о текущем состоянии безопасности системы или среды и ничего не говорят о том, какие действия следует предпринять. Их ценность с точки зрения активизации контактов других сотрудников с группой безопасности также минимальна. Сводные панели не несут информации, кроме разве что «Глядите! Мы тут тоже делом занимаемся». То, что они сообщают, лучше изложить в итоговом электронном письме или в справке, периодически отправляемой тем, кому это нужно знать. Если не хватает места, то первое, что стоит убрать, – это сводные панели.

Какое сочетание информационных панелей наиболее разумно в вашей организации, решить можете только вы, но имеет смысл собрать статистику о взаимодействиях, пусть даже забавных, которые порождают

различные сочетания панелей. Если панели, отображаемые на большом экране, доступны также в интранете, то количество просмотров может стать хорошим показателем интереса, который вызывают различные панели.

На *показушных панелях* отображаются данные или визуализации, которые не имеют почти никакого отношения к повседневной работе группы безопасности. Но они выглядят круто и привлекают внимание проходящих мимо. Люди останавливаются, смотрят и, быть может, даже зададут вопрос о том, что здесь показывается.

Показушные панели содержат меньше всего данных из всех информационных панелей, их задача – красиво выглядеть, а не показывать данные, на основе которых можно действовать. Типичные примеры – трехмерный глобус, на котором показана география источников атак, или самые распространенные на текущий момент угрозы, обнаруженные в почтовых сообщениях и заблокированные. Многие поставщики продуктов, связанных с безопасностью, бесплатно предлагают такие анимированные панели на своих сайтах в надежде, что анимация, как в фильме «Пароль “Рыба-меч”», побудит купить у них что-нибудь. Как бы то ни было, они могут стать хорошим источником показушных панелей, и никаких усилий для этого прилагать не понадобится. Для тех, кто хочет создать собственную озвученную карту, имеется проект с открытым исходным кодом pewpew (*https://github.com/hrbrmstr/pewpew*), который может послужить отправной точкой и предоставляет многочисленные возможности настройки и усовершенствования (см. рис. 15.3). Создание интересных, постоянно изменяющихся карт – непринужденный способ выстроить позитивную культуру безопасности – и для самой группы безопасности, и для тех, кто остановится взглянуть на результаты.

Несмотря на броскую внешность и малое информационное содержание, показушные панели могут оказаться наиболее ценными с точки зрения культуры, поскольку играют на интересе людей к безопасности (или к *хакерам*). Они также открывают возможность вступить в контакт с группой безопасности – ведь многие захотят поговорить о том, что это за лазерные выстрелы идут из Восточной Европы к нашему центру обработки данных. Это также единственный вид панелей, который может заставить человека остановиться на несколько секунд, чтобы посмотреть, и тогда у безопасников появится шанс немного поболтать о недавнем срабатывании сигнала на его IP-адрес – нет-нет, ничего серьезного, просто интересно.

Из четырех типов информационных панелей только показушные панели следует время от времени менять, чтобы интерес к ним не угасал.

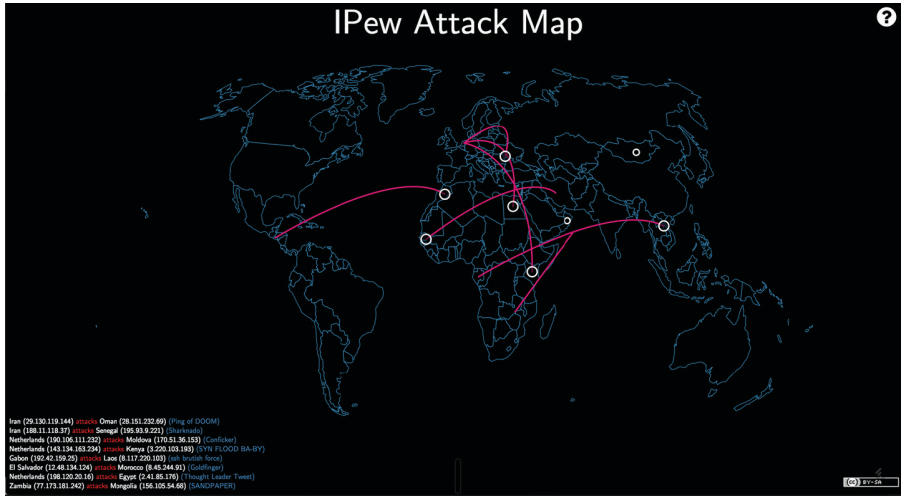

Рис. 15.3. Пример карты PewPew, публикуется с разрешения проекта pewpew

Аварийные панели, наверное, самые простые из всех и одновременно самые редкие (хочется надеяться). Их единственное назначение – уведомить группу безопасности о критической ситуации, требующей немедленного вмешательства; они срабатывают в ответ на заранее сконфигурированное событие и должны приостанавливать показ всех остальных панелей. Например, таким событием может быть утечка критических учетных данных на GitHub или взаимодействие с внутренней медовой ловушкой.

У аварийных панелей есть два преимущества над другими типами уведомлений и оповещений, которыми и так уже пользуется группа безопасности. Во-первых, это публичное коллективное уведомление. Если срабатывает аварийная панель, то вся группа одновременно получает сигнал и может назначить ситуации приоритет по отношению к другим текущим делам. Во-вторых, и это, пожалуй, более важно, аварийная панель показывает всей организации, что происходит что-то серьезное и группа безопасности занята, пытаясь понять и (или) отреагировать на ситуацию. И хотя второй момент может показаться самоочевидным, тот факт, что люди, не принадлежащие к группе безопасности, знают о серьезной угрозе, не только вызывает сочувствие к работе группы, но и открывает возможности для сотрудничества и просвещения в вопросах безопасности в будущем, коль скоро люди интересуются, что происходит.

Дадим несколько советов об аварийных панелях. Их следует зарезервировать для самых критических из отслеживаемых событий безопас-

ности, появляться они должны редко, и каждое появление должно быть равнозначно сигналу «Аврал! Все на борт!». Сообщение на аварийной панели должно побуждать к немедленным действиям, в идеале для него должен быть готовый сценарий, в котором описаны начальная реакция и порядок сбора информации. Плотность данных на аварийной панели должна быть низкой, должны присутствовать четкое описание события, временная метка и, возможно, ссылка на сценарий или первые действия в ответ на событие. Внешний вид панели более-менее безразличен, лишь бы она бросалась в глаза и выделялась на фоне других сменяющихся панелей. Подойдет, к примеру, красный экран с мигающей белой надписью шрифтом Arial или изображение голодного медоеда (правда, медоед покруче будет).

Чтобы информационные панели присутствовали и приносили пользу, необходимо вести мониторинг и собирать данные. Различные подходы к мониторингу информации, относящейся к безопасности, описаны в главе 8.

Сухой остаток

Вот и завершилось наше культурное путешествие! Перечислим, о чем следует подумать, приступая к выстраиванию культуры безопасности.

- Какая-то культура безопасности уже существует, знаете вы об этом или нет. Вопрос в том, захотите ли вы принять владение ей.
- С точки зрения безопасности, люди не менее важны, чем технологии, но во многих традиционных подходах им уделяется недостаточно внимания. Чтобы культура безопасности давала эффект, центром решения должны быть люди.
- Ваша культура безопасности уникальна. Не существует одного решения, годящегося для всех.
- Содействие, прозрачность и отказ от поиска виноватых – вот три главных принципа культуры безопасности в компании Etsy, лежащие в основе всего остального.
- Относитесь к культуре безопасности как к бренду, принимайте ее со всей серьезностью и помните, что выстроить ее трудно, а утратить можно в одночасье.

Глава 16

Что такое гибкая безопасность?

Словосочетания «гибкая методика» и «гибкая безопасность» несут разный смысл для разных людей. И осуществить их можно по-разному.

У каждого из нас есть свой, отличный от других опыт гибкой безопасности и своя история. Мы сталкивались с разными проблемами и находили разные решения. И кое-чем из этого хотели бы поделиться с вами.

История Лауры

Мой путь сюда был извилистым, а карьера складывалась в основном из интуитивных озарений и вопросов «Что, если?», заданных в самый неподходящий момент.

Мои друзья и сообщество по безопасности в Новой Зеландии знают меня как «пастуха котов» и силой хаоса (хочется надеяться, полезного), но не как человека, которого понимают с самого начала.

Но я отвлеклась. Позвольте рассказать, как я очутилась здесь (прошу вас, не уходите, это важно, честное слово).

Не инженер, а хакер

Я из семьи, где делают вещи. Мы создаем все: мосты, вертолеты, биохимические препараты. Это у нас в крови. Мы не очень-то уважаем правила и формальности, принятые в отрасли, но каждый из нас прекрасно умеет задавать вопрос «Как я могу сделать *x*?», а потом отвечать на него делом. Мы – преисполненные решимости бойцы, которые обожают делать вещи и делают их лучше, быстрее и крепче.

В моей семье это называли хакерством, у меня никогда не возникало вопросов по этому поводу.

По разным причинам моя детская мечта стать Скалли из сериала «Секретные файлы» не сбылась, и в 17 лет я стала работать программистом-стажером (на COBOL). То был 2001 год.

Неожиданно приобретенные в семье навыки конструирования всяких штучек и починки того-сего обернулись кодом. Я научилась создавать системы и познакомилась с системами налогообложения и банками. Я была в своей стихии.

Шли годы, я училась и окончила университет, попутно нанизывая цепочку интересных мест работы. Я работала в ЦЕРНе в Швейцарии, в правительственных агентствах Великобритании и как-то даже провела странное лето, подвизавшись в качестве веб-разработчика. Я писала производственный код на разных языках, создавала особо ответственные в плане безопасности системы и роботов. Жизнь была полна приключений.

Беда в том, что под этой тонкой шкуркой я по-прежнему осталась целеустремленным хакером, и это давало мне суперсилу. Я отлично умела искать дефекты в программах, особенно относящиеся к безопасности. Я видела, как мог сбиться с курса рабочий процесс, и нутром чуяла всякие странные граничные случаи, которые забыли рассмотреть.

Но это делало меня не полезным членом команд, а источником проблем.

Всюду, где я появлялась, возникали задержки и осложнения. Не это требовалось от разработчика программного обеспечения.

Твое дитя – уродец, и ты должен чувствовать себя виноватым

Ну, что ж, раз обнаруживается, что у тебя лучше получается ломать программы, чем создавать элегантно спроектированные решения, значит, нужно подыскать для себя подходящую нишу.

Для меня такой нишей стало участие в команде красных и тестирование на проникновение.

Я потратила годы, взламывая программы по всему миру и объясняя напряженно трудившимся разработчикам, что их новое дитя – уродец и что они должны чувствовать себя виноватыми. Я научилась делать то же самое с человеческими системами, используя социальную инженерию и манипулирование.

Поначалу это было забавно, но я отдалялась от своих корней. Я больше не создавала вещи и не решала задач. Я разрушала.

И самым худшим было то, что я понимала, как делают ошибки. Я достаточно долго работала программистом и к тому времени прекрасно осознавала, какие условия приводят к небезопасному коду, и сочувствовала бедолагам. Медленно и постепенно все это накапливалось и соби-

ралось в кучу, пока я наконец не пришла к выводу, что одним взламыванием систем ничего не изменишь. Мы должны начать с изменения способа создания программ.

Поменьше говори, побольше слушай

В 2013 году я занялась исследовательской работой. Я должна была понять, почему так много окружавших меня программистов пишут уязвимый код и почему так мало команд занимаются безопасностью. В течение шести месяцев я покупала разработчикам и менеджерам чай с пирожными и задавала один и тот же вопрос.

Как обстоит дело с безопасностью в твоем мире?

Ответы были удручающими.

Раз за разом смысл послания становился все яснее. Все всё понимали. Мы знали, что должны думать о безопасности, но предлагаемые средства и методы не работали. Они были либо менторскими, либо неподходящими, субъективными и медленными.

Я встречалась с людьми, которые хотели делать свою работу лучше, хотели создавать потрясающие воображение продукты, и им мое сообщество (мир безопасности) представлялось нелюдями, которые приходят незваными, хамят и доставляют страдания.

Что-то нужно было менять.

Давайте двигаться быстрее

В 2014 году я основала компанию SafeStack. Я ставила себе целью помочь компаниям двигаться быстрее, не жертвуя безопасностью. Компания была основана в августе, и я положила себе до декабря найти клиента. Если бы это не получилось, я устроилась бы на настоящую работу.

Старые взгляды на управление разработкой и подходы на основе контрольно-пропускных пунктов были отправлены на свалку (они и так вот уже 15 лет как не работали), и мы начали с чистого листа.

Первым шагом было строительство с учетом выживания, мы сосредоточили усилия на реагировании на инциденты и узнавании о наличии проблем.

Затем мы рассмотрели все стандартные средства контроля, которые рекомендовали бы в старом подходе. Мы выделяли из каждого истинные цели (чего пытались достичь авторы) и искали простейшие способы автоматизации или внедрения. Мы называли это минимальной жизнеспособной безопасностью.

Это был бесцеремонный, практический, скудный на документацию подход, опирающийся на прагматизм и идею о том, что лучше со временем постепенно улучшать безопасность, чем пытаться сделать всё и сразу.

Я вернулась к своим корням. Я стала хакером в том смысле, в котором меня научила семья, но с намерением сделать жизнь людей безопаснее. Я не собиралась возводить собственный цифровой храм. Я для этого не гожусь. Я собиралась помогать десяткам других людей создавать удивительные творения и защищать их.

Проблема заключалась в том, что я была одна. К декабрю 2014-го я сотрудничала с 11 организациями, и на подходе было еще несколько.

Мне нужна была помощь. Я сколотила команду, и мы наняли разработчиков и инженеров, чтобы те помогали решать проблемы. В конце концов, это их мир, они рождены, чтобы находить решения.

Создание круга поклонников и друзей

Во всем этом была одна фундаментальная идея – нам (группе безопасности) больше не пристало быть наделенным особой властью раздражителем. Мы должны были стать всеми любимой кастой помощников и пособников. Нам нужно было создать круг поклонников, друзей и пропагандистов, которые стали бы нашими глазами и ушами внутри команд. Только так можно было обеспечить масштабирование.

Все это должно показаться вам знакомым. Все, чем я поделилась в этой книге, вытекает из того опыта.

Мы невелички, но нас много

Начиная с 2014 года я помогла осуществить эту миссию более чем 60 организациям в 7 странах, от крохотных команд, насчитывающих всего 4 человека, до гигантских корпораций и государственных учреждений.

В одних случаях ресурсов было немерено, в других не было ничего, но мы могли сами написать скрипты и сляпать что-то на коленке. Зачастую в организации не было ни группы безопасности, ни начальника отдела информационной безопасности и вообще никого, кто специально занимался бы безопасностью. Здесь я чувствовала себя как рыба в воде. Здесь я пасла своих котов.

Для обеспечения безопасности необязательно быть богатым и располагать огромными ресурсами.

Безопасность – это о коллективном выживании. Если рядом с вами работает команда союзников, то можно добиться потрясающих результатов и двигаться вперед быстро, не жертвуя безопасностью и невзирая на обстоятельства.

История Джима

Мой опыт по большей части связан с разработкой, эксплуатацией и управлением программными проектами. Я никогда не работал ни тестировщиком проникновения, ни хакером. Мое место всегда было на другом конце поля – играть в обороне, строить, а не разрушать. В течение 20 с лишним лет я работал на финансовых рынках, разрабатывая платформы электронной торговли и клиринга для фондовых бирж и банков по всему миру. В этих системах безопасность – конфиденциальность и целостность данных, доступность служб и соответствие нормативным требованиям – так же важна, как скорость, масштабируемость и функциональность.

Но за это время наши представления о безопасности и методы работы значительно изменились. Мне доводилось руководить большими программами, занимавшими несколько лет. В начале – бесконечный сбор требований и планирование, в конце – нескончаемое тестирование и стресс.

Безопасность тогда сводилась в основном к проектированию центра обработки данных и архитектуры сети. Если вы обустроили свою закрытую среду правильно, то плохие парни не могли попасть внутрь, а если бы попали, то ваши средства наблюдения и анализа мошеннических операций поймали бы их.

Все изменилось с пришествием Интернета и облака. А также с появлением гибкой разработки, а теперь еще и DevOps.

Сегодня мы внедряем изменения намного, намного быстрее. Циклы выпуска измеряются самое большее днями, а не месяцами или годами. И безопасность стала проблемой разработчиков не в меньшей мере, чем системных инженеров, группы построения сети и владельцев ЦОДов. Разработчики уже не могут спихнуть безопасность на кого-то другого. Мы помним о безопасности все время: при сборе требований, проектировании, кодировании, тестировании и внедрении.

Хочу поделиться несколькими важными вещами, которые я узнал о безопасности, работая в мире гибких методик.

Вы можете вырастить собственных экспертов по безопасности

В этой книге мы много говорили о формировании гибкой группы безопасности и о том, как эта группа должна работать с техническими подразделениями. Но можно построить безопасную систему и эффективную программу обеспечения безопасности и без специальной группы безопасности.

В организации, которой я помогаю руководить в настоящее время, нет независимой группы безопасности. Я – владелец программы безопасности, но ответственность за ее реализацию делят разные технические группы. Я понял, что если лишить безопасность правового и нормативного обрамления и превратить в набор технических задач, которые необходимо решить, то ваши лучшие технические кадры с энтузиазмом примут вызов.

Чтобы эта идея заработала, нужно дать людям достаточно времени на учебу и достаточно времени, чтобы сделать работу хорошо. Обеспечьте им помощь, инструменты и обучение. Но самое главное – нужно объявить безопасность инженерным приоритетом и подкреплять слова делом. Если вы будете пренебрегать безопасностью или срезать углы, чтобы уложиться в сроки или снизить затраты, то преданность делу пойдет прахом, и на программе безопасности можно будет ставить крест.

Это не разовый подвиг, совершаемый вами или командой. Нужно терпеливо и неутомимо повторять, как важна безопасность. Нужно, чтобы люди не боялись относиться к неудачам как к возможности обрести новый опыт. И обязательно нужно подкреплять успехи.

Когда инженеры и руководители команд перестанут считать тесты проникновения и аудиторские проверки дурацкой юридической формальностью, а увидят в них важную возможность учиться новому и совершенствоваться, станут рассматривать их как вызов своим знаниям и умениям, вот тогда прогресс будет налицо.

Не каждый будет «обеспечивать безопасность», но не каждому это и нужно. Конечно, основам нужно обучить всех, чтобы все знали, чего не нужно делать и как не разочаровать аудиторов. Нескольких часов, разбитых на непродолжительные занятия, будет достаточно, и это, наверное, максимум, с которым все готовы будут смириться. Поскольку большинству инженеров только и нужно знать, какие каркасы и шаблоны использовать, не забывать про секретные данные и научиться полагаться на свои инструменты.

Понимать, что такое безопасность на более глубоком техническом уровне, должны те, кто создает эти каркасы и шаблоны и реализует эти инструменты. Именно они должны активно участвовать в вашей программе безопасности. Именно их нужно посылать на конференции OWASP и на углубленные учебные курсы. Именно они будут работать с тестировщиками проникновения, аудиторами и другими специалистами по безопасности, бросая вызовы и принимая брошенные вызовы. Именно эти люди смогут осуществлять техническое руководство

командой, подкреплять правильные практики и следить за тем, чтобы команда принимала верные технические решения.

Вам не достичь всего этого без опоры на кого-то, кто на безопасности собаку съел, по крайней мере на начальном этапе. Наймите эксперта по безопасности, чтобы при проектировании и выборе платформы он с самого начала помог команде понять, какие существуют угрозы и риски безопасности и как с ними обходиться. Как было сказано в этой книге, вам, вероятно, придется время от времени обращаться к сторонним специалистам для контроля и корректировки, чтобы не сбиться с пути и оставаться в курсе новых угроз.

Лучшую реализацию этого подхода я наблюдал в программе «пояса карате» по безопасности (*http://www.adobe.com/content/dam/Adobe/en/security/pdfs/adb_security-culture-wp.pdf*) в компании Adobe. Все члены технической команды должны были иметь как минимум белый пояс, а лучше зеленый – т. е. знать достаточно, чтобы не попасть в беду. У кого-то в команде должен быть коричневый пояс – достаточно знаний, чтобы защититься от реальных атак. И на несколько команд должен приходиться один черный пояс – мастер, у которого можно спросить совета и руководства, когда команда не знает, что делать, и который может вытащить команду из беды.

Этот подход масштабируется как на небольшие команды, так и на гигантские организации. Наверное, это единственный способ масштабировать безопасность в гибких командах.

Выбирайте людей, а не инструменты

Невозможно быстро продвигаться вперед без инструментов. Но будьте осторожны, для одних платформ и проектов лучше подходят одни инструменты, для других – другие. Не бывает одного инструмента для всего на свете.

С инструментами так – чем проще, тем лучше. Можно взять инструменты для решения узких задач и сцепить их в сборочном конвейере. Инструменты, которые легко установить и использовать, которые быстро работают и допускают управление через API, полезнее стандартизованных корпоративных платформ с информационными панелями и встроенными отчетами о соответствии нормативным требованиям.

За последние 10 лет инструменты безопасности сильно улучшились, стали более точными, легче интегрируются, проще для понимания, быстрее и надежнее. Но одним инструментом сыт не будешь, как мы не раз видели в этой книге. Все зависит от людей (правильно подобранных).

Толковые люди, которые с самого начала принимают хорошие решения, задают правильные вопросы и думают о безопасности в процессе написания технического задания и проектирования, способны решить многие важные проблемы безопасности для команды. Опасные уязвимости типа внедрения SQL, атак CSRF и XSS и нарушения контроля доступа можно предотвратить, встроив защиту в каркасы и шаблоны и позаботившись о том, чтобы все использовали их как положено. Сделайте так, чтобы программистам было легко писать безопасный код, – и вы получите безопасный код.

Пользоваться всеми преимуществами автоматизированного сканирования и непрерывного тестирования необходимо. Но и ручные инспекции играют немаловажную роль: анализ требований, проекта и кода с целью обнаружить ошибки понимания и реализации. То же относится и к ручному тестированию. Правильно поставленное исследовательское тестирование, прогон реальных сценариев и охота за ошибками могут найти такие проблемы, которые автоматизированное тестирование не замечает, и подсказать, где в инспекциях и комплектах тестов имеются существенные упущения.

Учитесь работать со своими инструментами. Но только люди смогут вытащить вас из беды.

Безопасность должна начинаться с качества

В безопасности далеко не всё – черная магия. Нужно просто быть внимательным и аккуратным, думать о требованиях и писать хороший, чистый код.

Уязвимости – это дефекты. Чем больше в программе дефектов, тем больше уязвимостей. Перечитайте еще раз написанное в книге о таких серьезных дефектах безопасности, как Heartbleed и Apple Goto Fail. Они стали результатом плохого кодирования, небрежного тестирования или того и другого сразу. Ошибки, которые команда, не работай она в неподходящих условиях, могла бы найти в процессе инспекции кода или тестирования. Или предотвратить, следуя хорошим наставлениям по кодированию и дисциплинированно применяя рефакторинг.

Защитный образ мыслей, готовность к неожиданному, защита своего кода от чужих (и собственных) ошибок, проверка корректности данных, надлежащая обработка исключений – всё это сделает систему более надежной и устойчивой к ошибкам во время выполнения, а стало быть, и более безопасной.

Как многое в программной инженерии, это легко понять, но трудно правильно применить. Требуются дисциплина, аккуратность и время. Но в итоге вы получите более качественный и безопасный код.

Соответствие нормативным требованиям может стать повседневной практикой

Соответствие нормативным требованиям – неизбежное, но необходимое зло в таких отраслях, как финансовые рынки или здравоохранение. Но можно сделать это на своих условиях. Для этого нужно покинуть поле правовых политик и общих руководств по соблюдению нормативных требований и встать на почву конкретных требований и утверждений, задач, которые инженеры могут понять и решить, и тестов, которые необходимо добавить.

Комплект инструментов DevOps Audit Defense Toolkit, который мы рассматривали в главе о соответствии нормативным требованиям, – вещь, о которой должно знать больше народу. Применение его в качестве руководства поможет понять, как встроить соответствие требованиям в код, перенеся его из документов и электронных таблиц в сборочный конвейер и стандартные рабочие процессы эксплуатации. Реализация политик безопасности и соответствия непосредственно в коде и автоматизация тестов и сканирования, так чтобы эти политики гарантированно соблюдались, занимает время, но вы можете воспользоваться средствами автоматизации, которые команды, практикующие гибкие методики и DevOps, и так уже применяют. Ну и, снова повторим, инструменты становятся все лучше.

Стоит внедрить этот подход, как всё сразу меняется. Вы знаете (и можете доказать), что каждое изменение каждый раз проходит одну и ту же обработку. Вы знаете (и можете доказать), что все тесты и проверки были выполнены – каждый раз. В процесс встроены аудитопригодность и полная прослеживаемость: вы можете сказать, что было изменено, кем и когда – каждый раз. Соответствие нормативным требованиям становится просто еще одной частью повседневной работы. Конечно, аудиторы все равно захотят увидеть документально оформленные политики. Но ведь вы можете доказать, что всегда выполняете прописанные в них тесты, проверки и правила. Это бесценно.

История Майкла

Для меня, как и для моих соавторов, не вполне понятно, как я тут оказался. Я стоял на сцене перед сотнями людей, которым объяснял, как

правильно обеспечить безопасность, а про себя думал: «Какое у меня право поучать кого-то?»

Я работал в разных отраслях: с конструкторами оборудования, с людьми, которые в начале 2003 года программировали микропроцессоры Z80, с финансовым стартапом, где разрабатывал веб-серверы с малым временем задержки, с компанией по разработке игр, в которой писал сетевой код для приставок XBox, Playstation и PSP. В конечном итоге я оказался в газете *Guardian* как раз в тот момент, когда они начинали одну из самых крупных и современных программ с использованием гибких методик.

Я следовал за миром гибких технологий на протяжении большей части своей карьеры, разрабатывал код на C++ через тестирование еще в мире Bingo, а в финансовом стартапе практиковал ранние варианты Scrum, как в книжке написано (30-дневные спринты, неизменяемые журналы пожеланий – в общем, весь набор).

И вот теперь я стал членом команды, поставлявшей ПО с применением экстремального программирования, и работал с лучшими идеологами гибкого подхода из компании Thoughtworks.

Проект *Guardian* начинался как типичный XP-проект, но потом мы стали экспериментировать. Мы создавали информационные излучатели и изучали ценность систем сборки. Я работал и с разработчиками, и с системными администраторами, стараясь понять, что еще можно автоматизировать в наших ручных системах. Наверное, я тратил больше времени на возню с окрестными каркасами, которые писали код, чем писал код сам.

За семь лет мы превратили команду из беспорядочной организации, которая подключалась по SSH к серверу разработки, писала код в редакторе VIM и производила развертывание путем копирования по FTP кода из репозитория в производственную систему, в организацию, которая пользовалась распределенной системой управления версиями, облачными службами AWS и автоматически производила развертывание сотни раз в день.

Каждый шаг предпринимался, для того чтобы разрешить какую-то внутреннюю проблему. В нашей команде подобрался такой пестрый состав специалистов и умений, что мы могли вернуться на шаг назад и вместо небольшого итеративного улучшения задаться вопросом, а нельзя ли сделать что-то более фундаментальное и выиграть больше.

Оттуда я перешел в Правительственную цифровую службу Великобритании. Предполагалось, что я буду исполнять функции технического специалиста и эксперта по гибким методикам. Я знал, как строить

хорошо масштабируемые системы в облаке, и умел создавать команды, которые не просто исповедовали гибкий процесс, но могли активно адаптировать его и эффективно применять.

Я уже позабыл свой юношеский роман с безопасностью, но если что и присутствует в правительстве в избытке, так это безопасность.

Все команды, работавшие на правительство, боролись с одними и теми же проблемами. Они хотели скорее приступить к созданию ПО и быстро поставлять его, но правительственные механизмы безопасности были непрозрачны, и наладить работу с ними было трудно. Я встречал команды, которых проинструктировали о том, что IP-адреса производственных систем – закрытая информация и знать им об этом не положено. Я встречал команды, которым было сказано, что в правительственном руководстве по безопасности (закрытая информация, ознакомиться нельзя) написано, что им разрешается развертывать ПО только путем записи на компакт-диск. Я встречал группы безопасности, которые настаивали на том, чтобы разработчики писали только на C или C++, потому что – вы уже догадались – того требует секретное руководство по безопасности.

В моем арсенале было два секретных оружия. Поскольку у меня был достаточно высокий допуск, я имел право прочитать эти руководства, и Правительственная цифровая служба начала выстраивать отношения с ребятами из Центра правительственной связи, которые эти руководства писали.

Мы выяснили, что многие рекомендации утратили актуальность, что некоторые относились к определенному контексту (языки C и C++ рекомендовались только для программирования криптографических операций во встраиваемом оборудовании – это руководство никогда не предназначалось людям, разрабатывавшим веб-приложения!) и что налицо недопонимание и неправильное применение.

С этим мы разобрались. Так, шаг за шагом, я посетил почти все правительственные департаменты, познакомился с тамошними группами безопасности и понял, что с годами навыки обеспечения безопасности систематически утрачивались. Редко-редко можно было встретить технолога, который разбирался в правительственной безопасности, или безопасника, владевшего современными технологиями.

За время моей работы в качестве программиста отрасль перешла от каскадной модели к гибким методикам, от поставки ПО в виде пакетов к непрерывному развертыванию, от монолитных систем к перекрестно связанным сетям систем. Очень немногие безопасники в правительстве имели время или желание идти в ногу с этими изменениями.

Я решился стать проводником изменений, человеком, который разбирается и в безопасности, и в технологиях! В своем впустую растраченном детстве я имел случай познакомиться с повадками хакеров, а уж разработкой занимался даже дольше, чем нужно; я знал, как устроены технологии и как все приходит в движение.

Я пришел к выводу, что мы приближаемся к крайне опасному моменту в истории компьютеров. Уровень использования компьютерных систем выше, чем когда-либо в истории, и продолжает расти. Скоро мы увидим кардинальное изменение количества и масштаба взаимосвязанных систем, и как при этом изменятся сами системы, никому не известно. Автомобили, связывающиеся между собой и с городом, интернет вещей, компьютеризованное здравоохранение – это только вершина айсберга.

Распространение знаний о безопасности не поспевает за такими темпами. В то время как ярчайшие умы изучают эти проблемы, подавляющее большинство безопасников обеспокоено проблемами из начала 1990-х годов. Мы все еще ругаем пользователей за переходы по ссылкам в фишинговых письмах, мы требуем паролей, которые невозможно запомнить, и все еще настаиваем на бумажных дипломах, подтверждающих знания о принципах безопасности 1960-х годов (Белл и ЛаПадула, привет вам).

Сформулирую четыре главных принципа.

Знания о безопасности распределены неравномерно

Лучшие разработчики все еще пишут код, содержащий простейшие ошибки, связанные с безопасностью. Уровень знаний по безопасности, преподаваемых младшим и начинающим программистам, увы, неадекватен – они не знают даже о внедрении SQL и переполнении буфера.

Далее, языки и каркасы возлагают обязанность разбираться на разработчика, который не заинтересован или неспособен сделать правильный выбор. Большинство криптографических библиотек позволяют разработчику осознанно сделать небезопасный выбор или, хуже того, поставляются с небезопасными умолчаниями и примерами кода.

Можно было бы требовать, чтобы все программисты были экспертами по безопасности, но это не реализуемо. Для большинства людей безопасность – вещь скучная и неинтересная. К тому же, как во многих других случаях, нужно много времени, чтобы подняться с уровня ученика, который слепо делает то, что сказано, на уровень, где становятся видны и понятны закономерности и где можешь творчески применять свои знания в новых ситуациях.

Вместо этого нам нужно, чтобы безопасность была встроена по умолчанию в инструменты, языки и каркасы. Нам нужно, чтобы лучшие умы в области безопасности посвятили себя созданию инструментов, одновременно удобных и безопасных. Нам нужно, чтобы работать безопасно и правильно было гораздо легче.

Практическим специалистам нужно периодически проходить повышение квалификации

Мне доводилось бывать на встречах с экспертами по безопасности, которые говорили, что проработали в этой области десятки лет, и при этом убеждали меня, что каждое JSON-сообщение в архитектуре внутренних микросервисов необходимо пропустить через специальный компьютер с антивирусом.

Технические знания этих людей безнадежно устарели, они, вероятно, не в состоянии понять проблем, возникающих из-за того, что они настаивают на применении паттернов, которые им непонятны, в технологическом контексте, который им неведом.

Специалист по безопасности, работающий с современной технологической системой, должен понимать, как выглядит облачная архитектура. Он должен знать, в чем разница между отправленными пользователем данными и сгенерированными машиной сообщениями. Он должен понимать, как быстро вносятся изменения и производится развертывание и как в наше время легко использовать код повторно.

Учитывая, как трудно научить некоторых из этих людей особенностям управления серверами на платформе IaaS на основе Cattle, попытка объяснить, какие проблемы безопасности возникают при использовании технологии PaaS типа Kubernetes или, того хуже, функции как услуги типа AWS Lambda, скорее всего, ни к чему не приведет.

Раз эти люди не понимают технологий, которые призваны обезопасить, то нет никакой надежды, что они принесут организации пользу, содействуя изменениям и делая все, чтобы компания могла достичь поставленных целей.

Аккредитация и гарантии отмирают

Пакеты и системы старой школы, созданные по спецификациям, идеально сочетались с подходами на основе аккредитации и гарантий. Если можно сравнить ожидаемое поведение системы с фактическим, то возникает уверенность в том, как система будет работать.

Но по мере того как технология повышает уровень абстракции, а технические команды применяют все более сложные инструменты и платформы, возможности давать гарантии стремительно сокращаются.

Нам нужна какая-то замена этим возможностям. Мы должны понять, в чем состоит их предполагаемая ценность и как они должны измениться, чтобы соответствовать новым практикам.

В разработке ПО всегда будет оставаться место для этих механизмов. Я ни на секунду не допускаю, что система автоматического управления истребителем будет обновляться по беспроводному каналу несколько раз в час!

Но нужно знать, в каких случаях что применять, у нас должна быть возможность выбирать из множества вариантов тот, который даст необходимый уровень уверенности в системе.

Безопасность должна содействовать делу

Мне хотелось бы, чтобы эти слова стали эпитафией на моей могиле. Если безопасность вообще на что-то годна, то только для того, чтобы содействовать организации в выполнении ее миссии максимально быстро и безопасно.

Если людям кажется, что безопасность мешает их работе, то они будут искать обходные пути, игнорировать требования или отделываться формальностями, лишь бы отвязались. Мы не можем просто заставить бизнес склонить шею перед волей группы безопасности, любые попытки в этом направлении обречены на провал.

Задачей управления рисками должно быть создание условий, при которых организация может принять риски, а не стремление предотвратить всякий риск. Способная на это группа безопасности должна состоять из специалистов по технологиям и по безопасности, каждый из которых играет свою скрипку, а все вместе помогают организации.

История Рича

Мой путь сюда так же не похож на другие, как и у прочих авторов этой книги. В отрасли безопасности я оказался скорее благодаря удачному стечению обстоятельств, а не осознанному выбору. И этот путь не был прямым.

Ниже я кратко расскажу, как получилось, что меня пригласили написать книгу по безопасности. Собственно, это первый раз, когда я излагаю свою историю в письменном виде. И это позволило мне гораздо лучше понять самого себя – как развивались мои взгляды на безопас-

ность со временем и как я оказался там, где оказался. Надеюсь, что некоторые извивы моего пути найдут отклик в душе читателя, а если нет, то вы, по крайней мере, прочитаете забавный рассказ о том, как не надо идти в индустрию безопасности!

Первый раз бесплатно

Фанатом программирования я был с юных дней, первые программы на BASIC для компьютера Acorn Electron написал где-то в 7 или 8 лет. Не стану утомлять читателя описанием забавных игр и неблаговидных поступков, которые привели меня туда, где я смог честно зарабатывать на жизнь безопасностью. Но расскажу о том, что, как я уверен, стало первым успешно провернутым мной хаком.

Не помню точно, сколько мне было лет, наверное 8 или 9, а мишенью стал мой старший брат, которому в то время было 14 или 15. У нас был один компьютер Acorn на двоих, и, как часто бывает, он не хотел, чтобы младший брат играл в его игрушки. В данном случае это выразилось в запароливании дискет с играми, которые он копировал у одноклассников. А я ужасно хотел поиграть в них, так что у меня были все основания страстно желать обойти его защиту.

Атака была несложной. Слегка запутанный пароль был зашит в исходный код программы на BASIC, хранившейся на диске, и несколько команд файловой системы ADFS позволили мне прервать интерпретацию кода во время загрузки с диска, распечатать исходный код и вытащить пароль.

Я отчетливо помню, что, несмотря на тривиальность задачи, был на седьмом небе от счастья, когда добрался до игр, которые брат так старательно держал от меня подальше. И до сих пор я испытываю то же чувство, когда удается обойти препятствие, которое кто-то возвел на моем пути.

Спустя несколько лет щедрый коллега моего отца подарил мне старый модем, и я смог получить доступ к электронной доске объявлений местного интернет-провайдера, Diamond Cable, и там завис надолго. Я многому научился на практике, удовлетворяя невинное любопытство, а когда вступил в подростковый возраст, как раз подоспел Интернет, и я уже стал предметно интересоваться хакерством.

Недостаток формального образования в области компьютеров или программирования я успешно компенсировал знакомыми (в онлайне и в реале), которые щедро делились со мной знаниями и терпеливо помогали мне разобраться. Я часто задавал себе вопрос, как бы выглядел этот этап моего путешествия, если бы индустрия безопасности к тому времени уже сформировалась и на этом можно было бы зарабатывать.

Стали бы люди менее охотно делиться знаниями? Хотелось бы думать, что нет, но боюсь, что искреннее стремление исследовать и понять, как все работает, уступило бы желанию делать деньги и более строгим законам о неправомерном использовании компьютеров. Сейчас свободно доступной информации о хакерстве и безопасности гораздо больше, но я убежден, что тогдашняя скудость информации приносила куда больше удовлетворения от достигнутого, что и сообществу шло на пользу. Я считаю, что мне повезло взрослеть и узнавать о безопасности таким образом, поскольку электронные доски объявлений дали начало Интернету и всему, что за этим последовало. Если бы я начинал сейчас, то не уверен, что добился бы и половины моего нынешнего успеха.

А это может быть не просто хобби?

С таким-то началом моя деятельность в области профессиональной безопасности в течение очень долгого времени носила наступательный характер. Исследование уязвимостей, разработка эксплойтов и инструментов атаки, а также активное участие в тестировании на проникновение, командах красных и консультирование различных компаний в разных странах. В конце концов, это вылилось в создание небольшой компании хакеров старой школы, Syndis, со штаб-квартирой в Рейкьявике, Исландия, которая брала подряды на целенаправленные атаки.

Для меня безопасность была неразрывно связана с поиском новых способов взлома систем, и мне было мало дела до систематического устранения найденных мной уязвимостей, интересовали меня только сами уязвимости. Я был адептом церкви проблем. Оглядываясь на ранние ступени своей карьеры, с уверенностью могу сказать, что был образцом самых мерзких сторон стереотипного козла в области безопасности, от которых предостерегал вас на страницах этой книги. Не зря говорят: чтобы понять человека, нужно влезть в его шкуру.

Прозрение

Только после создания Syndis мои интересы сместились от чисто технического удовольствия от взлома и преодоления препятствий в сторону вызовов, которые несла задача построения оборонительных решений, куда было бы встроено реалистичное понимание того, как атакующий подходит к компрометации системы. Находясь во главе некоторых консультационных проектов по безопасности в Etsy, я обратил внимание, что подход *защита в ответ на атаку*, принятый в группе безопасности, хорошо гармонирует с методикой *целеустремленной атаки*. Оглядыва-

ясь назад, я понимаю, что это на самом деле две стороны одной медали, но тогда я этого не знал (да и не задумывался).

А затем произошло нечто странное – меня стали меньше занимать технические проблемы и самооценка в терминах уровня понимания какой-нибудь экзотической системы или технологии, а больше – происходящее на стыке людей и технологий. Зажегся свет, и я понял, что хотя мог считаться довольно успешным в выбранной области, но, сам того не понимая, видел и решал только половину проблемы. Мне стало как никогда ясно, что пользователь – это мягкое подбрюшье, которое с такой легкостью вскрывают методы социальной инженерии и нацеленные фишинговые атаки, является ключом к тому, чтобы добиться значимого прогресса в улучшении мира. Безопасность – это человеческая проблема, как бы усердно индустрия ни пыталась убедить себя в обратном и свести все к технологиям.

К тому времени я выполнил сотни подрядов на атаки, скомпрометировав десятки тысяч систем в разных отраслях и в разных странах, и всегда с одним и тем же результатом – я побеждал. При наличии достаточной мотивации и времени атака обязательно завершалась успехом, и заказчик получал подробный отчет, как именно его сложная система была обращена против него. За десять лет работы у меня не сложилось впечатления, что задача компрометации стала сложнее. Скорее, наоборот – росла уверенность, что компрометация неизбежна, оставалось только добавить примечания о том, как это было сделано в конкретном случае.

С компьютерами трудно, с людьми еще труднее

Так что же изменилось, после того как я прозрел? Очень просто, я стал лучше понимать те вызовы, которые связаны с человекоцентрическим взглядом на безопасность, и важность выстраивания особой и признанной культуры вокруг безопасности. Мне также стало гораздо яснее, что задача специалистов-безопасников – содействовать, а не препятствовать: почти незаметный когнитивный сдвиг, который, однако, ставит с головы на ноги функции безопасности и, что, пожалуй, важнее, восприятие безопасности теми, кто ее обеспечивает, и всеми остальными. Так, вместе с бесчисленными мелкими открытиями, которые я делаю до сих пор, были заложены основы для следующей ступени моего обучения тому, как сделать системы, а значит, и людей более безопасными. И хотя я по-прежнему получаю неизъяснимое удовольствие от нахождения и эксплуатации уязвимостей, ощущение полноты бытия теперь приходит от попыток найти решения, в центре которых – люди и удобство пользования.

И вот мы тут

Что ж, мне потребовалось немало времени, чтобы дойти до этого, но если вы теперь спросите меня, что такое гибкая безопасность, то я отвечу, что это ясное осознание того, что люди важны для безопасного решения не меньше, чем любая технология или математика. Только прогрессивный взгляд, согласно которому для безопасности людей нужно ставить во главу угла пряник, а не кнут, может привести к значимым изменениям. Это признание того, что до недавнего – постыдно недавнего – времени индустрия безопасности считала своей задачей продажу дорогих кнутов, которыми можно наказывать пользователей, не удосуживаясь при этом – ах, как удобно – измерить эффективность кнутов, но все равно трубя об успехе. Это понимание того, что, сосредоточившись на половине проблемы, никогда не найти полного решения, поскольку вы информированы в той же мере, в какой невежественны. Для меня гибкая безопасность имеет мало общего со Scrum, или бережливой разработкой, или непрерывным развертыванием, важно, чтобы в центре процесса разработки находились люди и чтобы были обеспечены эффективное взаимопонимание и сотрудничество.

Эта книга заставила меня задуматься и собрать воедино всё, чему я научился в своем путешествии в мир безопасности. В тот безумный период, когда писалась эта книга, я многому научился у Джима, Лауры и Майкла. Я искренне надеюсь, что это принесло вам пользу и хотя бы немного поможет на вашем собственном пути в мир безопасности. Спасибо, что, сами того не желая, стали частью моего.

Сведения об авторах

Лаура Белл – основатель и ведущий консультант в компании SafeStack, оказывающей услуги в области обучения безопасности, разработки и консультирования. Лаура – разработчик ПО и тестировщик проникновения, специализируется на управлении информацией и рисками безопасности в стартапах и организациях, практикующих гибкие методики. За последние десять лет занималась различными вещами, относящимися к обеспечению безопасности и разработке, принимая активное участие в создании производительных, масштабируемых и безопасных систем. Исторически сложилось, что служба безопасности была отделена от технических инноваций, однако Лаура разъясняет своим клиентам и более широкой аудитории, что в современном бизнесе этот подход уже не работает, потому что разработчики и эксплуатационники хотят сами понимать риски безопасности и справляться с ними.

Майкл Брантон-Сполл – главный архитектор безопасности в Правительственной цифровой службе, входящей составной частью в Секретариат кабинета министров Великобритании. Он помогает в написании и оценке стандартов безопасности и консультирует создание безопасных сервисов внутри правительства. Выполняет функции архитектора-консультанта для различных подразделений правительства, помогая им понять и внедрить гибкие методики, DevOps, наладить функционирование служб и выстроить современную веб-архитектуру. Прежний опыт работы Майкла включает новостную, финансовую, игровую и игорную индустрии.

Рич Смит – директор по исследованиям и разработкам в компании Duo Labs, занимающейся поддержкой передовых исследований по безопасности в интересах Duo Secutity. До перехода на работу в Duo Рич занимал должность директора по безопасности в компании Etsy, был одним из основателей исландского стартапа команды красных Syndis и занимал различные должности в группах безопасности в компаниях Immunity, Kyrus, Morgan Stanley и HP Labs. Рич профессионально занимается безопасностью с конца 1990-х годов: созданием организаций, специализирующихся на безопасности, консультированием, тестированием проникновения, созданием команд красных, исследованиями в области проведения атак, разработкой эксплойтов и инструментов

атаки. Он работал в общественном и частном секторе в США, Европе и Скандинавии, а в настоящее время проводит много времени в разъездах между Детройтом, Рейкьявиком и Нью-Йорком.

Джим Бэрд – технический директор, руководитель отдела разработки программ, менеджер проектов с более чем 20-летним стажем в области технологий финансовых сервисов. Работал с фондовыми биржами, центральными банками, клиринговыми центрами, регулирующими органами в области безопасности и торговыми компаниями в более чем 20 странах. В настоящее время занимает должность технического директора крупной альтернативной торговой системы в США.

Джим уже несколько лет практикует гибкие методики и DevOps в финансовой отрасли. Впервые он познакомился с инкрементной итеративной (пошаговой) разработкой в начале 1990-х годов, когда работал в технической компании на Западном побережье, занимавшейся разработкой, тестированием и поставкой программного обеспечения заказчикам в разных странах с частотой выпуска версий один раз в месяц. В то время он еще не понимал всей уникальности этого опыта. Джим активно участвует в жизни сообществ DevOps и AppSec, является одним из разработчиков открытого проекта безопасности веб-приложений (Open Web Application Security Project – OWASP) и выполняет функции аналитика для SANS Institute.

Об иллюстрации на обложке

На обложке изображен египетский желтоклювый коршун (*Milvus aegyptius*). По-латыни Milvus означает «коршун», а aegyptius – «из Египта», т. е. коршун из Египта. Достаточно точно, хотя эта птица встречается в большей части Африки.

Как следует из названия, птицу легко узнать по желтому клюву. У нее черный наряд, а ноги, лишенные перьев, желтого цвета, как и клюв. Длина взрослой особи примерно 55 см, размах крыльев – от 160 до 180 см. Желтоклювый коршун медленно парит в воздухе, расправив хвостовые и маховые перья, так что кажется неподвижным. Выполняющий функции руля хвост позволяет ему легко поворачивать. Полет грациозный, но птица способна на взрывное увеличение скорости.

Подобно многим хищным птицам, коршун питается в основном термитами. Он хватает термитов когтями и пожирает в полете. Желтоклювого коршуна также можно часто встретить на обочине дорог с небольшим движением, где он питается падалью. Городские районы также устраивают птицу, нередко можно увидеть, как коршун пикирует с неба и ворует еду у людей или у других птиц.

Желтоклювые коршуны моногамны на протяжении периода размножения. Если самец добывает достаточно еды, то самка может не охотиться в течение всего времени высиживания. Коршун строит чашевидное гнездо из прутиков в кроне подходящего дерева, выстилая его кусочками мягких материалов, которые удается найти. Самка откладывает два или три яйца белого цвета, инкубационный период продолжается примерно 35 дней. Птенцы покидают гнездо на 42–56-й день, но зависят от родителей еще от 15 до 50 дней.

Многие животные, изображенные на обложках книг O'Reilly, находятся под угрозой вымирания; все они важны для нашего мира. Если хотите узнать, чем вы можете помочь, зайдите на сайт *animals.oreilly.com*.

Предметный указатель

Р

С

Т

У

Ф

Х

Ц

Ч

Ш

Э

Книги изд тельств «ДМК Пресс» можно з к з ть в торгово-изд тельском холдинге «Пл нет Альянс» н ложенным пл тежом, высл в открытку или письмо по почтовому дресу: **115487, г. Москв , 2-й Н г тинский пр-д, д. 6А**.

При оформлении з к з следует ук з ть дрес (полностью), по которому должны быть высл ны книги; ф милию, имя и отчество получ теля. Жел тельно т кже ук з ть свой телефон и электронный дрес.

Эти книги вы можете з к з ть и в интернет-м г зине: **www.alians-kniga.ru**.

Оптовые з купки: тел. +7(499) 782-38-89

Электронный дрес: **books@alians-kniga.ru**.

Л ур Белл, М йкл Бр нтон-Сполл, Рич Смит, Джим Бэрд

Безопасность разработки в Agile-проектах

Обеспечение безопасности в конвейере непрерывной поставки

Главный редактор *Мовчан Д. А.*
dmkpress@gmail.com
Перевод с английского *Слинкин А. А.*
Корректор *Синяева Г. И.*
Верстка *Паранская Н. В.*
Дизайн обложки *Мовчан А. Г.*

Формат 70×100$^{1}/_{16}$. Печать цифровая.
Усл. печ. л. 36,4. Тираж 200 экз.

Веб-с йт изд тельств : *www.dmkpress.com*