МИНОБРНАУКИ РОССИИ
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ
ВЫСШЕГО ОБРАЗОВАНИЯ
«ВОРОНЕЖСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»
(ФГБОУ ВО «ВГУ»)

Факультет компьютерных наук

Кафедра технологий обработки и защиты информации

Принципы обеспечения безопасности веб-приложений

Направление 10.03.01 Информационная безопасность

Технологии обработки и защиты информации

Зав. кафедрой _____ *д.т.н., профессор, А.А. Сирота 2022*

Обучающийся _____ *А.А. Карелов, 4 курс, д/о*

Руководитель _____ *М.А.Дрюченко, к.т.н., доцент*

Воронеж 2022

Введение

Понятие веб-приложения относится к клиент-серверному приложению, в котором клиентом выступает браузер, а сервером — веб-сервер. Логика веб-приложения распределена между сервером и клиентом, хранение данных осуществляется, преимущественно, на сервере, обмен информацией происходит по сети. Одним из преимуществ такого подхода является тот факт, что клиенты не зависят от конкретной операционной системы пользователя, поэтому веб-приложения являются межплатформенными сервисами.[11]

Использование информационных систем и технологий связано с определенным набором рисков. Оценка вероятности обеспечения безопасности необходима для контроля эффективности мер по обеспечению информационной безопасности, принятия соответствующих защитных мер и построения эффективных систем защиты.

Оценка вероятности обеспечения безопасности основана на возможных уязвимостях и угрозах, поэтому выявление потенциальных или реальных рисков нарушения конфиденциальности и целостности информации, распространения вредоносного программного обеспечения и классификаций угроз информационной безопасности является одной из первоочередных задач защиты веб-приложения.

Безопасность веб-приложения нужно поддерживать постоянно, периодически проверяя веб-приложение на уязвимости. Хорошо выполненная и хорошо задокументированная первая оценка может значительно упростить последующие шаги. Несмотря на это деятельность по обеспечению информационной безопасности, должна быть интегрирована в жизненный цикл информационной системы. Тогда эффект оказывается наибольшим, а затраты — минимальными.

Источник угрозы

Источниками угроз информационной безопасности являются субъекты, действия которых можно квалифицировать как умышленные или случайные преступления. Методы противодействия напрямую зависят от организаторов защиты информации.

В качестве источника угроз так же можно рассматривать субъект, имеющий доступ (санкционированный или несанкционированный) к работе со штатными средствами веб-приложения. Это хакер-взломщик, либо их группа, либо персонал компании, мотивированный определенными факторами на совершение незаконных или противоправных действий. Субъекты (источники), действия которых могут привести к нарушению информационной безопасности, могут быть как внешними, так и внутренними.[1]

Внешние источники могут быть случайными или преднамеренными и иметь разный уровень квалификации. К ним относятся:

- криминальные структуры;
- потенциальные преступники и хакеры;
- недобросовестные партнеры;

Внутренние субъекты (источники), как правило, являются высококвалифицированными специалистами в области разработки и эксплуатации программно-технических средств, знакомы со спецификой решаемых задач, структурой и основными функциями и принципами работы программно-аппаратных средств, средств защиты информации. К ним относятся:

- основной персонал (пользователи, программисты, разработчики);
- представители службы защиты информации.

1 Процесс защиты современных веб-приложений

1.1 Архитектура защищённого ПО

Первый шаг в обеспечении безопасности любого веб-приложения делается на этапе выбора архитектуры приложения. В разработке ПО роль архитектора заключается в проектировании модулей на высоком уровне и в выборе оптимальных способов их взаимодействия друг с другом. Сюда можно добавить выбор наилучших способов хранения данных, будущих сторонних зависимостей, преобладающей парадигмы программирования и т. п. Выбор архитектуры ПО — тонкий процесс, который сопряжён с большим риском, поскольку переделка архитектуры уже готового приложения стоит очень дорого. Аналогично обстоят дела с архитектурой системы безопасности. Часто уязвимости можно легко предотвратить на этапе проектирования с помощью тщательного планирования и оценки. Недостаток планирования приводит к тому, что код приложения приходится переделывать, что не слишком дёшево. По результатам исследования популярных веб-приложений Национальный институт стандартов и технологий (National Institute of Standards and Technology, NIST) объявил, что «стоимость устранения уязвимости на этапе проектирования в 30–60 раз меньше, чем ее исправление в процессе производства».

Аутентификация и авторизация

Для реализации хранения учётных данных и разных уровней доступа для гостей и зарегистрированных пользователей нужны системы аутентификации и авторизации. Именно они разрешают пользователям входить в систему и позволяют нам определить допустимые действия для разных групп пользователей. Все учётные данные будут храниться в базе. Это означает, что для устранения риска утечки данных нужно тщательно планировать такие вещи, как:

- Способ обработки данных в процессе их передачи
- Способ хранения учётных данных
- Реализация различных уровней авторизации пользователей

Протоколы SSL и TLS

Первым делом решается, каким образом будут обрабатываться данные в процессе передачи. Это решение повлияет на поток всех данных в веб-приложении**.** Первое требование к передаче данных — шифрование. Оно снижает риск атаки посредника, которая может привести к похищению учётных данных пользователей. Два основных криптографических протокола, которые сегодня используются для защиты передаваемых по сети данных, — это SSL и TLS. Все основные современные веб-браузеры добавляют в адресную строку иконку замка, если связь должным образом защищена с помощью SSL или TLS. Спецификация HTTP предлагает расширение HTTPS в целях повышения безопасности. Данные в протоколе HTTPS передаются поверх криптографических протоколов TLS/SSL. Если при выполнении запроса HTTPS TLS/SSL соединения окажутся скомпрометированными, браузер покажет пользователю предупреждение.

Хеширование учётных данных

Конфиденциальные учётные данные недопустимо хранить в виде обычного текста. Перед первым сохранением пароль следует хешировать. Этот несложный процесс даёт огромные преимущества в плане безопасности.

Алгоритмы хеширования отличаются от большинства алгоритмов шифрования. Во-первых, они необратимы. При работе с паролями это ключевой момент. Нужно быть уверенным, что даже сотрудники компании не смогут украсть пароли пользователей и использовать их в других местах. Современные алгоритмы хеширования очень эффективны. Они могут преобра- зовывать многомегабайтные строки символов в 128–264-битные. При проверке присланного пользователем пароля мы повторно хешируем его и сравниваем с хешированным паролем в базе данных. Даже если у пользователя огромный пароль, мы сможем очень быстро выполнить поиск и сравнение по базе. Ещё одно ключевое преимущество хеширования — низкая вероятность коллизий. Правильно хешированные пароли защитят

пользователей в случае взлома базы и воровства личных данных. Хакер получит доступ только к хешам, и маловероятно, что он сможет реконструировать хотя бы один пароль. При взломе паролей алгоритмы медленного хеширования сильно усложняют работу хакера, который обязательно будет автоматизировать процесс. Как только хакер найдёт идентичный хеш к паролю, тот фактически взломан. Чрезвычайно медленные хеш-алгоритмы, такие как BCrypt, могут растягивать время такого поиска на годы. Так же можно использовать алгоритм PBKDF2. PBKDF2 — альтернатива функции BCrypt. В его основе лежит концепция, известная как растяжение ключей. Такие алгоритмы сначала быстро генерируют хеш, но каждая следующая попытка будет происходить медленнее, что превращает брутфорс в крайне дорогостоящий в вычислительном отношении процесс.

Двухфакторная аутентификация

В дополнение к хешированию паролей также можно использовать двухфакторную аутентификацию (2FA) тем пользователям, которые хотят ещё сильнее усилить защиту своей учётной записи. Большинство систем 2FA требуют в дополнение к вводимому в браузер паролю ввод сгенерированного пароля из мобильного приложения или текстового SMS-сообщения. Более продвинутые протоколы 2FA используют аппаратный токен, который обычно генерируется USB-накопителем при подключении к компьютеру пользователя. При отсутствии каких-либо уязвимостей в приложении 2FA или в протоколе обмена сообщениями двухфакторная аутентификация исключает возможность удалённого входа в веб-приложение, инициированного не владельцем учётной записи. Скомпрометировать учётную запись в этом случае можно, только получив доступ одновременно и к паролю, и к физическому устройству, на который присылается второй пароль.

1.2 Проверка безопасности кода

Данный этам проверки должен идти после проверки архитектуры ПО, такой подход позволяет получить более безопасный код. Этап проверки

безопасности кода жизненно важен как для функциональности, так и для безопасности приложения. Его следует обязательно внедрять в организациях, которые проводят только функциональные проверки. Это резко снизит количество серьёзных ошибок безопасности, которые в противном случае могли бы вылезти на этапе эксплуатации. В общем случае обзоры безопасности кода имеют наибольший смысл перед запросами на принятие изменений (так называемые merge requests, или pull requests). На этом этапе уже разработан полный набор функций и интегрированы все системы, требующие подключения. Соответственно, можно одновременно просмотреть весь объем кода.

Основные типы уязвимостей и пользовательские логические ошибки

В процессе проверки функциональности разработчики пытаются убедиться, что все соответствует спецификации, а при использовании приложения не возникают ошибки. При проверке безопасности кода убеждаются в отсутствии распространённых уязвимостей, таких как XSS, CSRF, DoS возможностей внедрения кода и т. п., но важнее всего провести проверку на логическом уровне, которая требует глубокого понимания контекста кода. Такие уязвимости нельзя легко обнаружить автоматическими инструментами или сканерами. Для поиска уязвимостей, связанных с логическими ошибками, первым делом нужно понять, для чего предназначена просматриваемая функциональность. Разработчик должен чётко осознавать, что делают пользователи и как реализована функциональность.

С чего следует начать проверку безопасности

Обзор кода следует начинать с компонентов приложения, подвергающихся наибольшему риску. Лучше всего начинать работу с исходным кодом с любого места, где клиент (браузер) делает запрос к серверу. Это позволит получить представление о том, с чем программист имеем дело. Можно узнать, какими данными обмениваются клиент и сервер, а также сколько серверов используются приложением. Кроме того, можно понять, как присылаемые

данные интерпретируются на сервере. После оценки клиента нужно проследить за обратными вызовами его API к серверу. После вызова API к серверу, рассматривается возможность отслеживания вспомогательных методов, зависимостей и функционала, на которые полагаются эти API. То есть оценка баз данных, журналов регистрации, загруженных файлов, библиотек преобразования и всего остального, что конечные точки API вызы- вают напрямую, или через вспомогательную библиотеку. Затем анализируется каждая часть функциональности, которая может предъявляться клиенту, но не вызывается напрямую. Это могут быть API-интерфейсы, созданные для поддержки будущих функций, или, возможно, внутренние функции, которые внезапно стали видимыми снаружи. После этого можно проанализировать остальную часть кодовой базы. Последовательность определяется путем анализа бизнес-логики и расстановки приоритетов на основе предполагаемых рисков.

Антипаттерны безопасного программирования

Проверки безопасности на уровне кода имеют некоторые общие черты с планированием архитектуры приложения, которое происходит перед написанием кода. Но если на этапе планирования архитектуры все уязвимости являются гипотетическими, в процессе проверки кода их действительно можно обнаружить. Есть несколько антипаттернов, на которые следует обращать внимание при любых проверках безопасности, такие как:

- Черные списки
- Шаблонный код
- Разделение клиента и сервера

Черные списки

Черные списки — это пример временного или неполного решения. В сфере безопасности веб-приложений лучше не использовать временные решения, а сразу искать постоянные, даже если это займет больше времени. Единственный случай, когда допустимо временное или неполное решение, —

это заранее запланированный график, на основе которого будет спроектировано и реализовано полное решение.

Шаблонный код

Так же следует обратить внимание на применение шаблонного кода, который фреймворк генерирует по умолчанию. Дело в том, что фреймворки и библиотеки зачастую требуют дополнительных усилий для повышения безопасности, тогда как по идее она должна быть по умолчанию. Классическим примером является ошибка конфигурации в базе данных MongoDB, из-за которой установленные на веб-сервере более старые версии базы по умолчанию оказывались доступными через интернет. При этом там не требовалась обязательная аутентификация, в результате чего с помощью специальных сценариев были захвачены десятки тысяч баз MongoDB.

Разделение клиента и сервера

Следует обращать внимание на слишком тесную связь клиента и сервера. Этот антипаттерн возникает, когда код клиентской и серверной частей приложения настолько переплетается, что одно не может работать без другого. В основном такое встречается в старых веб-приложениях. В безопасном приложении клиентская и серверная части должны быть разработаны независимо друг от друга. Они взаимодействуют по сети с использованием заранее определенного формата данных и сетевого протокола.

1.3 Обнаружение уязвимостей

После разработки, написания и проверки кода, дальнейшим шагом будет проверка на незамеченные уязвимости. Как правило, чем лучше спроектирована архитектура приложения, тем меньше в нем уязвимостей и тем меньший риск они несут. Но даже надежная архитектура и достаточное количество проверок не гарантируют полного отсутствия уязвимостей. Иногда они просто остаются незамеченными, а иногда возникают как результат неожиданного поведения: например, запуска приложения в другой

среде, следовательно требуются процедуры обнаружения уязвимостей в готовом коде.

Автоматизированнния проверка

После анализа кода его следует первым делом подвергнуть автоматизированной проверке, данная проверка является дешевой, эффективный, но долговременный способ. Методы автоматического обнаружения великолепно подходят для поиска непримечательных недостатков кода, которые часто ускользают от внимания проектировщиков и рецензентов. Для поиска уязвимостей, связанных с логической структурой приложения, они не подходят. Нельзя с их помощью выявлять и уязвимости, которые эффективны в цепочке, а именно несколько слабых уязвимостей, которые вместе дают сильную. Наиболее распространенные виды автоматизированной пароверки безопасности:

- Статический анализ
- Динамический анализ
- Регрессионное тестирование

Каждая из этих форм автоматизации имеет свою цель и занимает важное место в жизненном цикле разработки приложения, поскольку обнаруживает свои типы уязвимостей.

Статический анализ

Для начала работы со статическим анализом прежде всего нужно написать тесты для статического анализа. Статический анализ может выполняться локально во время разработки и по запросу в репозитории исходного кода или при каждом коммите. Плюсы статического анализа:

- Выявляет дефекты до начала code review
- Анализатор может работать круглые сутки
- Можно найти ошибки, даже не зная о таком паттерне
- Можно найти ошибки, которые при обзоре крайне сложно заметить

Минусы статического анализа:

- Нельзя выявить высокоуровненвые ошибки

- Ложные срабатывания

Инструменты статического анализа должны быть настроены на поиск десяти наиболее распространенных уязвимостей по версии OWASP.

Статический анализ позволяет обнаружить следующие угрозы:

- Ищутся манипуляции с DOM через innerHTML (Межсайтовый скриптинг)
- Ищутся переменные, извлеченные из URL-адреса (Отраженный XSS)
- Ищутся приемники DOM, такие как setInterval (DOM XSS)
- Ищутся предоставляемые пользователем строки, используемые в запросах (Внедрение SQL-кода)
- Ищутся запросы GET, меняющие состояния (CSRF)
- Ищутся неправильные регулярные выражения (DoS)

Статический анализ — мощное средство обнаружения уязвимостей общего плана, но, к сожалению, он дает много ложных срабатываний.

Кроме того, статический анализ хуже работает с динамическими языками (например, с Java Script). Языки со статической типизацией, такие как Java или C #, подходят для него намного лучше, поскольку инструментальные средства понимают ожидаемый тип данных, который не меняется при прохождении через функции и классы.

Динамический анализ

В отличие от статического анализа, при котором рассматривается только сам код, динамический анализ происходит в процессе выполнения кода. Соответственно, проводить его гораздо дороже и дольше.

Динамический анализ отлично выявляет реальные уязвимости, тогда как статический зачастую обнаруживает потенциальные уязвимости, но имеет ограниченные способы их подтверждения. В процессе динамического анализа выполняется код, а затем происходит срав- нение выходных данных с моделью, описывающей уязвимости и неправильные конфигурации. Так что это отличный вариант для тестирования динамических языков, ведь он позволяет посмотреть на результат работы кода, а не только на входные данные и поток их обработки. Подходит динамический анализ и для поиска

уязвимостей, возникающих как побочный эффект работы приложения, таких как неправильное сохранение в памяти конфиденциальных данных или атаки по побочным каналам.

Плюсы динамического анализа:

- В большинстве реализаций появление ложных срабатываний исключено, так как обнаружение ошибки происходит в момент ее возникновения в программе, таким образом, обнаруженная ошибка является не предсказанием, сделанным на основе анализа модели программы, а констатацией факта ее возникновения;
- Зачастую не требуется исходный код, это позволяет протестировать программы с закрытым кодом.

Минусы динамического анализа:

- Динамический анализ обнаруживает дефекты только на трассе, определяемой конкретными входными данными, дефекты, находящиеся в других частях программы, не будут обнаружены;
- Не может проверить правильность работы кода, что код делает то, что должен;
- Требуются значительные вычислительные ресурсы для проведения тестирования;
- При тестировании на реальном процессоре исполнение некорректного кода может привести к непредсказуемым последствиям.

1.4 Управление уязвимостями

В крупных приложениях уязвимости будут обнаруживаться на всех этапах — от проектирования архитектуры до окончательной версии кода. Уязвимости, обнаруженные на этапе проектирования, исправить проще всего, а контрмеры можно разрабатывать до начала написания кода. А вот все уязвимости, которые были выявлены после этого этапа, необходимо должным образом контролировать, чтобы в конечном итоге устранить.

Воспроизведение уязвимостей

Для эффективного воспроизведения уязвимостей необходимо создать промежуточную среду, которая максимально точно имитирует реальную работу с приложением. Настройка такой среды может оказаться сложной, поэтому процесс лучше полностью автоматизировать. Все готовые к выпуску новые функции должны появиться в сборке, доступной только во внутренней сети или защищенной зашифрованным входом в систему. Данная тестовая среда реальных пользователей не трубует, но чтобы визуально и логически представить функционирование приложения в рабочем режиме, потребуются mock-объекты и mock-пользователи. Такой процесс воспроизведения уязвимостей дает более глубокое представление о том, что могло стать ее причиной. Это важный первый шаг к ее устранению. После воспроизведения уязвимости нужно понять, каким образом происходит ее эксплуатация. Нужно установить механизм доставки вредоносного кода и определить, что в результате попадает под удар (данные, ресурсы и т. п.). На основе этой информации происходит классификация уязвимостей по степени их серьезности. Для классификации важна четко определенная и отслеживаемая система оценки: с одной стороны, достаточно надежная, чтобы давать точное сравнение, а с другой — достаточно гибкая, чтобы ее можно было применять и к необычным формам уязвимостей. Чаще всего для этой цели используют общую систему оценки уязвимостей.

Общая система оценки уязвимостей

Общая система оценки уязвимостей (Common Vulnerability Scoring System, CVSS) представляет собой бесплатный и открытый отраслевой стандарт для оценки серьезности уязвимостей системы безопасности компьютера. Оценка зависит от того, насколько легко эксплуатируется уязвимость и какие типы данных или процессов оказываются скомпрометированными. Данная система применяет три типа метрик:

- базовые метрики описывают характеристики уязвимости, не меняющие- ся с течением времени и не зависящие от среды исполнения;

- временные метрики оценивают, как серьезность уязвимости меняется с течением времени;
- контекстные метрики оценивают уязвимость с учетом характеристик информационной среды.

CVSS:Базовая метрика

Алгоритм оценки в случае базовой метрики определяется восьмью параметрами:

1) Вектор атаки. Вектор атаки может принимать значения «сетевой» (Network), «соседняя сеть» (Adjacent), «локальный» (Local) и «физический» (Physical) по степени убывания угрозы. Эти значения описывают степень удаленности атакующего от эксплуатируемого объекта. Самым тяжелым является доступ по сети. Значение «физический» указывает, что атакующему требуется физический доступ к уязвимой подсистеме. Так как получить его крайне сложно, этот вариант считается несущим наименьший риск.

2) Сложность доступа. Этот параметр принимает два значения — «низкая» (low) или «высокая» (high) — и характеризует сложность эксплуатации системы, которую можно определить, например, через количество шагов (предварительный сбор информации, настройка), необходимых перед доставкой вредоносного кода, а также через количество переменных, находящихся вне контроля хакера. Например, атака, которую можно осуществлять снова и снова без настройки, имеет «низкую» сложность доступа, в то время как атака, требующая входа в систему определенного пользователя в определенное время и на опреде- ленной странице, будет характеризоваться «высокой» сложностью.

3) Требуемый уровень привилегий. Этот параметр характеризует уровень авторизации, необходимый для проведения атаки. Возможны три значения: «нет» (none, анонимный пользователь), «низкий» (low) и «высокий» (high). Атака с «высоким» уровнем привилегий может осуществляться только администратором системы, в то время как «низкий» уровень относится к обычным пользователям.

4) Взаимодействие с пользователем. Этот параметр допускает всего два значения: «не требуется» (none) и «требуется» (required), и указывает, нужно ли взаимодействие с пользователем (например, его переход по ссылке) для успешной атаки.
5) Сфера воздействия. Этот параметр показывает, затрагивает ли атака только уязвимый компонент или же распространяется и на другие. Значение «неизменяемая» (unchanged) указывает на атаку, затронувшую только компонент, через который она была осуществлена. Значение «изменяемая» (changed) относится к атакам, распространяющимся за пределы уязвимой функциональности. Это может быть, например, атака на базу данных, влияющая на операционную или файловую систему.
6) Влияние на конфиденциальность. Для этого параметра возможны три значения: «отсутствует» (none), «низкая» (low) и «высокая» (high). Каждая эксплуатация уязвимости может сопровождаться компрометацией данных, не предназначенных для неавторизованных пользователей. Серьезность компрометации зависит от степени воздействия на организацию, которая, в свою очередь, определяется бизнес-моделью приложения. Ведь некоторые предприятия (например, в сфере здравоохранения) хранят гораздо больше конфиденциальных данных, чем остальные.
7) Влияние на целостность. Этот параметр также может принимать три значения: «отсутствует» (none), «низкая» (low) и «высокая» (high). Первый вариант относится к атаке, не влияющей на состояние приложения. При втором меняется какое-то состояние приложения в ограниченном объеме, а «высокое» влияние означает изменение всех или большинства состояний приложения. Состояние приложения обычно используется при обращении к данным, хранящимся на сервере, но может применяться и в отношении хранилищ на стороне клиента (локальное хранилище, хранилище сеансов, indexedDB).
8) Влияние на доступность. Этот параметр может принимать одно из трех значений: «отсутствует» (none), «низкая» (low) и «высокая» (high). Он

характеризует доступность информационных ресурсов для обычных пользователей. На доступность системы влияют атаки, потребляющие пропускную способность сети (DoS), циклы процессора или атаки на выполнение кода, которые перехватывают предполагаемую функциональность.

CVSS:Временная метрика

Алгоритм оценки в случае временной метрики определяется тремя параметрами:

1) Зрелость доступных средств эксплуатации (Exploitability, E). Принимает значения «непроверенная» (unproven) и «высокая» (high). Эта метрика пытается определить, является ли заявленная уязвимость теоретической (в этом случае код или технология не действуют в большинстве ситуаций или требуют существенной доработки) или может быть развернута и использована как есть (рабочая уязвимость).

2) Уровень исправления (Remediation Level, RL). Уровень исправления принимает значение, предлагающее уровень доступных смягчений. Для известной уязвимости с рабочим и протестированным исправлением статус будет «O» — «официальное исправление» (official fix), в то время как для уязвимости, решение которой неизвестно, будет стоять «U» — «Исправление недоступно» (fix unavailable).

3) Степень достоверности отчета (Report Confidence, RC). Этот параметр характеризует качество отчета об уязвимостях. Теоретический отчет без кода воспроизведения или понимания того, как начать процесс воспроизведения, получит значение «неподтвержденная» (unknown) достоверность. Если же существует хорошо написанный отчет и уязвимость легко воспроизводится, параметр получает значение «подтвержденная» (confirmed).

CVSS:Контекстная метраика

Алгоритм оценки среды принимает все параметры базовой оценки и дополни- тельно к ним еще три, уточняющие требования к

конфиденциальности, целост- ности и доступности приложения. Алгоритм оценки в случае контекстной метрики определяется тремя параметрами:

1) Требования к конфиденциальности (Confidentiality Requirement, CR). Уровень конфиденциальности, которого требует приложение. Низкую оценку получают общедоступные приложения, в то время как приложения со строгими контрактными требованиями (здравоохранение, правительство) будут оценены выше.

2) Требования к целостности (Integrity Requirement, IR). Последствия для приложения, состояние которого поменялось в результате эксплуатации уязвимости. Приложение, создающее тестовые «песочницы», получит более низкую оценку, чем, к примеру, приложение, в котором хранятся налоговые записи предприятия.

3) Требования к доступности (Availability Requirement, AR). Последствия для приложения в результате простоя. Приложение, от которого ожидается круглосуточная работа, будет затронуто больше, чем приложение, не обещающее безотказного доступа.

Вывод

При создании приложения следует учитывать множество факторов. Когда организация разрабатывает новое приложение, опытный инженер или проектировщик должен тщательно проанализировать его будущую структуру, дизайн и архитектуру. Глубокие недостатки безопасности, такие как неправильная схема аутентификации или неполноценная интеграция с поисковой системой, могут подвергнуть приложение риску, который нелегко устранить. Как только коммерческие клиенты начнут использовать приложение в своих рабочих процессах, устранение ошибок безопасности на уровне архитектуры превратится в чрезвычайно трудную задачу, что в своб очередь привдет в большим финансовым потерям.