**Тема 27 Тестирование защиты программного обеспечения**

1 Область применения

2 Приемы выявления уязвимостей

3 Ручной (экспертный анализ)

4 Статический анализ безопасности (по шаблону)

5 Динамический анализ безопасности

На сегодняшний день много сказано и написано о тестировании программного обеспечения. Как правило**, под тестированием программного обеспечения** подразумевается тестирование Web-приложений — порядка 70% , на оставшиеся 30% приходиться  тестирование баз данных и все остальные виды тестирования. Таким образом, понятно, что информацию о тестировании программного обеспечения *по требованиям безопасности* в литературе найти очень сложно. Но даже если удается, что-то найти, то  представленная информация очень поверхностная. Все что удалось найти о тестировании безопасности — это в лучшем случае сканирование приложений сканерами безопасности и антивирусные проверки, т.е. тестирование не самого программного продукта, а скорее его окружения, где данный продукт может применяться. Но не стоит забывать, что наибольшую опасность представляют уязвимости, заложенные (чаще допущенные из-за невнимательности разработчиков) в приложении на самых ранних этапах его жизненного цикла на этапе разработки, а именно уязвимые сигнатуры в исходных текстах тестируемого программного обеспечения (причем не обязательно Web- приложений).  И действительно на практике большая часть программного обеспечения на рынке информационных  технологий проходит лишь этапы внутреннего тестирования (альфа-тестирование), основной целью которого является выявление ошибок в разработанном ПО. При этом о тестировании безопасности программного обеспечения чаще всего не задумываются. Основная проблема заключается в том, что пока нет никакого российского стандарта или руководящего документа, в котором описывался бы процесс тестирования по требованиям безопасности.

Исходя из вышеизложенного, появилась необходимость разработать стандарт, в котором будут отражены требования и процедуры тестирования программных изделий по требованиям безопасности!

В рамках этой лекции мы рассмотрим, собственно говоря, что мы будем тестировать (объекты тестирования и область их применения) и на соответствие каких именно требований безопасности мы будем проводить тесты.

**2. ОБЛАСТЬ ПРИМЕНЕНИЯ**

Для начала определим, что такое тестирование программного обеспечения в общем понимании. ***Тестирование*** – процесс проверки соответствия заявленных к продукту требований и реально реализованной функциональности, осуществляемый путем наблюдения за его работой в искусственно созданных ситуациях и на ограниченном наборе тестов, выбранных определенным образом. Тогда ***тестирование по требованиям безопасности*** есть не что иное, как процесс выявления наличия или отсутствия уязвимостей в продукте в искусственно созданных ситуациях и на ограниченном наборе тестов, выбранных определенным образом.

Значит, тестировать мы будем программное изделие на возможность наличия в нем опасных сигнатур (на уровне кода) заложенных, либо оставленных невнимательными (некомпетентными) разработчиками. Так же нужно понимать, что если при тестировании нам удалось обнаружить потенциально опасные участки кода, это еще не значит, что мы нашли уязвимость (поэтому такие сигнатуры и называются ПОТЕНЦИАЛЬНО опасными), нам будет необходимо провести «экспериментальное тестирование», т.е. попробовать использовать опасные участки кода с целью компрометации нашего приложения. На выходе мы должны добиться нарушения доступности, целостности, конфиденциальности, чтобы удостовериться в наличии уязвимости. И только в этом случае мы можем говорить, что обнаруженные сигнатуры влияют, либо не влияют на безопасность тестируемого программного обеспечения.

Таким образом, при проведении тестирования по требованиям безопасности мы проводим следующие испытания:

—        аудит безопасности кода (направленный на выявление уязвимостей);

—        функциональное тестирование (характеристик безопасности тестируемого программного обеспечения);

—        экспериментальное тестирование (проверяется возможность, или невозможность эксплуатации обнаруженных сигнатур).

**3. ПРИЕМЫ ВЫЯВЛЕНИЯ УЯЗВИМОСТЕЙ**

3.1. Ручной (экспертный анализ)

3.2. Статически анализ безопасности (по шаблону)

3.3. Динамический анализ безопасности

При ручном подходе выявления уязвимостей применяется экспертный анализ, т.е. специалист, который проводит данное исследование, полагается на свои знания и опыт. Данный подход не подразумевает использования, каких-либо автоматизированных средств. Понятно, что данный прием имеет большие затраты по времени и предполагает наличие специалистов высокой квалификации. Данный подход считается самым эффективным с точки зрения точности и полноты покрытия проверок.

Прием выявления уязвимостей «по шаблону» подразумевает использование материалов и наработок полученных исходя из опыта работы в данной области. При подходе выявления уязвимостей «по шаблону» часто применяется автоматизированный подход поиска уязвимостей по заданным шаблонам (спискам потенциально опасных сигнатур). Часто при данном подходе используется совмещение методов автоматизированного и ручного поиска уязвимостей. Используются средства автоматизированного поиска уязвимостей кода PREfix, FlawFinder, RATS, UCA, ITS4, АК-ВС и другие. Подробнее о средствах автоматизации поиска уязвимостей поговорим в разделе 5 («Обзор средств выявления уязвимостей»). Кроме того всплывает проблема связанная с тем, где именно взять эти списки (базу) потенциально опасных сигнатур. На данный момент сложилась такая практика, что каждая испытательная лаборатория или даже отдельный департамент тестирования используют свои собственные базы сигнатур. Естественно это приводит к частым разногласиям. Поэтому возникает необходимость создания стандарта, в котором будут приведены списки потенциально опасных сигнатур. Можно воспользоваться положительной практикой уже существующей в международных проектах CWE, либо известным ресурсом для Web-приложений OWASP .

Динамический анализ является обязательным подходом при выявлении уязвимостей. Он позволяет проводить тестирование при непосредственном выполнении программного изделия. В процессе динамического тестирования программного комплекса реализуется составленный список тестов, направленный на достижение, либо провал нарушения функций безопасности продукта. Т.е. определяется возможность или невозможность эксплуатировать найденную потенциально опасную сигнатуру в рамках работающего программного продукта, при заданном тестовом окружении.

**4. ТРАДИЦИОННЫЕ МЕТОДЫ И ПОДХОДЫ ТЕСТИРОВАНИЯ**

В данном разделе будут рассмотрены традиционные методы, виды и подходы тестирования. Многие из представленных методов применяются при тестировании по требованиям безопасности. Главным отличием при тестировании по требованиям безопасности является то, что мы ищем не ошибки (bugs), а уязвимости, ведущие к нарушению целостности, конфиденциальности и доступности.

4.1. Уровни тестирования:

4.1.1. Модульное тестирование (Unit testing)

Этот уровень тестирования позволяет проверить функционирование отдельно взятого элемента системы. Что считать элементом – модулем системы определяется контекстом. Наиболее полно данный вид тестов описан в стандарте IEEE 1008–87 “Standard for Software Unit Testing”, задающем интегрированную концепцию систематического и документированного подхода к модульному тестированию.

4.1.2. Интеграционное тестирование (Integration testing)

Данный уровень тестирования является процессом проверки взаимодействия между программными компонентами/модулями. Классические стратегии интеграционного тестирования – “сверху-вниз” и “снизу-вверх” – используются для традиционных, иерархически структурированных систем и их сложно применять, например, к тестированию слабосвязанных систем, построенных в сервисно-ориентированных архитектурах (SOA).

4.1.3. Системное тестирование (System testing)

Системное тестирование охватывает целиком всю систему. Большинство функциональных сбоев должно быть идентифицировано еще на уровне модульных и интеграционных тестов. В свою очередь, системное тестирование, обычно фокусируется на нефункциональных требованиях – безопасности, производительности, точности, надежности т.п. На этом уровне также тестируются интерфейсы к внешним приложениям, аппаратному обеспечению, операционной среде и т.д.

4.2. Виды тестирования:

4.2.1. Приёмочное тестирование (Acceptance/qualification testing)

Проверяет поведение системы на предмет удовлетворения требований заказчика.

4.2.2. Установочное тестирование (Installation testing)

Из названия следует, что данные тесты проводятся с целью проверки процедуры инсталляции системы в целевом окружении.

4.2.3. Альфа- и бета-тестирование (Alpha and beta testing)

Перед тем, как выпускается программное обеспечение, как минимум, оно должно проходить стадии альфа (внутреннее пробное использование) и бета (пробное использование с привлечением отобранных внешних пользователей) версий. Отчеты об ошибках, поступающие от пользователей этих версий продукта, обрабатываются в соответствии с определенными процедурами, включающими подтверждающие тесты (любого уровня), проводимые специалистами группы разработки.

4.2.4. Функциональные тесты/тесты соответствия (Conformance testing/Functional testing/Correctness testing)

Эти тесты могут называться по разному, однако, их суть проста – проверка соответствия системы, предъявляемым к ней требованиям, описанным на уровне спецификации поведенческих характеристик.

4.2.5. Достижение и оценка надежности (Reliability achievement and evaluation)

Помогая идентифицировать причины сбоев, тестирование подразумевает и повышение надежности программных систем. Случайно генерируемые сценарии тестирования могут применяться для статистической оценки надежности.

4.2.6. Регрессионное тестирование (Regression testing)

Определение успешности регрессионных тестов (IEEE 610–90 “Standard Glossary of Software Engineering Terminology”) гласит: “повторное выборочное тестирование системы или компонент для проверки сделанных модификаций не должно приводить к непредусмотренным эффектам”. На практике это означает, что если система успешно проходила тесты до внесения модификаций, она должна их проходит и после внесения таковых.

4.2.7. Тестирование производительности (Performance testing)

Специализированные тесты проверки удовлетворения специфических требований, предъявляемых к параметрам производительности. Существует особый подвид таких тестов, когда делается попытка достижения количественных пределов, обусловленных характеристиками самой системы и ее операционного окружения.

4.2.8. Нагрузочное тестирование (Stress testing)

Необходимо понимать отличия между рассмотренным выше тестированием производительности с целью достижения ее реальных (достижимых) возможностей производительности и выполнением программной системы c повышением нагрузки, вплоть до достижения запланированных характеристик и далее, с отслеживанием поведения на всем протяжении повышения загрузки системы.

4.2.9. Сравнительное тестирование (Back-to-back testing)

Единичный набор тестов, позволяющих сравнить две версии системы.

4.2.10. Восстановительные тесты (Recovery testing)

Цель – проверка возможностей рестарта системы в случае непредусмотренной катастрофы (disaster), влияющей на функционирование операционной среды, в которой выполняется система.

4.2.11. Конфигурационное тестирование (Configuration testing)

В случаях, если программное обеспечение создается для использования различными пользователями (в терминах “ролей”), данный вид тестирования направлен на проверку поведения и работоспособности системы в различных конфигурациях.

4.2.12. Тестирование удобства и простоты использования (Usability testing)

Цель – проверить, насколько легко конечный пользователь системы может ее освоить, включая не только функциональную составляющую – саму систему, но и ее документацию; насколько эффективно пользователь может выполнять задачи, автоматизация которых осуществляется с использованием данной системы; наконец, насколько хорошо система застрахована (с точки зрения потенциальных сбоев) от ошибок пользователя.

4.3. Техники, ориентированные на код (Code-based techniques):

4.3.1. Тесты, базирующиеся на блок-схеме (Control-flow-based criteria)

Набор тестов строится исходя из покрытия всех условий и решений блок-схемы. В какой-то степени напоминает тесты на основе конечного автомата. Отличие – в источнике набора тестов. Максимальная отдача от тестов на основе блок-схемы получается когда тесты покрывают различные пути блок-схемы – по-сути, сценарии потоков работ (поведения) тестируемой системы. Адекватность таких тестов оценивается как процент покрытия всех возможных путей блок-схемы.

4.3.2. Тесты на основе потоков данных (Data-flow-based criteria)

В данных тестах отслеживается полный жизненный цикл величин (переменных) – с момента рождения (определения), на всем протяжении использования, вплоть до уничтожения (неопределенности). В реальной практике используются нестрогое тестирование такого вида, ориентированное, например, только на проверку задания начальных значений всех переменных или всех вхождений переменных в код, с точки зрения их использования.

**5. ОБЗОР СРЕДСТВ ВЫЯВЛЕНИЯ УЯЗВИМОСТЕЙ**

Существует ряд средств автоматизации выявления уязвимостей, которые могут быть использованы при статическом анализе безопасности на уровне кода «по шаблону», основные из них представлены в таблице 1. Хотелось бы отметить, что на фоне иностранных средств, в таблице представлены и российские разработки, реализованные на высоком техническом и профессиональном уровне.

Таблица 1. Обзор средств выявления уязвимостей, работающих на уровне кода.

|  |  |  |  |
| --- | --- | --- | --- |
| **Наименование средства** | **Назначение** | **Поддерживаемые языки программирования** | **Примечание** |
| **Иностранные средства выявления уязвимостей** | | | |
| **BOON** | на основе глубокого семантического анализа автоматизирует процесс сканирования исходных текстов на Си в поисках уязвимых мест | С | выявляет возможные дефекты, предполагая, что некоторые значения являются частью неявного типа с конкретным размером буфера |
| **CQual** | расширяет язык Си дополнительными определяемыми пользователем спецификаторами типа, позволяя обнаруживать ошибки в программах | С | может использоваться, чтобы обнаружить потенциальную уязвимость форматной строки |
| **MOPS (Mоdel checking Programs for Security)** | динамическая корректировка, обеспечивающая соответствие программы на Си статической модели | С | использует модель аудита программного обеспечения, которая призвана помочь выяснить, соответствует ли программа набору правил, определенному для создания безопасных программ |
| **ITS4** | статически просматривает исходный код для обнаружения потенциальных уязвимостей защиты | С/С++ | отмечает вызовы потенциально опасных функций, таких, как strcpy/memcpy, и выполняет поверхностный семантический анализ, пытаясь оценить, насколько опасен такой код, а так же дает советы по его улучшению |
| **RATS(Rough Auditing Tool for Security)** | просматривает исходный текст, находя потенциально опасные обращения к функциям | С/С++, PHP, Perl, Python | использует сочетание проверок надежности защиты от семантических проверок в ITS4 до глубокого семантического анализа в поисках дефектов, способных привести к переполнению буфера, полученных из MOPS |
| **PScan** | статически просматривает исходный код для обнаружения потенциальных уязвимостей защиты | C | cканирует исходные тексты на Си в поисках потенциально некорректного использования функций, аналогичных printf, и выявляет уязвимые места в строках формата |
| **Flawfinder** | просматривает исходный текст, находя потенциально опасные обращения к функциям | С/С++ | выполняет поиск функций, которые чаще всего используются некорректно, присваивает им коэффициенты риска (опираясь на такую информацию, как передаваемые параметры) и составляет список потенциально уязвимых мест, упорядочивая их по степени риска |
| **Bunch** | средство анализа и визуализации программ на Си | С | строит граф зависимостей, помогающий аудитору разобраться в модульной структуре программы |
| **UNO** | нахождение таких ошибок, как неинициализированные переменные, нулевые указатели и выход за пределы массива | С | позволяет выполнять несложный анализ потока управления и потоков данных, осуществлять как внутри- так и межпроцедурный анализ, специфицировать свойства пользователя |
| **FlexeLint (PC-Lint)** | производит семантический анализ исходного кода, анализ потоков данных и управления | С/С++ | в конце работы выдаются сообщения нескольких основных типов:- возможен нулевой указатель- проблемы с выделением памяти (например, нет free() после malloc())- проблемный поток управления (например, недостижимый код)  — возможно переполнение буфера, арифметическое переполнение  — предупреждения о плохом и потенциально опасном стиле кода |
| **Parasoft C++ Test** | формирование тестов анализа уязвимостей на уровне метода, класса, файла и проекта | С++ | генерирует тестовый код, вызывая для его подготовки компилятор Visual C++ |
| **Coverity** | используется для выявления и исправления дефектов безопасности и качества в приложениях критического назначения | С/С++, Java | способен с минимальной положительной погрешностью обрабатывать десятки миллионов строк кода, обеспечивая 100-процентное покрытие трассы |
| **KlocWork K7** | предназначен для автоматизированного статического анализа кода, выявления и предотвращения дефектов программного обеспечения и проблем безопасности | С/С++, Java | выявляет коренные причины недостатков качества и безопасности программного обеспечения |
| **Frama-C** | анализ исходного кода на языке Си | С | включает ACSL (ANSI/ISO C Specification Language) — специальный язык, позволяющий подробно описывать спецификации функций Си, например, указать диапазон допустимых входных значений функции и диапазон нормальных выходных значений |
| **CodeSurfer** | может применяться для поиска ошибок в исходном коде, для улучшения понимания исходного кода | С/С++ | позволяет проводить анализ указателей, использовать и определять переменные, зависимости данных, строить графы вызовов |
|  |  |  |  |
| **FxCop** | способен обнаружить более 200 недочетов (или ошибок) в следующих областях:- архитектура библиотеки- правила именования- производительность  — безопасность | С/С++ | откомпилированный код проверяется с помощью механизмов рефлексии, парсинга MSIL и анализа графа вызовов |
| **JavaChecker** | позволяет выявлять следующие дефекты кода:- небрежная обработка исключений (пустые catch-блоки)- сокрытие имен- нарушения стиля  — нарушения стандартных контрактов  — использования (например, когда переопределен метод equals, но не hashCode);  — нарушения синхронизации | Java | по сути, является статическим анализатором Java программ, основанным на технологии TermWare |
| **Simian** | быстрый поиск дублирующихся фрагментов кода во многих файлах одновременно | C#, T-SQL, JavaScript и Visual Basic R | помогает сохранить целостность и эффективность продукта |
| **RPVS(Remote PHP Vulnerability Scanner)** | полное собрание линков с сайта, после чего подставляют спецсимволы в переменные и проводится анализ на показанных PHP ошибках и сигнатурах | РНР | позволяет находить такие уязвимости: XSS, SQL-Injection, уязвимости подключения файлов функциями include() и fopen() и раскрытия инсталляционного пути |
| **Qaudit** | быстрый анализ исходных файлов на наличие переполнения буфера, ошибок форматной строки, запросов исполняемых вызовов, переменных среды, и функций, имеющих проблемы защиты | С/С++ | написать на интерпретируемом языке Perl, прост в использовании |
| **Российские средства выявления уязвимостей** | | | |
| **АК-ВС** | автоматизированный анализ исходных текстов, с целью выявления потенциально опасных сигнатур | С/С++, Java, Pascal, C#, PHP, Assembler | позволяет проводить статический анализ исходных текстов, динамический анализ, имеет базы сигнатур для каждого из поддерживаемых языков программирования |
| **АИСТ-С** | автоматизированный анализ исходных текстов | С/С++ | позволяет проводить статический анализ исходных текстов |
| **КСАИТ** | автоматизированный анализ исходных текстов | С/С++ | позволяет проводить статический анализ исходных текстов |
| **UCA** | предназначено для выявления потенциально опасных сигнатур | С/С++, Pascal, Perl, PLM | имеет базы сигнатур для каждого из поддерживаемых языков программирования |
| **Viva64** | помогает отслеживать в исходном коде потенциально опасные фрагменты, связанные с переходом от 32-битных систем к 64-битным | С/С++ | помогает писать корректный и оптимизированный код для 64-битных систем |

**6. ВЫВОДЫ**

В данной лекции мы определили, что такое тестирование по требованиям безопасности. Выяснили область определения и основные подходы при проведении тестирования безопасности программного продукта. Рассмотрели основные средства автоматизации выявления уязвимостей. Хотя нужно понимать, что любые средства автоматизации являются лишь вспомогательными инструментами для эксперта, проводящего тестирование. Главные выводы о наличии уязвимостей в программном продукте делает эксперт, проводящий тестирование по требованиям безопасности. Так же встает важный вопрос о необходимости создания стандарта тестирования по требованиям безопасности, в котором будут приведены списки (базы) потенциально опасных сигнатур.