Т.А. Раченко

тольяттинский
государственный
университет

ОБЕСПЕЧЕНИЕ БЕЗОПАСНОСТИ ПРИ РАЗРАБОТКЕ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

Учебно-методическое пособие

Тольятти
Издательство ТГУ
2026

Министерство науки и высшего образования
Российской Федерации
Тольяттинский государственный университет

Т.А. Раченко

ОБЕСПЕЧЕНИЕ БЕЗОПАСНОСТИ ПРИ РАЗРАБОТКЕ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

Учебно-методическое пособие

Тольятти
Издательство ТГУ
2026

УДК 004.056.5(075.8)
ББК 16.84я73
Р278

Рецензенты:
канд. техн. наук, доцент высшей школы передовых производственных технологий Поволжского государственного университета сервиса *Т.С. Яницкая*;
канд. техн. наук, доцент института цифровых технологий Тольяттинского государственного университета *Н.В. Хрипунов*.

Р278 Раченко, Т.А. Обеспечение безопасности при разработке программного обеспечения : учебно-методическое пособие / Т.А. Раченко. – Тольятти : Издательство ТГУ, 2026. – 119 с. – ISBN 978-5-8259-1779-5.

Пособие разработано в соответствии с требованиями федерального государственного образовательного стандарта высшего образования – бакалавриата по направлению подготовки 09.03.03 «Прикладная информатика». В нём рассмотрены ключевые аспекты безопасности на всех этапах жизненного цикла разработки программного обеспечения (SDLC), включая проектирование, кодирование, тестирование и эксплуатацию.

Предназначено для студентов очной и заочной форм обучения, в том числе с использованием дистанционных образовательных технологий.

УДК 004.056.5(075.8)
ББК 16.84я73

Рекомендовано к изданию научно-методическим советом Тольяттинского государственного университета.

ISBN 978-5-8259-1779-5

ВВЕДЕНИЕ

Дисциплина «Обеспечение безопасности при разработке программного обеспечения» входит в блок 1 программы бакалавриата, реализуемой в соответствии с федеральным государственным образовательным стандартом высшего образования по направлению подготовки 09.03.03 «Прикладная информатика».

Целью освоения дисциплины является формирование у обучающихся знаний и навыков в области обеспечения безопасности на всех этапах разработки программного обеспечения (ПО), включая:

– понимание ключевых угроз и уязвимостей в ПО;
– освоение методов безопасного кодирования и применения стандартов безопасности;
– изучение инструментов тестирования и мониторинга безопасности (SAST, DAST, SCA);
– интеграцию практик безопасности в процессы DevOps (DevSecOps).

Задачи изучения дисциплины:

– изучить основы безопасности при разработке ПО и её роль в SDLC;
– освоить методы безопасного кодирования и стандарты (OWASP, CERT);
– научиться выявлять и устранять уязвимости с помощью инструментов тестирования;
– применять принципы DevSecOps для интеграции безопасности в процессы разработки.

Изучив дисциплину, студент должен:

знать:

– основные угрозы безопасности при разработке ПО;
– принципы безопасного кодирования и стандарты (OWASP, CWE, ГОСТ);
– методы и инструменты тестирования безопасности (SAST, DAST, пентестинг);
– основы DevSecOps и её применения в современных проектах;

уметь:

– анализировать код на наличие уязвимостей;
– применять инструменты статического и динамического тестирования;

– интегрировать безопасность в процессы CI/CD;
– разрабатывать рекомендации по устранению уязвимостей;

владеть:

– навыками работы с инструментами SAST и DAST;
– методами безопасного проектирования и кодирования;
– практиками DevSecOps для обеспечения безопасности на всех этапах разработки.

Данное учебно-методическое пособие сочетает теоретические основы с практическими рекомендациями, что позволяет студентам не только понять принципы безопасности, но и научиться применять их на практике. Особый акцент сделан на интеграцию безопасности в процессы разработки (DevSecOps), что соответствует современным требованиям к созданию защищённого ПО.

Особое внимание уделено современным методам и инструментам обеспечения безопасности, таким как безопасное кодирование, DevSecOps, тестирование уязвимостей (SAST, DAST, IAST) и анализ состава программного обеспечения (SCA). Приведён детальный обзор угроз безопасности, включая уязвимости приложений, несанкционированный доступ, инъекции, межсайтовый скриптинг (XSS) и другие. Рассмотрены стандарты безопасного кодирования (OWASP, CERT, ГОСТ) и их применение в реальных проектах.

Пособие состоит из теоретической части, раскрывающей основные темы дисциплины, и практических работ, направленных на применение изученных методов в реальных проектах. Материал предназначен для подготовки специалистов, способных обеспечивать безопасность ПО на профессиональном уровне. Пособие также может быть полезно студентам других направлений и специальностей, изучающим вопросы кибербезопасности и защиты информации в современных цифровых системах.

Тема 1. ВВЕДЕНИЕ В БЕЗОПАСНОСТЬ ПРИ РАЗРАБОТКЕ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

Форма проведения занятия – лекция.

Оборудование к занятию: проектор, ноутбук.

Цель занятия – ознакомить студентов с основами безопасности при разработке ПО и ее важностью.

Вопросы для обсуждения

1. Вопросы безопасности при разработке ПО.
2. Почему важна безопасность при разработке ПО?
3. Этапы обеспечения безопасности при разработке ПО.
4. Обзор основных угроз безопасности при разработке ПО.

Методические указания по проведению занятия

При освоении темы необходимо:

1) изучить учебный материал по теме 1;
2) акцентировать внимание на основополагающих понятиях и определениях;
3) ответить на контрольные вопросы по теме 1;
4) выполнить тест по теме 1.

Методические материалы к занятию

1. Вопросы безопасности при разработке программного обеспечения

Разработка программного обеспечения – это процесс создания программных продуктов, которые решают определенные задачи. Однако в процессе разработки ПО может быть поставлена под угрозу безопасность приложения, а также данные, которые хранятся и обрабатываются внутри приложения. Поэтому безопасность при разработке ПО является критически важным аспектом, который должен быть учтен на всех этапах разработки.

Важно понимать, какие данные мы защищаем (рис. 1).

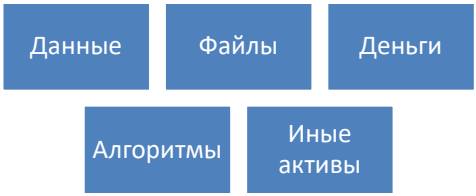

Рис. 1. Элементы защиты при разработке программного обеспечения

Безопасность при разработке ПО – это процесс организации защиты ПО от угроз безопасности, таких как хакерские атаки, вирусы, вредоносное ПО и другие угрозы. Она включает множество аспектов: конфиденциальность, целостность, доступность, аутентификацию и авторизацию, шифрование, обеспечение безопасности сети и тестирование безопасности. Разработчики должны учитывать эти аспекты на всех этапах разработки ПО, чтобы гарантировать, что приложение будет безопасным и защищенным от различных угроз:

1) *конфиденциальность* – защита информации от несанкционированного доступа, которая включает защиту конфиденциальных данных (личная информация, финансовые данные и др.);

2) *целостность* – сохранение данных в неповрежденном и неизменном состоянии, в том числе защита данных от несанкционированных изменений или искажений;

3) *доступность* – возможность получения доступа к информации и функциональность приложения в нужное время, включая обеспечение доступности приложения в случае сбоев и отказов, а также защиту от DDoS-атак и других угроз, которые могут привести к недоступности приложения;

4) *аутентификация и авторизация* – процессы, которые позволяют управлять доступом пользователей к приложению. Аутентификация проверяет, является ли пользователь тем, за кого себя выдает, а авторизация определяет, какие действия пользователь может выполнять в системе;

5) *шифрование* – процесс преобразования информации в непонятный для посторонних вид. Используется для защиты конфиденциальной информации от несанкционированного доступа;

6) *обеспечение безопасности сети* – процесс обеспечения защиты сетевого трафика от несанкционированного доступа. Разработчики должны обеспечить безопасность сети, используя специальные механизмы, такие как брандмауэры и VPN;

7) *тестирование безопасности* – процесс проверки наличия уязвимостей в приложении. Разработчики должны проводить тестирование безопасности для выявления и устранения уязвимостей в приложении.

Для обеспечения безопасности ПО необходимо использовать практики безопасного кодирования. Небезопасный, плохо спроектированный код может привести к проблемам безопасности ПО, таким как переполнение буфера, неправильно обработанные исключения, утечки памяти и неочищенный ввод. Если их не устранить, эти ошибки могут превратиться в полномасштабные уязвимости приложений, которые могут быть использованы – и часто используются – злоумышленниками для эксплуатации и атаки на программную инфраструктуру.

Современное ПО характеризуется высокой степенью сложности, обусловленной разнообразием функциональности, взаимодействием с различными компонентами и учетом множества потенциальных уязвимостей и угроз безопасности. В связи с этим меры безопасности ПО требуют комплексного подхода, включающего анализ уязвимостей, применение эффективных алгоритмов шифрования, установку защитных механизмов и постоянное обновление системы для защиты от новых угроз.

Цепочка поставок зависимостей, или процесс использования сторонних компонентов и библиотек при разработке ПО, даже для базовых приложений, может быстро стать сложной и запутанной. Эта цепочка состоит из множества взаимосвязанных элементов, каждый из которых представляет собой отдельную стороннюю библиотеку или модуль, предоставляемый другими разработчиками или организациями.

При использовании сторонних зависимостей разработчики полагаются на функциональность, которую предоставляют эти компоненты, и интегрируют их в свое приложение. Однако каждая из этих зависимостей может содержать свои собственные ошибки

или потенциальные уязвимости, которые могут оставаться незамеченными на первый взгляд.

Таким образом, цепочка поставок зависимостей может быстро превратиться в запутанную мозаику, где каждая сторонняя библиотека или модуль скрывают свои проблемы. Это означает, что даже при разработке базовых приложений необходимо тщательно анализировать и контролировать используемые зависимости, чтобы минимизировать риски возникновения ошибок или уязвимостей, которые могут повлиять на безопасность и стабильность приложения.

Для того чтобы гарантировать безопасность и надежность ПО, разработчики должны уделять особое внимание выбору и оценке сторонних зависимостей, следить за их обновлениями и устранением обнаруженных уязвимостей, а также принимать меры для контроля и проверки целостности и безопасности этих компонентов в рамках своего приложения.

Ключевым моментом в обеспечении безопасности ПО является наличие надлежащих инструментов и процессов, которые позволяют выявлять и устранять ошибки. Однако еще важнее, чтобы организации гарантировали, что их инженеры-программисты обладают правом собственности на свою работу и имеют свободу действий в борьбе с ошибками.

Принцип DevOps, который подразумевает интеграцию и сотрудничество между командой разработчиков и командой безопасности, играет важную роль в обеспечении безопасности ПО. В рамках этого принципа цикл быстрой обратной связи имеет особое значение. Быстрая и действенная обратная связь означает, что ошибки и уязвимости могут быть обнаружены и исправлены на ранних этапах разработки, что в итоге приводит к снижению общей частоты ошибок и уязвимостей, особенно на более поздних этапах жизненного цикла разработки.

Организации, где команды разработчиков и безопасности работают раздельно и с длинными циклами исправления и отчетности, обычно сталкиваются с проблемами. В таких случаях в ПО появляется большее количество ошибок, оно более уязвимо, что создает значительные трудности для обеспечения безопасности приложений.

Таким образом, важно стремиться к совместной работе и интеграции между командами разработчиков и безопасности, чтобы обеспечить непрерывную обратную связь, быстрое выявление и устранение ошибок и в конечном счете повысить уровень безопасности ПО.

2. Почему важна безопасность при разработке ПО

Согласно современным ИТ-концепциям, процессы разработки ПО и обеспечения безопасности идут рука об руку. Каждый аспект жизненного цикла разработки ПО (SDLC – Software Development Life Cycle) должен быть спроектирован таким образом, чтобы включать компоненты, ориентированные на безопасность. Эти усилия приводят к созданию скоординированной сети мер предосторожности в отношении данных, которые защищают всю информацию, используемую в ходе проекта, и предопределяют высококачественные ИТ-продукты, очищенные от опасных для данных ошибок, лазеек в коде и ошибок при разработке ПО.

В прошлом безопасность часто рассматривалась как второстепенный аспект при разработке ПО и учитывалась главным образом на этапе тестирования. Однако с появлением новых методологий, таких как Agile, внедрение непрерывного тестирования на каждом этапе SDLC стало нормой.

Хакеры и киберпреступники постоянно ищут новые способы использования уязвимостей программных систем. Придавая безопасности приоритетное значение на всех этапах SDLC, разработчики и заинтересованные стороны получают больше возможностей для выявления и устранения потенциальных рисков безопасности на ранних этапах разработки ПО. Это становится неотъемлемой частью процесса разработки ПО, где проблемы безопасности рассматриваются и решаются вместе с другими функциональными требованиями.

Интеграция безопасности на всех этапах SDLC, начиная с планирования и проектирования и заканчивая реализацией и тестированием, позволяет выявлять и устранять уязвимости как можно раньше. Это снижает риск возникновения серьезных проблем безопасности в конечном продукте и упрощает процесс обнару-

жения и исправления ошибок. Подобный подход также способствует более эффективному использованию ресурсов и времени разработчиков.

Интеграция принципов безопасности разработки ПО и непрерывное тестирование в SDLC, по-видимому, намного дешевле и выгоднее, чем тестирование и исправление пробелов в безопасности за ночь до выпуска продукта.

Тестирование безопасности должно быть постоянной областью внимания, а не разовой акцией. Чтобы избежать столкновения с дорогостоящими уязвимостями при использовании ПО, нельзя упускать из виду создание фундамента безопасности на этапе проектирования и разработки.

Безопасность разработки приложений называется гарантией безопасности (аналогично обеспечению качества) и зависит от следующих факторов:

1) непрерывного анализа архитектуры ПО на протяжении всего этапа проектирования;
2) непрерывных проверок кода на протяжении всего этапа разработки;
3) тщательного тестирования на проникновение (имитация попыток взлома) перед выпуском продукта.

Безопасный SDLC обеспечивает ряд преимуществ для организации:

1) устранение ошибок в дизайне. Благодаря безопасному SDLC возможно выявление и устранение ошибок в дизайне на ранних этапах разработки, еще до того как они будут реализованы в коде. Это позволяет предотвратить возникновение уязвимостей и проблем безопасности, снижая риск для организации;

2) снижение затрат. Безопасный SDLC способствует раннему обнаружению и устранению недостатков безопасности. Это позволяет избежать потенциальных угроз и атак, которые могут привести к финансовым потерям и репутационным проблемам. Раннее выявление и исправление ошибок также снижает затраты на последующие ремонтные работы и реагирование на инциденты безопасности;

3) поддержка осознания заинтересованными сторонами. Создание безопасного SDLC помогает заинтересованным сторонам осознать важность инвестиций в методологии безопасности. Это позволяет им понимать, что безопасность является неотъемлемой частью разработки ПО и не должна приноситься в жертву ускоренному выпуску продукта. Заинтересованные стороны будут больше поддерживать процессы безопасной разработки и обеспечивать необходимые ресурсы для успешной реализации безопасного SDLC.

SDLC – это последовательность шагов, которые разработчики ПО совершают при создании и сопровождении программных продуктов. Включение безопасности на каждом этапе SDLC является важным аспектом для защиты приложений от уязвимостей и атак.

Рассмотрим более подробно этапы рабочего процесса SDLC.

Этап 1. Концепция ПО + планирование

Первый этап включает определение требований к безопасности и к самому проекту. Необходимо установить основные принципы безопасности, которые должны быть реализованы в приложении. Также необходимо выбрать подходящую безопасную методологию SDLC и обеспечить тренинг по безопасности для членов команды разработки.

Важными моментами на данном этапе будут:

– определение требований к безопасности и целей успеха/соответствия для проекта;
– выбор безопасной методологии SDLC;
– тренинг по безопасности для членов команды;
– отбор человеческих ресурсов, обладающих экспертными знаниями/навыками в области безопасности приложений.

Этап 2. Архитектура ПО + дизайн

На этом этапе необходимо разработать безопасную архитектуру и дизайн приложения – создать безопасное ПО и моделировать киберугрозы. Разработчики должны определить вероятные сценарии атак и реализовать соответствующие контрмеры в архитектуре приложения. Также следует проверить стороннее ПО, используемое в проекте, на наличие уязвимостей.

Важными моментами на данном этапе будут:

– безопасная разработка ПО, поддерживаемая многочисленными проверками проекта, чтобы убедиться, что в проекте не осталось дефектов безопасности;
– моделирование киберугроз: определение вероятных сценариев атак и реализация контрмер в архитектуре ПО;
– проверка любого задействованного стороннего ПО на наличие уязвимостей.

Этап 3. Внедрение (разработка) ПО

На этом этапе происходит фактическая разработка приложения. Разработчики должны применять и совершенствовать принципы безопасности при написании кода. Статическое сканирование кода и ручные/автоматические проверки помогут выявить потенциальные уязвимости. Важно также, чтобы каждый член команды был обучен индивидуальным привычкам безопасности и основным мерам защиты данных.

Важными моментами на данном этапе будут:

– применение и совершенствование принципов безопасности в процедурах кодирования;
– статическое сканирование кода и ручные/автоматические проверки кода;
– отработка индивидуальных привычек безопасности и основных мер защиты данных на уровне каждого отдельного члена команды.

Этап 4. Тестирование ПО + исправление ошибок

Тестирование приложения должно включать обнаружение и исправление уязвимостей приложения. Ведение согласованной документации по обеспечению качества и использование различных методов тестирования помогут обеспечить высокое качество кода и безопасность приложения.

Важными моментами на данном этапе будут:

– обнаружение и исправление ошибок приложения;
– ведение согласованной документации по обеспечению качества;
– использование нескольких методов для обеспечения высочайшего качества кода.

Этап 5. Выпуск ПО + обслуживание

На этом этапе приложение готово к выпуску и эксплуатации. Однако безопасность должна продолжать быть приоритетом.

Важными моментами на данном этапе будут:

– постоянное совершенствование функций безопасности ПО;
– ведение журналов и обнаружение инцидентов, реагирование на них;
– управление ИТ-средой и повышение культуры пользователей;
– регулярные проверки безопасности и диагностика для обнаружения потенциальных проблем и уязвимостей.

Этап 6. Фаза окончания срока службы

На последнем этапе жизненного цикла ПО, когда оно больше не поддерживается создателями, необходимо обеспечить защиту важных или конфиденциальных данных, которые могут содержаться в приложении. Это может включать тщательное сохранение данных или полное прекращение использования приложения.

Рассмотрим более подробно основные причины, по которым безопасность при разработке ПО так важна:

1. *Защита конфиденциальной информации.* Конфиденциальная информация, такая как личные данные или финансовая информация, может быть скомпрометирована, если приложение не обеспечивает безопасность. Несанкционированный доступ к такой информации может привести к краже личных данных и финансовых средств, что может иметь серьезные последствия для пользователей.

2. *Защита от вредоносного ПО.* Вредоносное ПО может проникнуть в систему через уязвимости в приложении. Если приложение не обеспечивает безопасность, оно может стать легкой добычей для вредоносного ПО, что повлечет кражу данных и другие серьезные проблемы.

3. *Защита от хакерских атак.* Хакерские атаки могут нарушить работу приложения, привести к краже данных и повредить репутации приложения. Защита от хакерских атак является критически важным аспектом безопасности при разработке ПО.

4. *Защита от отказа в обслуживании.* Отказ в обслуживании (DDoS) – это атака, которая направлена на перегрузку сервера приложения. Если приложение не обеспечивает безопасность, оно может стать легкой добычей для таких атак, что может привести к недоступности приложения для пользователей.

5. *Соответствие требованиям законодательства.* Существуют законодательные требования в отношении защиты данных, такие

как GDPR (General Data Protection Regulation) – «Общий регламент по защите данных», нормативный акт Европейского союза. Несоблюдение этих требований может привести к штрафам и уголовной ответственности.

3. Этапы обеспечения безопасности при разработке ПО

На этапе *проектирования безопасности* определяются требования к безопасности и разрабатываются концепции безопасности для приложения с учетом оценки угроз.

На этапе *разработки безопасности* создаются механизмы безопасности, которые будут использоваться в приложении. Он включает разработку механизмов аутентификации и авторизации, шифрования данных, защиты сети и других механизмов, которые обеспечивают безопасность приложения.

На этапе *тестирования безопасности* проводится проверка безопасности приложения. Он включает проведение тестирования на уязвимости, анализ безопасности приложения и проверку соответствия требованиям безопасности.

На этапе *обновления безопасности* производится обновление механизмов безопасности приложения и устранение обнаруженных уязвимостей. Включает корректировку кода, изменение настроек безопасности и применение патчей безопасности.

На этапе *мониторинга безопасности* производится непрерывный мониторинг логов и событий приложения, анализ угроз безопасности и принятие мер по обеспечению безопасности приложения.

4. Обзор основных угроз безопасности при разработке ПО

Проблемы, связанные с уходом зарубежных производителей ПО, отсутствием обновлений безопасности и нарушением привычных цепочек поставок ПО, оказывают существенное влияние на информационную безопасность компаний. Разрыв связей между разработчиками и исследователями безопасности из разных стран приводит к тому, что в ПО будет значительно больше уязвимостей, о которых не знают разработчики, но которые могут быть выявлены злоумышленниками. Необходимость выстраивать новые цепочки поставок ПО и интегрировать в инфраструктуру новые решения, безопасность которых может быть под вопросом, также оказывает

негативное влияние на уровень защищенности организаций. Выделяют следующие угрозы безопасности при разработке ПО [5]:

1. *Уязвимости приложения*. Это слабые места в коде приложения, которые могут быть использованы злоумышленниками для получения несанкционированного доступа к приложению или данным, обрабатываемым приложением. Уязвимости могут возникать на любом этапе разработки ПО – от проектирования до тестирования и внедрения.

2. *Несанкционированный доступ* – доступ к приложению или данным, обрабатываемым приложением, без разрешения. Может быть осуществлен злоумышленниками, которые могут использовать различные методы, такие как взлом пароля, использование уязвимостей приложения и другие.

3. *Межсайтовый скриптинг (Cross-Site Scripting, XSS)*, это тип атаки, при которой злоумышленник внедряет вредоносный код в веб-страницу. Код будет выполняться на компьютере пользователя, открывшего эту страницу. Это может привести к краже данных пользователя, перенаправлению на другие сайты и другим негативным последствиям.

4. *Межсайтовая подделка запросов (Cross-Site Request Forgery, CSRF)* – это тип атаки, при которой злоумышленник отправляет поддельный запрос, выдавая его за запрос от другого пользователя. Это может привести к выполнению нежелательных действий на сервере или к краже данных пользователя.

5. *Атаки на сетевой уровень*. Это атаки на сетевые протоколы и сервисы, которые используются для обмена данными между компьютерами, в том числе атаки на протоколы TCP/IP, DNS, DHCP и другие сетевые протоколы.

6. *Фишинг*. Это тип атаки, при которой злоумышленник пытается обмануть пользователя, выдавая себя за легитимный источник, например, банк, электронную почту или социальную сеть. Целью фишинга является получение доступа к учетным данным пользователя или другой конфиденциальной информации.

Современное мышление подчеркивает важность безопасной разработки ПО как подхода, при котором программные приложения создаются и выполняются с учетом аспектов безопасности.

Это означает, что безопасность должна быть встроена в процессы разработки, а не рассматриваться как отдельный этап или дополнительная задача.

Даже если у вас есть доступ к лучшим инструментам тестирования для сканирования и анализа вашего ПО, успешный процесс безопасной разработки должен включать применение различных практик и методологий для выявления и устранения потенциальных угроз безопасности и слабых мест на каждом этапе жизненного цикла разработки ПО.

На начальных этапах разработки важно проводить анализ угроз и рисков, чтобы идентифицировать потенциальные уязвимости и определить соответствующие контрмеры. Затем следует аккуратное планирование и проектирование системы с учетом принципов безопасности. На этапе реализации необходимо использовать безопасные практики программирования, включая проверку входных данных, обработку ошибок и предотвращение уязвимостей, таких как инъекции и переполнения буфера.

После завершения разработки тестирование безопасности должно быть проведено с помощью различных методов, включая статический и динамический анализ кода, тестирование на проникновение и проверку соответствия стандартам безопасности. Важно также уделять внимание обновлениям и патчам, чтобы исправить обнаруженные уязвимости и поддерживать безопасность приложения в актуальном состоянии.

В целом безопасная разработка ПО требует интеграции безопасности в каждый этап жизненного цикла разработки, начиная с планирования и проектирования и заканчивая тестированием и обновлениями. Это позволяет обнаруживать и устранять уязвимости на ранних этапах разработки и обеспечивать непрерывную безопасность приложения.

Рассмотрим ведущие практики, рекомендованные для обеспечения безопасной разработки ПО:

1. Моделирование угроз для безопасного ПО – анализ архитектуры ПО и выявление потенциальных угроз безопасности и уязвимостей. Это помогает в разработке ПО с учетом безопасности и реализации необходимых средств контроля безопасности.

2. Безопасное кодирование ПО. Разработчики должны придерживаться методов безопасного кодирования, таких как проверка ввода, безопасное хранение данных и безопасные протоколы связи; они помогают предотвратить распространенные уязвимости системы безопасности: внедрение кода SQL, межсайтовые сценарии и атаки переполнения буфера.

3. Проверка кода, написанного разработчиками, помогает обнаруживать и исправлять уязвимости безопасности на ранних этапах процесса разработки.

4. Тестирование должно быть регулярным, тестирование на проникновение и сканирование уязвимостей может помочь выявить потенциальные слабые места безопасности в ПО. Это помогает устранить проблемы безопасности до развертывания ПО.

5. Безопасное управление конфигурацией – это настройка управления доступом, сетевых параметров и других параметров, связанных с безопасностью, для снижения риска несанкционированного доступа.

6. Контроль доступа гарантирует, что только авторизованный персонал может получить доступ к программной системе; для этого реализуют механизмы аутентификации и авторизации пользователей, а также управление доступом на основе ролей.

7. Регулярные обновления и исправления ПО помогают устранить уязвимости в системе безопасности и снизить риск нарушения безопасности. Важно быть в курсе исправлений безопасности и обновлений для всех программных компонентов, используемых в системе.

8. Обучение безопасности. Разработчики и другой персонал, участвующий в процессе разработки ПО, должны регулярно проходить обучение по вопросам безопасности, чтобы убедиться, что они понимают важность безопасности и знают лучшие практики безопасной разработки ПО.

9. Реагирование на инциденты. Организации должны иметь четко определенный план реагирования на инциденты безопасности: порядок выявления потенциальных инцидентов безопасности, минимизацию их последствий и последующее восстановление.

10. Непрерывный мониторинг помогает обнаруживать инциденты безопасности и реагировать на них в режиме реального времени; предусматривает мониторинг системных журналов, сетевого трафика и поведения пользователей на предмет любых признаков нарушений безопасности.

Следуя этим рекомендациям, организации могут разрабатывать безопасные и надежные программные приложения, способные противостоять потенциальным угрозам безопасности и уязвимостям. Крайне важно уделять приоритетное внимание безопасности на каждом этапе разработки ПО, чтобы предотвратить несанкционированный доступ и защитить конфиденциальные данные.

Контрольные вопросы

1. Что подразумевается под безопасностью при разработке ПО?
2. Почему важно учитывать аспекты безопасности ещё на этапе проектирования?
3. Какие основные цели обеспечиваются при соблюдении принципов безопасной разработки?
4. Какие риски могут возникнуть при игнорировании мер безопасности в процессе разработки ПО?
5. Перечислите основные этапы жизненного цикла ПО, где должна учитываться безопасность.

Тесты для самоконтроля

1. К жизненному циклу разработки ПО не относится этап

а) проектирования
б) тестирования
в) упаковки дисков
г) поддержки

2. Процесс активного поиска уязвимостей в системе описывается термином

а) статический анализ кода
б) тестирование на проникновение
в) резервное копирование
г) документирование

3. Принцип минимальных привилегий – это концепция безопасности, согласно которой

а) все пользователи получают полный доступ

б) пользователь имеет только необходимые права

в) доступ разрешён всем без проверки

г) права выдаются случайным образом

4. Чаще всего уязвимости обнаруживаются на этапе

а) планирования

б) тестирования

в) документирования

г) обучения

5. Расшифруйте аббревиатуру SDLC.

а) Software Development Life Cycle

б) Secure Data Loss Control

в) System Design Lifecycle

г) Security Detection and Logging Center

Рекомендуемая литература

1. Бабушкин, В. М. Разработка защищенных программных средств информатизации производственных процессов предприятия : учеб. пособие / В. М. Бабушкин, М. В. Тумбинская. – Москва [и др.] : Инфра-Инженерия, 2024. – 257 с. – URL: znanium.ru/catalog/document?id=451746 (дата обращения: 04.06.2025). – Режим доступа: по подписке. – ISBN 978-5-9729-1618-4.
2. Зиновьева, О. М. Интегрированные системы управления безопасностью : Разработка и аудит : практикум / О. М. Зиновьева, А. М. Меркулова, Н. А. Смирнова. – Москва : МИСиС, 2021. – 84 с. – URL: e.lanbook.com/book/238376 (дата обращения: 04.06.2025). – Режим доступа: по подписке.

Тема 2. БЕЗОПАСНОЕ КОДИРОВАНИЕ: ОСНОВЫ, СТАНДАРТЫ И МЕТОДЫ

Форма проведения занятия – лекция.

Оборудование к занятию: проектор, ноутбук.

Цель занятия – познакомить слушателей с основными аспектами безопасного кодирования, стандартами и методами, необходимыми для создания безопасного программного продукта.

Вопросы для обсуждения

1. Основные вопросы безопасного кодирования.
2. Стандарты безопасного кодирования.
3. Методы и инструменты безопасного кодирования.

Методические указания по проведению занятия

При освоении темы необходимо:

1) изучить учебный материал по теме 2;
2) акцентировать внимание на основополагающих понятиях и определениях;
3) ответить на контрольные вопросы по теме 2;
4) выполнить тест по теме 2.

Методические материалы к занятию

1. Основные вопросы безопасного кодирования

Быстрое предоставление безопасного ПО имеет решающее значение для большинства современных организаций. Некоторые разработчики могут склоняться к приоритизации скорости разработки перед безопасностью. Однако важно понять, что необходим компромисс между этими двумя требованиями. На самом деле возможно достичь их совмещения при условии правильного подхода, использования соответствующих инструментов и реализации эффективных процессов разработки. Фактически, стандарты безопасного кодирования позволяют организациям улучшать состояние безопасности своих приложений, не снижая скорости разработки.

Безопасное кодирование имеет решающее значение для предоставления высококачественного ПО, не подвергая организации риску инцидента безопасности. Небезопасный код, особенно в отраслях, имеющих дело с конфиденциальными данными, может привести к финансовому и репутационному ущербу, манипулированию рынком и краже, а также к другим негативным последствиям.

Основная идея безопасного кодирования с самого начала заключается в том, что оно обеспечивает создание приложений на прочной основе безопасности, что в свою очередь гарантирует защиту предприятий и их заинтересованных сторон. Путем интеграции безопасности во все этапы разработки можно снизить риски возникновения уязвимостей и потенциальных атак. Такой подход позволяет предотвратить поздние изменения и исправления, которые потребуют затрат средств и времени. В результате безопасное кодирование с самого начала помогает создавать надежные приложения, которые сохраняют конфиденциальность, целостность и доступность данных, а также укрепляют репутацию предприятия.

Безопасное кодирование – это набор методов, помогающих выявлять и устранять уязвимости кода, которые могут поставить под угрозу безопасность ПО. При безопасном кодировании злоумышленникам становится сложно взломать защиту и нанести вред ПО или данным.

Безопасное кодирование, безусловно, усиливает защиту ПО от злоумышленников. Тем не менее оно не может быть единственным барьером между программой и злоумышленником. ПО может быть защищено только с помощью набора уровней и методов безопасности, а не только с помощью безопасного кодирования.

Преимущество безопасного кодирования заключается в защите ПО от вредоносных атак. Безопасное кодирование осуществляется за счет:

1) устранения ошибок кодирования: наличие ошибочного кода – распространенная ошибка, совершаемая разработчиками непреднамеренно. Хотя ошибка невелика, последствия могут привести к эксплуатации злоумышленником. Могут быть зафиксированы такие действия, как переполнение буфера, неправильный формат строк и SQL-инъекции. При безопасном кодирова-

нии разработчик, как правило, проверяет код, что позволяет ему исправлять любую ошибку в коде, устраняя возможность распространенных атак;

2) следования стандартам кодирования. Набор руководящих принципов и стандартов даёт разработчику четкий путь к безопасности. Стандарты безопасного кодирования предоставляют разработчикам дорожную карту, которой они должны следовать.

3) более быстрого развертывания: разработка ПО сравнима с гонкой, где более быстрое развертывание приводит к лучшему результату. Чтобы не увеличивать затраты времени, методы безопасности внедряются на этапе разработки. На этом же этапе могут быть выявлены и исправлены распространенные ошибки. Все эти факторы сокращают SDLC, что приводит к более быстрому и безопасному развертыванию.

Благодаря безопасному кодированию разработчики могут предотвращать вредоносные атаки, исправляя код и устраняя уязвимости. С помощью безопасного кодирования можно выявить четыре типа уязвимости системы безопасности.

1. *Переполнение буфера*. Иногда разработчики неверно распределяют резервы памяти, необходимые для оптимального функционирования ПО. Недостаточное выделение приводит к утечке конфиденциальных данных в стеках памяти. Хакер может получить доступ к открытым данным и перезаписать их, что позволит ему контролировать ПО. Наиболее восприимчивыми к атакам переполнения буфера являются C++ и C.

2. *Внедрение кода*. Атаки с внедрением кода являются одними из наиболее распространенных атак. В отличие от многих других, эти атаки не распространены в каком-либо конкретном языке. Популярные языки: Ruby, SQL, Python, PHP, C++ и Java – могут стать жертвами таких атак, если не будет реализовано безопасное кодирование. В таких атаках злоумышленник отправляет в приложение код, который влияет на нужную функцию и заставляет ее работать так, как ему нужно.

Одним из распространенных типов внедрения кода является SQL-инъекция, при которой злоумышленник может получить доступ к базе данных (БД) веб-сайта. С помощью такой атаки зло-

умышленник может получить доступ к конфиденциальным данным, включая личные контактные данные, адреса электронной почты и банковские реквизиты.

3. *Программы с открытым исходным кодом*. Каждый разработчик любит использовать инструменты и программы с открытым исходным кодом для разработки. Организации с ограниченным бюджетом также предпочитают использовать такие программы, поскольку они бесплатны. Однако код таких программ является общедоступным и имеет несколько пробелов в безопасности. Злоумышленники знают о таких пробелах, которыми они, как правило, злоупотребляют, чтобы получить доступ к данным пользователей.

4. *Утечка ключей доступа*. Ключи доступа должны быть зашифрованы, чтобы никакая другая организация не могла их использовать. Однако разработчики могут встраивать их в var-файлы или локальные хранилища. Когда желаемый репозиторий является общедоступным, он становится уязвимым, поскольку злоумышленник может его использовать в преступных целях.

При написании кода необходимо использовать следующие методы защиты разработанной программы.

1. *Аутентификация* – предоставление доступа только авторизованным пользователям и только к важным данным, которые необходимы для выполнения. При этом нужно использовать надежные пароли и надежную систему управления паролями.

2. *Сканирование и проверка кода*. С помощью SQL-инъекции и XSS хакер пытается получить доступ к конфиденциальным данным. Использование средств автоматического сканирования может помочь выявить любые основные распространенные уязвимости безопасности в коде. Можно добавить ручные проверки кода. Частое обновление кода позволит исправить любую проблему безопасности в коде, которая могла проскользнуть мимо инструмента.

3. *Осторожное использование компонентов с открытым исходным кодом*. Бесплатные и популярные, они добавляют несколько функций программы, но также и являются причиной многочисленных вредоносных атак на их ПО. Перед использованием надо проверить, не содержит ли компонент какие-либо известные уязвимо-

сти. Кроме того, разработчик должен часто отслеживать компонент на наличие новых недостатков безопасности.

4. *Обфускация и минификация*. Обфускация – это метод, позволяющий сделать существующий код сложным для понимания злоумышленников. Минификация – это минимизация кода; разработчики стирают разрывы строк, избыточные символы, комментарии и пробелы из кода. Изначально минификация была направлена на повышение производительности и ускорение загрузки. Одновременно с этим она затрудняет чтение, предотвращая хакерские атаки.

5. *Управление ошибками*. Даже опытный разработчик обязательно столкнется с несколькими ошибками при написании кода. Правильная практика безопасного кодирования заключается в том, чтобы выявить и исправить ошибку, как только она будет замечена. Ведение журнала событий помогает точно выявлять проблемы. В этих журналах не должно быть конфиденциальных данных на случай несанкционированного доступа.

6. *Динамическое тестирование безопасности приложений* (DAST) помогает оценить степень устойчивости ПО. После того как ПО разработано, необходимо имитировать несколько сценариев кибератак, которые программа может наблюдать после выпуска. Успешное завершение DAST обнаружит любую существующую уязвимость безопасности в коде, что сделает его жизненно важной практикой безопасного кодирования.

7. *Неукоснительное следование рекомендациям стандартов безопасности*. Стандарты безопасности, такие как OWASP, CWE и NVD, устанавливают конкретные рекомендации для обеспечения надежности ПО в случае любых атак. Знание и соблюдение рекомендаций должно быть обязательным.

2. Стандарты безопасного кодирования

Недавнее исследование показало, что 28 % российских компаний не могут устранить уязвимости в своем оборудовании и ПО на этапе разработки. Это приводит к росту числа проблем безопасности.

Стандарты безопасного кодирования – это наборы правил и рекомендаций, используемых организацией для сокращения количества уязвимостей и числа ошибок во время разработки. Ис-

пользуемые стандарты будут различаться в разных организациях из-за разных требований безопасности (например, требования соответствия PCI для приложений, обрабатывающих платежи).

Существует ряд стандартизированных российских и зарубежных БД уязвимостей, которые разработчики могут использовать для управления уязвимостями.

1. База данных уязвимостей Российского центра киберзащиты (ЦКБ) – центрального органа, отвечающего за координацию и анализ киберзащиты в России – предоставляет информацию о российских и международных уязвимостях.

2. База данных уязвимостей Федеральной службы безопасности РФ (ФСБ России) включает информацию о критических уязвимостях, влияющих на информационную безопасность России.

3. ИСА Центра компетенции по информационной безопасности (ИСА ЦКИБ) включает информацию о различных уязвимостях, связанных с ПО и аппаратным обеспечением.

4. Open Web Application Security Project (OWASP) – некоммерческая организация, предлагающая бесплатные ресурсы для тестирования приложений. Она выявила десяток основных угроз безопасности веб-приложений, которые часто обновляются, информируя разработчиков о каждой распространенной атаке. Их руководство по тестированию веб-безопасности также постоянно обновляется.

5. Национальная база данных уязвимостей – National Vulnerability Database (NVD) управляется Национальным институтом стандартов и технологий – агентством правительства США. Эти данные позволяют автоматизировать управление уязвимостями, оценивать безопасность и обеспечивать соответствие требованиям.

6. Common Weakness Enumeration (CWE) – система классификации распространенных недостатков безопасности аппаратного и программного обеспечения в наиболее часто используемых языках, включая C++, Java и C. Список CWE составляется на основе постоянных исследований и отзывов пользователей, что позволяет выявлять наиболее катастрофические уязвимости безопасности.

7. Defense Information Systems Agency (DISA) предназначено для приложений, развертываемых в министерстве обороны США (DoD), оно помогает внутренним и сторонним приложениям

в разработке и оценке. Агентство по оборонным информационным системам предлагает различные руководства по технической реализации безопасности – Security Technical Implementation Guides (STIG), которые помогают внедрить методы безопасности в приложениях, разрабатываемых в военных целях.

8. Группа реагирования на компьютерные инциденты – Computer Emergency Response Team (CERT), управляемая Институтом разработки ПО Университета Карнеги – Меллона, представляет собой глобальную сеть экспертов по кибербезопасности. Основное внимание уделяется анализу уязвимостей, реагированию на инциденты и разработке лучших практик для повышения кибербезопасности. Сотрудничая с организациями по всему миру, CERT предоставляет ресурсы, опыт и рекомендации для эффективного предотвращения, обнаружения инцидентов кибербезопасности и реагирования на них.

9. Common Vulnerabilities and Exposures (CVE) – база данных, запущенная в 1999 году корпорацией MITRE, представляет собой список публично раскрываемой информации об уязвимостях безопасности. Это бесплатный словарь, доступный организациям, стремящимся легко делиться информацией о широко известных уязвимостях. Присваивая стандартизированный идентификатор каждой уязвимости, он упрощает обмен информацией между организациями.

В настоящее время многие стандарты безопасности содержат рекомендации и информацию о предотвращении кибератак и повышении безопасности ПО. Вот основные из них:

1. Стандарт безопасности данных платежных приложений (PA-DSS) принят в 2008 году Советом по безопасности индустрии платежных карт (PCI SSC), представляет собой глобальный стандарт безопасности, в котором изложены требования безопасности к ПО платежных приложений, участвующему в обработке транзакций по картам. Этот стандарт безопасности направлен на то, чтобы платежные приложения, разработанные для третьих лиц, не хранили конфиденциальную информацию о карте. Чтобы соответствовать этому стандарту, поставщик должен сертифицировать приложения по четырнадцати защитам, установленным PA-DSS.

2. ГОСТ Р ИСО/МЭК 27001-2013 определяет требования к системам управления информационной безопасностью (СУИБ). Он устанавливает подходы к управлению рисками, защите информации и непрерывности бизнеса.

3. ГОСТ Р ИСО/МЭК 27002-2015 предоставляет рекомендации и руководство по управлению информационной безопасностью. Он охватывает широкий спектр мер и контролей, направленных на обеспечение безопасности информации.

В целях своевременной защиты процесса разработки ПО Международная организация по стандартизации (ИСО/МЭК СТК 1/ПК 27) – совместный технический комитет 1 и подкомитет 27 – включила «Безопасное кодирование» в качестве нового элемента управления 8.28 в новые стандарты ИСО/МЭК 27002 и ИСО/МЭК 27001:2022. Целью технической меры безопасного кодирования является превентивная защита ПО.

Минимальный уровень безопасности создается уже на этапе написания на основе процедурных регламентов, что сводит к минимуму количество потенциальных уязвимостей безопасности в ПО. Нормативно-правовая база для безопасной генерации кода должна применяться целостно как к собственному программному коду компании, так и к ПО от третьих лиц и источников с открытым исходным кодом. Основными принципами реализации являются проектная безопасность и наименьшие привилегии, которые также необходимо учитывать при кодировании.

Положение 8.28 затрагивает так называемые **принципы безопасного кодирования**. Рекомендации о том, какими они могут быть, предоставляются многочисленными организациями и институтами, которые регулярно выпускают руководства и передовые методы безопасного кодирования. К ним относятся, например, Методы безопасного кодирования OWASP, Безопасные стандарты кодирования CERT Института программной инженерии (SEI) и каталог мер безопасности веб-приложений BSI Германии. Разработки из различных источников имеют много параллелей, например в отношении следующих моментов:

1) валидации вводимых данных;
2) обеспечения безопасности аутентификации и управления паролями;
3) безопасного контроля доступа;
4) простого, прозрачного кода;
5) устойчиво протестированных криптографических мер и компонентов;
6) обработки ошибок и ведения журнала;
7) защиты данных;
8) моделирования угроз.

В стандарте ISO/IEC 27001:2022 рекомендуемые действия для Control 8.28 сгруппированы в три раздела: «Планирование и предварительное кодирование», «Во время кодирования» и «Проверка и обслуживание».

Перед вводом ПО в эксплуатацию Control 8.28 требует оценки поверхностей атаки и реализации принципа наименьших привилегий. Должен быть проведен анализ наиболее распространенных ошибок программирования и документации по их исправлению.

Что касается проверки и обслуживания, то в Control 8.28 дополнительно перечислены четкие инструкции по использованию внешних инструментов и библиотек, таких как ПО с открытым исходным кодом. Этими компонентами кода следует управлять, обновлять их и инвентаризировать. Это можно сделать, например, с помощью спецификации ПО (Software Bill of Materials – SBOM). SBOM – это формальная, структурированная запись пакетов и библиотек ПО и их взаимосвязей друг с другом и в цепочке поставок, в частности, для отслеживания повторно используемого кода и компонентов с открытым исходным кодом. SBOM поддерживает удобство сопровождения ПО и целевые обновления безопасности.

3. Методы и инструменты безопасного кодирования

Исходный код представляет собой набор инструкций, которые определяют поведение приложения и реализуют его функциональные возможности. По сути, это «ДНК» приложения. Исходный код преобразуется в инструкции, которые затем считываются и выполняются компьютером.

Безопасное кодирование, также известное как безопасное программирование, включает написание кода на языке высокого уровня в соответствии с жесткими принципами, чтобы предотвратить возможные уязвимости, которые могут привести к компрометации данных или нанести вред целевой системе.

Однако безопасное кодирование – это не только написание, компиляция и выпуск кода в приложения. Чтобы полностью охватить безопасное программирование, необходимо также создать безопасную среду разработки, основанную на надежной ИТ-инфраструктуре, используя безопасное оборудование, ПО, услуги и поставщиков.

Основные методы безопасного кодирования:

1) санитизация входных данных:

– проверка и фильтрация пользовательского ввода перед его использованием в программе;
– использование механизмов валидации и санитизации данных для предотвращения атак вроде инъекций SQL или XSS (межсайтового скриптинга);

2) правильное управление аутентификацией и авторизацией:

– использование сильных алгоритмов шифрования для хранения паролей пользователей;
– ограничение доступа к определенным функциям и ресурсам, предоставление его только уполномоченным пользователям;

3) защита от межсайтовых сценариев (XSS):

– экранирование и фильтрация выводимых данных, чтобы предотвратить внедрение вредоносных скриптов в веб-страницы;
– использование контекстно зависимых механизмов экранирования, таких как Content Security Policy (CSP);

4) защита от инъекций:

– использование параметризованных запросов (prepared statements) или хранимых процедур для предотвращения инъекций SQL;
– использование специфических API и библиотек, которые предотвращают инъекции кода;

5) обработка исключений и ошибок:

– корректная обработка ошибок и исключений в коде с целью предотвращения утечек информации и возникновения непредвиденного поведения программы;

– минимизация раскрытия деталей ошибок, чтобы злоумышленники не получили информацию, которая может помочь им в атаке;

6) регулярные обновления и исправления:

– поддержка и обновление используемых библиотек, фреймворков и компонентов ПО для устранения известных уязвимостей;
– применение патчей и исправлений безопасности, предоставляемых разработчиками, на регулярной основе;

7) обеспечение надежного шифрования. Существует множество легкодоступных библиотек, которые помогут реализовать шифрование, что требует написания минимального пользовательского кода. Однако важно использовать только стандартные алгоритмы и библиотеки. Кроме того, следует убедиться, что всякий раз, когда требуется соответствие FIPS, используются только проверенные библиотеки;

8) управление уязвимостями: никогда нельзя жестко кодировать или загружать секреты, такие как пароли или ключи доступа, в репозитории кода.

Вышеуказанные меры защитят систему и будут первой линией защиты, но также важно сделать сам код более безопасным.

Лучшие практики безопасного кодирования

Минимизация и обфускация кода. Затруднение доступа к коду и его чтения может отпугнуть потенциальных злоумышленников. В мире JavaScript распространенной практикой является минимизация кода. Минификация уменьшает вес файлов, удаляя пробелы и разрывы строк из кода, это повышает производительность и значительно затрудняет чтение открытого кода. Другим похожим, более эффективным методом является обфускация кода, которая превращает удобочитаемый код в текст, который трудно понять.

Избегание ярлыков. У разработчиков может возникнуть соблазн использовать ярлыки для более быстрого выпуска кода в рабочую среду, но это может иметь серьезные последствия для безопасности. Например, атаки часто происходят, когда жестко запрограммированные учетные данные и маркеры безопасности остаются в качестве комментариев. Эта информация должна быть почищена задолго до выпуска приложений. По мере того, как кодовая база увеличивается и растет давление, связанное с предоставлением ра-

бочего кода во все более сжатые сроки выпуска, вероятность пробелов в безопасности возрастает.

Автоматическое сканирование и проверка кода. Атаки XSS и SQL-инъекции возникают из-за слабости кода, которая не может различать данные и команды. XSS выполняет вредоносный код в домене. Атаки SQL-инъекций пытаются украсть данные или манипулировать ими во внутренних хранилищах. Сочетание регулярных проверок кода безопасности и автоматизированных инструментов, которые сканируют код на наличие этих уязвимостей, может помочь предотвратить такие атаки.

Отказ от использования компонентов и библиотек с открытым исходным кодом. Они являются общей точкой входа для злоумышленников и отличным источником потенциальных эксплойтов.

Аудит и ведение журнала позволят обнаруживать потенциальные инциденты при развертывании кода в производственной среде. Snyk – один из лучших бесплатных инструментов для сканирования и мониторинга безопасности кода. Можно использовать сканер уязвимостей с открытым исходным кодом или Snyk Code для поиска и устранения уязвимостей кода с помощью удобного для разработчиков интерфейса.

Создание культуры безопасности в организации включает обучение разработчиков, ИТ, организационного менеджмента, а также внутренних и внешних заинтересованных сторон. Необходимо создавать модели угроз и планировать управление потенциальными рисками и их устранение.

Во все этапы жизненного цикла безопасной разработки ПО (SSDLC) нужно интегрировать методы безопасного программирования – от начального этапа сбора требований к новому приложению (или добавления функций и возможностей к существующему приложению) до разработки, тестирования, развертывания и обслуживания. На самом раннем этапе сбора заинтересованные лица проекта уже должны начать анализировать требования и отмечать потенциальные риски безопасности, особенно те, которые связаны с исходным кодом.

Использование автоматизированных инструментов в рамках SSDLC или других инициатив по безопасному кодированию эко-

номит время и усилия. На ранних этапах цикла разработки может быть реализован метод анализа безопасности, при котором исходный код приложения проверяется без его выполнения.

Статическое тестирование безопасности приложений (Static Application Security Testing – SAST) ограничено длительным временем сканирования и низкой точностью, возвращая слишком много ложных срабатываний и подрывая доверие разработчиков. Семантический анализ кода в режиме реального времени предоставляет действенные предложения сразу после написания кода, обеспечивая скорость и качество результатов в рабочем процессе разработчика.

Безопасная проверка кода (Secure Code Review) – это диагностика и устранение любых проблем безопасности или функциональности в коде. Проверка кода может быть выполнена с помощью инструментов или вручную, так как она зависит от требований проекта и предпочтений разработчика.

Средства обучения безопасному коду созданы, чтобы помочь разработчикам с методами безопасного кодирования, чтобы гарантировать, что каждый код является безопасным, надежным и совместимым. Эти инструменты помогают выявлять и анализировать потенциальные проблемы или уязвимости в режиме реального времени, что позволяет программистам исправить свою работу до того, как она будет выпущена в производственную среду. Разработчики обучаются методам безопасного кода для конкретных языков программирования с использованием различных инструментов.

Статические анализаторы кода выполняют анализ и проверку исходного кода на наличие потенциальных уязвимостей и проблем безопасности. Они могут выявлять такие проблемы, как некорректное использование аутентификации, небезопасные операции с памятью, несанкционированный доступ к данным и другие уязвимости. Примеры статических анализаторов: SonarQube, Fortify, Checkmarx и Coverity.

Динамические анализаторы кода проводят анализ кода во время его выполнения и могут обнаруживать уязвимости, которые могут быть сложными для обнаружения статическим анализом. Они позволяют выявлять проблемы, связанные с некорректной обработкой пользовательского ввода, уязвимостями сетевого вза-

имодействия и другими динамическими аспектами приложения. Примеры динамических анализаторов: Burp Suite, OWASP ZAP и IBM AppScan.

Обучающие онлайн-платформы и курсы предлагают обучающие материалы, практические задания и тесты, которые помогают разработчикам изучить методы безопасного кодирования и применить их на практике. Примеры таких платформ: OWASP Juice Shop, WebGoat и HackEDU.

Руководства и онлайн-ресурсы предлагают советы, рекомендации и примеры по безопасному кодированию. Эти ресурсы могут быть полезными для самообразования и получения дополнительной информации о методах безопасного кодирования.

Важно отметить, что инструменты обучения безопасному коду могут различаться в зависимости от языка программирования и применяемых технологий. Разработчики могут выбирать те инструменты, которые наиболее подходят для их конкретного стека технологий и задач.

При внедрении стандартов безопасного кодирования крайне важно, чтобы команды безопасности не пытались навязать новые инструменты командам разработчиков. Это может создать трения и замедлить развертывание новых инструментов и процессов безопасности в долгосрочной перспективе. Вместо этого группы безопасности должны выбрать удобные для разработчиков инструменты, чтобы обеспечить эффективное внедрение и повысить их производительность.

В частности, удобные для разработчиков инструменты должны легко интегрироваться с существующими рабочими процессами разработчика. Разработчики с гораздо большей вероятностью будут использовать инструменты безопасности, которые не замедлят их работу и не добавят много дополнительных усилий. Беспроблемный инструмент безопасности также побудит разработчиков взять на себя большую ответственность за безопасность приложений.

Контрольные вопросы

1. Какие основные ошибки могут привести к уязвимостям в коде?
2. Что такое OWASP Top 10?
3. Какие существуют стандарты безопасного кодирования?
4. Какова роль статического анализа кода в обеспечении безопасности?
5. Какие практики помогают предотвратить SQL-инъекции?

Тесты для самоконтроля

1. Для предотвращения SQL-инъекций используют

а) хардкодинг паролей
б) параметризованные запросы
в) отключение шифрования
г) открытие всех пор сервера

2. SAST – это

а) Static Application Security Testing
б) Safe Authentication System Test
в) Secure Access Service Token
г) Server Analysis of Security Threats

3. Для анализа уязвимостей в зависимостях может использоваться

а) Bandit
б) SQLmap
в) Snyk
г) Wireshark

4. Валидация ввода – это

а) удаление данных
б) проверка корректности данных от пользователя
в) шифрование данных
г) запись логов

5. Этот стандарт рекомендует использовать криптографически стойкие алгоритмы.

а) ISO 9001

б) ISO/IEC 27002

в) IEEE 802.11

г) PCI DSS

Рекомендуемая литература

1. Авдошин, С. М. Технологии и продукты Microsoft в обеспечении информационной безопасности : учеб. пособие / С. М. Авдошин, А. А. Савельева, В. А. Сердюк. – 4-е изд. стер. электрон. – Москва : Интернет-Университет Информационных Технологий [и др.], 2025. – 431 с. – URL: www.iprbookshop.ru/146405.html (дата обращения: 04.06.2025). – Режим доступа: по подписке. – ISBN 978-5-4497-0935-6.
2. Зиновьева, О. М. Интегрированные системы управления безопасностью : Разработка и аудит : практикум / О. М. Зиновьева, А. М. Меркулова, Н. А. Смирнова. – Москва : МИСиС, 2021. – 84 с. – URL: e.lanbook.com/book/238376 (дата обращения: 04.06.2025). – Режим доступа: по подписке.

Тема 3. БЕЗОПАСНОСТЬ ПРИЛОЖЕНИЙ APPSEC

Форма проведения занятия – лекция.

Оборудование к занятию: проектор, ноутбук.

Цель занятия – познакомить студентов с основными методами и принципами безопасного кодирования приложений, которые позволяют предотвратить уязвимости и обеспечить высокий уровень безопасности в разрабатываемых программных продуктах.

Вопросы для обсуждения

1. Какие основные методы и принципы безопасного кодирования приложений существуют?
2. Какие уязвимости и угрозы могут возникать в приложениях из-за небезопасного кодирования?
3. Какие инструменты и практики можно использовать для обнаружения и предотвращения уязвимостей в коде приложений?

Методические указания по проведению занятия

При освоении темы необходимо:

1) изучить учебный материал по теме 3;
2) акцентировать внимание на основополагающих понятиях и определениях;
3) ответить на контрольные вопросы по теме 3;
4) выполнить тест по теме 3.

Методические материалы к занятию

1. Какие основные методы и принципы безопасного кодирования приложений существуют?

Безопасность приложений имеет критическое значение, поскольку уязвимости в ПО остаются одной из самых распространённых причин инцидентов информационной безопасности: по имеющимся данным, на долю прикладного уровня приходится 84 % всех зафиксированных случаев компрометации. Приложения, как правило, обрабатывают конфиденциальные данные компаний и пользователей, что делает их первоочередной целью для атак.

В случае компрометации канала взаимодействия между легитимной организацией и её пользователем атакующий может воспользоваться рядом методов – от внедрения вредоносного кода и нарушения контроля доступа до ошибок в настройке криптографических механизмов и небезопасной конфигурации – с целью похищения учётных данных, корпоративной информации и других ценных ресурсов.

AppSec (Application Security) – это подход к защите приложений от внешних угроз и уязвимостей на всех этапах их жизненного цикла. Это не одна технология, а широкий спектр методов.

Все подходы к обеспечению безопасности приложений направлены на одну цель – выявление, снижение и предотвращение уязвимостей. Различия между ними обусловлены контекстом применения: где, когда и каким образом осуществляется тестирование, а также какие методологии и практики используются.

Безопасность мобильных приложений фокусируется на защите ПО, разрабатываемого для мобильных платформ – в первую очередь Android и iOS (поддержка Windows Phone в настоящее время минимальна). Она охватывает приложения, работающие как на смартфонах, так и на планшетах, и предполагает оценку их устойчивости к угрозам с учётом особенностей целевой платформы, используемых инструментов разработки и профиля конечных пользователей – будь то сотрудники организации или внешние клиенты. Тестирование безопасности мобильных приложений моделирует действия злоумышленника, стремящегося скомпрометировать приложение. Эффективный процесс начинается с анализа назначения приложения и характера обрабатываемых данных, после чего применяется комбинация статического анализа (SAST), динамического анализа (DAST) и тестирования на проникновение, что позволяет выявить уязвимости, которые остаются незамеченными при использовании только одного из этих методов.

Безопасность облачных приложений представляет собой совокупность политик, процессов и технологических средств, обеспечивающих защиту приложений и данных в облачных средах. Ключевые направления облачной безопасности включают управление идентификацией и доступом, защиту данных, обеспечение безо-

пасности инфраструктуры, ведение журналов и мониторинг, реагирование на инциденты, а также непрерывное устранение уязвимостей и анализ конфигураций. Особое внимание уделяется модели совместной ответственности, в рамках которой облачный провайдер отвечает за безопасность самой инфраструктуры, а заказчик – за безопасность размещаемых в облаке приложений, данных и их настройки.

Безопасность веб-приложений – это практика проектирования и разработки веб-сайтов таким образом, чтобы они сохраняли функциональность и целостность даже в условиях активных атак. Она включает внедрение механизмов защиты непосредственно в код приложения для охраны его активов от вредоносных воздействий. Любое ПО неизбежно содержит дефекты, некоторые из них могут быть эксплуатированы как уязвимости, безопасность веб-приложений направлена на их своевременное выявление и устранение. Это достигается за счёт применения принципов безопасной разработки на всех этапах жизненного цикла ПО – от проектирования до эксплуатации. В процессе верификации используются такие методы, как динамический анализ (DAST), статический анализ (SAST), тестирование на проникновение и механизмы самозащиты приложений во время выполнения (RASP).

Основные методы и принципы использования AppSec включают:

– идентификацию потенциальных угроз и оценку рисков (анализ уязвимостей, угроз, модели атаки и другие методы);

– защиту в глубину – принцип, основанный на использовании нескольких слоев защиты: на уровне сети, операционной системы, приложения и данных;

– аутентификацию и авторизацию для проверки подлинности пользователей и присвоение соответствующих привилегий доступа (использование сильных паролей, многофакторной аутентификации, управления сеансами и других техник);

– обработку пользовательского ввода для предотвращения уязвимостей, связанных с инъекциями (например, SQL-инъекции или XSS-атаки). Включает валидацию данных, экранирование символов и использование параметризованных запросов;

– защиту данных – применение механизмов шифрования и защиты данных в покое и в движении: шифрование хранимых данных, использование защищенных протоколов связи (например, HTTPS) и корректная обработка конфиденциальной информации;

– управление уязвимостями – регулярное обновление и патчинг приложений для устранения известных уязвимостей (мониторинг уязвимостей, установка обновлений и применение лучших практик в области безопасности разработки);

– тестирование безопасности – проведение тестирования на проникновение (penetration testing) и др. видов тестирования (сканирование уязвимостей, анализ кода и т. д.);

– обучение и осведомленность – обучение разработчиков и других заинтересованных лиц.

AppSec является непрерывным процессом, требующим постоянного внимания и обновления с учетом новых угроз и развития технологий. Эффективная реализация методов AppSec помогает уменьшить риски и обеспечить безопасность приложений в современной информационной среде.

2. Какие уязвимости и угрозы могут возникать в приложениях из-за небезопасного кодирования?

Неправильное или небезопасное кодирование может привести к серьезным проблемам.

Инъекции (Injection) возможны, когда недостаточно проверяется или фильтруется пользовательский ввод, который затем выполняется как код или запрос к БД без должной обработки.

Примеры: SQL-инъекции, инъекции команд в операционной системе (OS Command Injection), инъекции кода JavaScript (XSS).

Меры предосторожности: используйте параметризованные запросы или хранимые процедуры для работы с БД, фильтруйте и проверяйте входные данные, используйте контекстно-зависимые санитайзеры для предотвращения XSS.

Межсайтовый скриптинг (Cross-Site Scripting – XSS): злоумышленник внедряет вредоносный скрипт в веб-страницу, который выполняется в браузере другого пользователя.

Для термина используют сокращение XSS, чтобы не было путаницы с CSS – Cascading Style Sheets (каскадные таблицы стилей).

По местонахождению вредоносного скрипта атаки делят на хранимые (stored) и отраженные (reflected).

Меры предосторожности: экранируйте и кодируйте выводимые данные, используйте контекстно зависимые санитайзеры, применяйте Content Security Policy (CSP) для ограничения источников выполнения скриптов.

Межсайтовая подделка запроса (Cross-Site Request Forgery – CSRF): злоумышленник вынуждает авторизованного пользователя выполнить нежелательное действие в веб-приложении, в котором пользователь имеет активную сессию, например, злоумышленник отправляет пользователю ссылку или вредоносный код, который автоматически выполняет нежелательное действие на сайте.

Меры предосторожности: используйте механизмы защиты от CSRF (токены CSRF), проверка HTTP-заголовков Referer и Origin, двухфакторная аутентификация.

Небезопасное хранение данных: конфиденциальная информация (пароли или личные данные пользователей) хранится в незашифрованном или неправильно защищенном виде. Например, пароли хранятся в виде чистого текста, не зашифрован кэш браузера с конфиденциальными данными.

Меры предосторожности: хешируйте пароли перед сохранением, используйте криптографические алгоритмы для защиты конфиденциальных данных.

Небезопасная аутентификация и управление сеансами. Если система аутентификации и управления сеансами не реализована должным образом, это может привести к компрометации учетных данных пользователей и несанкционированному доступу к аккаунтам.

Примеры: хранение паролей в открытом виде, использование слабых алгоритмов хеширования, отсутствие механизмов защиты от подбора паролей, недостаточно длительные или неправильно управляемые сеансы.

Меры предосторожности: используйте безопасные методы хеширования паролей (bcrypt или Argon2), внедрите механизмы двухфакторной аутентификации, установите правильные сроки действия сеансов, используйте случайные токены сеансов, предотвращайте атаки подбора паролей.

Недостаточная обработка ошибок: приложение не адекватно обрабатывает и регистрирует ошибки, что может раскрыть ценную информацию или привести к отказу в обслуживании.

Примеры: отображение подробных сообщений об ошибках, включающих конфиденциальную информацию, отсутствие обработки исключений, неправильное логирование ошибок.

Меры предосторожности: отображайте общие сообщения об ошибках без раскрытия конфиденциальной информации, обрабатывайте исключения и ошибки, регистрируйте их в журнале событий без раскрытия подробностей, используйте мониторинг ошибок для обнаружения проблем.

Недостаточная проверка прав доступа: приложение недостаточно проверяет и ограничивает доступ пользователей к ресурсам или функциям, что может привести к несанкционированному доступу и компрометации данных.

Примеры: отсутствие проверки прав доступа на серверной или клиентской стороне, недостаточная проверка идентификаторов объектов или параметров запросов.

Меры предосторожности: реализуйте принцип наименьших привилегий, проверяйте права доступа на серверной стороне перед выполнением операций, применяйте контроль доступа на уровне ролей и разрешений.

Мы рассмотрели некоторые примеры уязвимостей и угроз, связанных с небезопасным кодированием. Безопасность приложений требует всестороннего подхода – от обучения разработчиков до использования правильных практик и инструментов разработки.

3. Какие инструменты и практики можно использовать для обнаружения и предотвращения уязвимостей в коде приложений?

Обеспечить надежность кода и защитить приложение от потенциальных уязвимостей можно с помощью нескольких практик.

1. Аудит кода – процесс ревизии и анализа исходного кода приложения с целью обнаружения потенциальных уязвимостей и слабых мест в безопасности. Проводите регулярные кодовые ревью, используйте статические анализаторы кода и инструменты сканирования уязвимостей для автоматического обнаружения проблем.

2. Внедрение тестирования на проникновение. Penetration Testing – это процесс активного тестирования приложения с целью выявления уязвимостей и оценки его устойчивости к атакам. Нанимайте профессионалов по тестированию на проникновение для проведения тестов на реальных или контролируемых средах, используйте автоматизированные инструменты для обнаружения уязвимостей.

3. Применение принципа наименьших привилегий (Least Privilege), который заключается в предоставлении пользователям только тех прав доступа, которые необходимы для выполнения их работы. Ограничивайте доступ пользователей только к необходимым ресурсам и функциям, рассматривайте применение привилегированных аккаунтов с осторожностью, регулярно аудитируйте права доступа.

4. Регулярное обновление и патчинг приложений и используемых компонентов. Следите за обновлениями и патчами для используемых вами фреймворков, библиотек и операционных систем, регулярно обновляйте свое приложение, используйте системы управления конфигурацией для облегчения процесса обновления.

5. Обеспечение безопасности данных включает шифрование конфиденциальной информации, правильное хранение паролей, защиту данных в памяти и в пути. Используйте надежные алгоритмы шифрования для хранения паролей и конфиденциальных данных, применяйте шифрование при передаче данных по открытым сетям, используйте механизмы защиты от инъекций (например, SQL-инъекций), обеспечивайте безопасное хранение и обработку конфиденциальных данных.

6. Обучение разработчиков в области безопасного кодирования и знание актуальных уязвимостей помогают предотвратить возникновение уязвимостей на ранних стадиях разработки. Проводите регулярные тренинги и обучение разработчиков по безопасному кодированию, осознавайте значимость безопасности в процессе разработки, используйте проверенные руководства и ресурсы по безопасности кода.

7. Регулярный мониторинг и журналирование позволяют обнаружить подозрительную активность и атаки, а также получать

информацию о возможных уязвимостях в приложении. Внедрите механизмы мониторинга безопасности, включая журналирование аудита, событий безопасности и сбор статистики, анализируйте ошибки и реагируйте на подозрительную активность.

8. Тестирование безопасности на всех этапах разработки начиная с проектирования и заканчивая эксплуатацией приложения. Включите тестирование безопасности в свои процессы разработки, используйте инструментарий для автоматизации тестирования безопасности, проводите регулярные проверки с учетом актуальных угроз и уязвимостей.

Помимо перечисленных практик также существуют **инструменты безопасной разработки**, которые могут значительно облегчить процесс обнаружения и предотвращения уязвимостей в коде приложений.

Сканеры уязвимостей:

– OWASP ZAP – бесплатный инструмент для сканирования уязвимостей, основанный на прокси-сервере;

– Burp Suite – коммерческий инструмент для сканирования уязвимостей, который включает прокси-сервер, сканер уязвимостей, инструменты для тестирования на проникновение и другие функции.

Сканеры кода:

– SonarQube – платформа для статического анализа кода, которая помогает обнаружить уязвимости и проблемы безопасности на ранних стадиях разработки;

– Checkmarx – коммерческий инструмент для статического анализа кода, который идентифицирует уязвимости и проблемы безопасности в исходном коде приложений.

Инструменты для тестирования на проникновение:

– Metasploit – популярный фреймворк для тестирования на проникновение, который включает набор инструментов для обнаружения и эксплуатации уязвимостей;

– Nessus – коммерческий инструмент для сканирования сети и обнаружения уязвимостей, который также включает функциональность тестирования на проникновение.

Фреймворки безопасности:

– Spring Security – фреймворк безопасности для приложений на платформе Java, который предоставляет инструменты для аутентификации, авторизации, защиты от атак и других функций безопасности;

– Django Security – набор инструментов и рекомендаций безопасности для фреймворка Django в языке Python.

Инструменты для анализа уязвимостей в зависимостях:

– Dependency-Check – инструмент, который проверяет зависимости проекта на наличие известных уязвимостей и предоставляет отчет о найденных проблемах;

– Snyk – инструмент, который обнаруживает и предупреждает об уязвимостях в сторонних пакетах и библиотеках, используемых в проекте.

Инструменты для тестирования безопасности API:

– Postman – популярный инструмент для тестирования и разработки API, который также предоставляет функции для проверки безопасности API;

– OWASP API Security Project – набор инструментов и рекомендаций для тестирования безопасности API.

Это всего лишь некоторые из множества инструментов AppSec, доступных на рынке. Выбор конкретных инструментов зависит от потребностей и требований вашего проекта, а также от предпочтений команды разработки и безопасности.

Допустим, у вас есть веб-приложение и вам необходимо провести сканирование уязвимостей для обнаружения потенциальных проблем безопасности. В этом случае вы можете использовать OWASP ZAP (Zed Attack Proxy), бесплатный инструмент для сканирования уязвимостей.

Процесс использования OWASP ZAP может выглядеть следующим образом:

1) установите OWASP ZAP на свою машину или сервер, где будет выполняться сканирование уязвимостей;

2) запустите OWASP ZAP и настройте его для сканирования вашего веб-приложения. Вы можете указать URL-адрес вашего приложения и другие параметры сканирования в интерфейсе OWASP ZAP;

3) запустите процесс сканирования, нажав кнопку «Сканировать». OWASP ZAP начнет обращаться к вашему веб-приложению, отправлять запросы и анализировать ответы, чтобы обнаружить потенциальные уязвимости;

4) по завершении сканирования вы получите отчет о найденных уязвимостях. OWASP ZAP предоставляет подробные сведения о каждой уязвимости, включая описание, рекомендации по устранению и дополнительную информацию;

5) используйте отчет OWASP ZAP для исправления обнаруженных уязвимостей и улучшения безопасности вашего веб-приложения.

Это лишь один пример использования инструмента AppSec. Конечно, в зависимости от конкретных потребностей и сценариев использования, можно выбрать другие инструменты AppSec и применять их в соответствии с вашими требованиями и целями безопасности.

Контрольные вопросы

1. Какие инструменты используются для автоматизированного анализа безопасности кода?
2. Чем отличаются DAST и SAST?
3. Как работает механизм Content Security Policy (CSP)?
4. Что такое IAST и в чём его преимущество?
5. Какие действия необходимо выполнить перед запуском пентеста?

Тесты для самоконтроля

1. Работающее приложение можно протестировать методом

а) SAST

б) DAST

в) IAST

г) SCA

2. Для выявления ошибок бизнес-логики лучше всего подходит

а) автоматизированное сканирование

б) пентестирование

в) статический анализ

г) динамическое тестирование

3. Для интерактивного тестирования безопасности используется инструмент

а) Burp Suite

б) Dependency-Check

в) Ansible

г) Terraform

4. CSP – это

а) Central Security Protocol

б) Content Security Policy

в) Code Signing Process

г) Client-Server Protection

5. Для анализа состава ПО используется инструмент

а) OWASP ZAP

б) OWASP Dependency-Check

в) Wireshark

г) Nmap

Рекомендуемая литература

1. Бабушкин, В. М. Разработка защищенных программных средств информатизации производственных процессов предприятия : учеб. пособие / В. М. Бабушкин, М. В. Тумбинская. – Москва [и др.] : Инфра-Инженерия, 2024. – 257 с. – URL: znanium.ru/catalog/document?id=451746 (дата обращения: 04.06.2025). – Режим доступа: по подписке. – ISBN 978-5-9729-1618-4.
2. Зиновьева, О. М. Интегрированные системы управления безопасностью : Разработка и аудит : практикум / О. М. Зиновьева, А. М. Меркулова, Н. А. Смирнова. – Москва : МИСиС, 2021. – 84 с. – URL: e.lanbook.com/book/238376 (дата обращения: 04.06.2025). – Режим доступа: по подписке.

Тема 4. DEVSECOPS: ИНТЕГРАЦИЯ БЕЗОПАСНОСТИ В СОВРЕМЕННЫЕ МЕТОДЫ РАЗРАБОТКИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

Форма проведения занятия – лекция.

Оборудование к занятию: проектор, ноутбук.

Цель занятия – представление понятия DevSecOps и объяснение принципов интеграции безопасности в разработку ПО. Рассмотрим основные аспекты DevSecOps, его преимущества и роли, а также узнаем о методах и инструментах, которые помогают обеспечить безопасность при разработке ПО.

Вопросы для обсуждения

1. Что такое DevSecOps и каковы его основные принципы.
2. Методы и практики, используемые в DevSecOps.
3. Обеспечение безопасности на различных этапах SDLC.

Методические указания по проведению занятия

При освоении темы необходимо:

1) изучить учебный материал по теме 4;
2) акцентировать внимание на основополагающих понятиях и определениях;
3) ответить на контрольные вопросы по теме 4;
4) выполнить тест по теме 4.

Методические материалы к занятию

1. Что такое DevSecOps и каковы его основные принципы

DevSecOps – подход, который помогает интегрировать безопасность в процесс разработки и получить более надежное и безопасное ПО. Это методология, которая объединяет разработку (Dev), операции (Ops) и безопасность (Sec). Её целью является интеграция безопасности на всех этапах разработки и эксплуатации ПО.

DevSecOps является эволюцией DevOps – подхода, который стремится улучшить сотрудничество между разработчиками и операционными командами. DevSecOps вносит дополнительный аспект

безопасности в этот процесс, обеспечивая безопасную разработку, развертывание и эксплуатацию ПО.

Методология DevSecOps основана на трех ***основных принципах***.

1. *Автоматизация безопасности*. Это ключевой аспект DevSecOps. Она позволяет:

1) автоматизировать процессы проверки безопасности, тестирования и мониторинга, что помогает обнаруживать и устранять уязвимости на ранних этапах разработки;
2) снизить риск человеческого фактора и обеспечить постоянную проверку безопасности при каждом изменении кода или конфигурации.

Принципы Continuous Integration (CI) и Continuous Deployment (CD) в DevSecOps позволяют автоматизировать процессы сборки, тестирования и развертывания ПО, включая проверку безопасности, что позволяет быстро выявлять и устранять уязвимости.

2. *Раннее включение безопасности (Shift Left)* заключается в том, чтобы внедрять безопасность на ранних этапах разработки, а не оставлять ее до последних стадий. Включение безопасности с самого начала позволяет выявлять и исправлять уязвимости на ранней стадии, что экономит время и ресурсы в долгосрочной перспективе.

Разработчики играют важную роль в обеспечении безопасности. Они должны быть обучены безопасному программированию и использованию инструментов и методов, которые помогают предотвратить уязвимости.

3. *Культура безопасности*. Цифровая трансформация стала объективной необходимостью для большинства современных организаций. Её реализация основана на трёх взаимосвязанных компонентах: росте роли ПО как ключевого элемента бизнес-процессов, широком внедрении облачных технологий и переходе к гибким методологиям разработки и эксплуатации, известным как DevOps.

По мере увеличения масштабов и сложности программных систем возрастает и их уязвимость. Принятие решений, направленных на сокращение сроков разработки в ущерб качеству проектирования и кода, приводит к росту скрытых дефектов и сложности системы,

что делает проблемной её поддержку и создаёт условия для возникновения уязвимостей, затрудняя защиту цифровых активов.

Облачные среды кардинально меняют подходы к обеспечению безопасности. Традиционная модель защищённого периметра утрачивает актуальность, уступая место динамичной, программно-определяемой инфраструктуре. Хотя облачные провайдеры обеспечивают безопасность физической и виртуальной инфраструктуры, ответственность за конфигурацию, управление доступом и защиту данных полностью лежит на заказчике. В этих условиях корректная настройка прав доступа и строгое соблюдение принципа наименьших привилегий становятся центральными элементами стратегии информационной безопасности (ИБ).

Методология DevOps основана на сокращении времени ввода ПО в эксплуатацию и подразумевает передачу операционной ответственности непосредственно командам разработки. В соответствии с принципом «ты его написал – ты его запускаешь» разработчики отвечают за все этапы жизненного цикла приложения – от проектирования до его безопасной эксплуатации. В такой модели традиционные ручные и централизованные процедуры проверки безопасности теряют свою эффективность и препятствуют достижению необходимой скорости и гибкости процессов.

Традиционные подходы к обеспечению безопасности приложений, ориентированные на финальный аудит и ручное тестирование, не соответствуют темпам современной разработки. При этом функция ИБ, зачастую ограниченная в ресурсах и кадрах, не успевает за объёмом изменений. Это приводит к тому, что уязвимости проникают в эксплуатацию, а безопасность воспринимается бизнесом как препятствие для инноваций.

Для решения этой проблемы была разработана концепция DevSecOps – интеграция практик ИБ непосредственно в процессы DevOps. Её суть заключается в том, чтобы сделать безопасность неотъемлемой частью повседневной работы команд разработки и эксплуатации. Современные специалисты по кибербезопасности не навязывают контроль сверху, а помогают разработчикам работать безопасно, предоставляя им готовые решения: автоматизированные проверки, безопасные шаблоны кода, стандартизиро-

ванные практики, сканеры, шаблоны, чек-листы, платформы. Это позволяет командам быстро выпускать продукты без ущерба для защиты, сохраняя высокий темп разработки, но при этом соблюдая нормативные и бизнес-требования к безопасности

Модель DevSecOps имеет шесть преимуществ:

1) более быстрая доставка. Скорость доставки ПО повышается, когда безопасность интегрирована в конвейер. Ошибки выявляются и исправляются перед развертыванием, что позволяет разработчикам сосредоточиться на поставке функций;

2) улучшенное состояние безопасности. Модель общей ответственности обеспечивает тесную интеграцию безопасности – от создания, развертывания до защиты производственных рабочих нагрузок;

3) снижение затрат. Выявление уязвимостей и ошибок перед развертыванием приводит к экспоненциальному снижению рисков и эксплуатационных расходов;

4) повышение ценности DevOps. Повышение общего состояния безопасности как культуры общей ответственности создается за счет интеграции методов обеспечения безопасности в DevOps;

5) повышение темпов. Время безопасной доставки ПО сокращается за счет устранения необходимости модернизации средств контроля безопасности после разработки;

6) обеспечение большего общего успеха в бизнесе. Повышенное доверие к безопасности разработанного ПО и внедрение новых технологий обеспечивают ускоренный рост доходов и расширение бизнес-предложений.

Внедрение DevSecOps: интеграция безопасности в конвейер CI/CD

Организации, чья разработка построена на принципах DevOps, повсеместно используют конвейеры непрерывной интеграции и доставки (CI/CD). Эти конвейеры служат идеальной основой для внедрения автоматизированных проверок безопасности, что позволяет исключить необходимость ручного вмешательства на каждом этапе.

Интеграция безопасности начинается на самых ранних стадиях жизненного цикла разработки – до написания первой строки кода. Уже на этапе проектирования архитектуры системы, при-

ложения или пользовательского сценария возможны моделирование угроз и анализ рисков. Статический анализ безопасности (SAST), линтеры кода и механизмы политик (policy-as-code) могут запускаться автоматически при каждом коммите, что позволяет выявлять и устранять простые уязвимости до их попадания в основной репозиторий.

Анализ зависимостей (SCA) применяется комплексно для проверки сторонних библиотек с открытым исходным кодом на наличие известных уязвимостей и на соответствие лицензионным требованиям. Побочным, но важным эффектом этого подхода является формирование у разработчиков ответственности за безопасность, поскольку они получают немедленную обратную связь о качестве написанного кода.

После коммита и сборки кода запускаются динамические тесты безопасности (DAST). Выполнение кода в изолированных средах – контейнерах – позволяет автоматически проверять сетевые вызовы, валидацию ввода и механизмы авторизации. Эти тесты обеспечивают быструю обратную связь, позволяя оперативно исправлять обнаруженные проблемы с минимальным воздействием на общий рабочий процесс. При обнаружении аномалий, таких как неавторизованные сетевые запросы или некорректная обработка ввода, конвейер останавливается, генерируя детальные отчёты и уведомления для ответственных команд.

После успешного прохождения начальных этапов тестирования собранное приложение размещается в изолированной среде тестирования, которая максимально приближена к производственной среде. На этой стадии выполняются расширенные интеграционные тесты безопасности, направленные на выявление уязвимостей, которые могут проявиться только в условиях, имитирующих реальную эксплуатацию.

После того как артефакт развертывания проходит первую батарею интеграционных тестов, он переходит к следующему этапу интеграционного тестирования. Теперь он будет развернут в более широкой песочнице, ограниченной копии возможной производственной среды. На этом этапе может быть выполнено дальнейшее тестирование интеграции безопасности, хотя и с другой целью.

Теперь можно протестировать правильное ведение журнала и контроль доступа. Правильно ли приложение регистрирует соответствующие метрики безопасности и производительности? Ограничен ли доступ к нужному подмножеству лиц (или полностью запрещен)? Неудача требует действий соответствующих команд.

Наконец, приложение попадает в производство. Тем не менее работа DevSecOps продолжается. Автоматизированное управление исправлениями и конфигурацией гарантирует, что в производственной среде всегда будут работать последние и наиболее безопасные версии зависимостей ПО. В идеале неизменяемая инфраструктура означает, что вся среда часто сносится и перестраивается, постоянно подвергаясь тестированию.

Использование конвейера CI/CD DevSecOps помогает интегрировать цели безопасности на каждом этапе, не обременяя бюрократией и контролем, что позволяет поддерживать быструю доставку бизнес-продукта.

2. Методы и практики, используемые в DevSecOps

DevSecOps – это концепция интеграции аспектов ИБ в культуру и процессы DevOps. Цель DevSecOps – внедрить практики безопасной разработки ПО и защиты инфраструктуры на протяжении всего жизненного цикла, от проектирования до эксплуатации.

Основные преимущества DevSecOps:

– снижение рисков за счет раннего обнаружения уязвимостей;
– ускорение вывода обновлений безопасности за счет автоматизации;
– повышение уровня безопасности благодаря вовлеченности всех участников.

Методы обеспечения безопасности в DevSecOps

1. Безопасное проектирование – архитектура и дизайн системы учитывают необходимые механизмы безопасности.

2. SAST – статический анализ исходного кода на наличие уязвимостей.

3. DAST – динамический анализ работающих приложений и поиск уязвимостей в runtime.

4. Тестирование на проникновение – этичный хакинг системы для поиска слабых мест.

5. Сканирование инфраструктуры – поиск уязвимостей в сетях, операционных системах, ПО серверов и рабочих станций.

6. Мониторинг безопасности – анализ логов, обнаружение вредоносной активности.

Основными инструментами DevSecOps являются:

1) CI/CD – Jenkins, TeamCity, CircleCI, TravisCI, GitLab CI/CD;
2) SAST – SonarQube, Checkmarx, Veracode, Fortify;
3) DAST – Burp Suite, OWASP ZAP, Netsparker, Acunetix;
4) сканеры уязвимостей – Nessus, Qualys, OpenVAS;
5) системы обнаружения вторжений – Snort, Suricata, Wazuh;
6) мониторинг – ELK, Splunk, Datadog.

Передовые практики внедрения

Рассмотрим подробнее практики обеспечения безопасности в DevSecOps.

1. Безопасная разработка кода – использование проверенных библиотек, валидация ввода, защита от инъекций, криптография, обработка ошибок и исключений.

2. Статический анализ кода (SAST) – автоматизированный поиск уязвимостей в исходном коде с помощью инструментов вроде SonarQube, Fortify, Checkmarx. Выполняется как часть CI/CD.

3. Динамический анализ (DAST) – сканирование работающего приложения на предмет уязвимостей. Может включать fuzzing – подачу некорректных данных на вход приложения.

4. Тестирование на проникновение – этичный хакинг системы специалистами по ИБ для поиска уязвимостей в реальных условиях.

5. Сканирование зависимостей – проверка используемых библиотек и компонентов на наличие уязвимостей с помощью Snyk, Whitesource, OWASP Dependency Check.

6. Конфигурация как код – хранение конфигураций инфраструктуры в виде кода (Ansible, Terraform, Chef) для возможности анализа и версионирования.

7. Сегментация сети и минимизация доступа на основе принципа наименьших привилегий.

8. Мониторинг и анализ логов безопасности – выявление аномалий и следов атак.

DevSecOps – важная концепция для построения надежных и защищенных систем. Интеграция практик ИБ в процессы DevOps позволяет минимизировать риски и ускорить реагирование на инциденты. Ключевыми факторами успеха DevSecOps являются автоматизация контролей безопасности и тесное взаимодействие команд разработки и ИБ.

3. Обеспечение безопасности на различных этапах SDLC

Традиционно безопасность рассматривается разработчиками как препятствие для инноваций и творчества, которое создает задержки в выводе продукта на рынок. Такой подход может негативно сказаться на прибыли бизнеса, поскольку исправление ошибок на этапе внедрения обычно обходится значительно дороже, чем на более ранних стадиях разработки. Согласно исследованиям, стоимость устранения дефектов в процессе эксплуатации системы может превышать затраты на их предотвращение в ходе проектирования и тестирования в шесть и более раз. Поэтому, откладывая решение проблем качества и безопасности ПО на поздние этапы, компании рискуют столкнуться с большими расходами в будущем.

1. Этапы разработки безопасного ПО

В рамках изучения данного вопроса мы рассмотрим проблемы обеспечения безопасности на различных этапах SDLC (рис. 2).

Рис. 2. Жизненный цикл разработки ПО (SDLC)

Этап I. Анализ и сопоставление требований к безопасности.

Здесь необходимо определить, какие аспекты защиты информации должны быть реализованы в проектируемой системе.

Задачи на этом этапе:

1) изучение бизнес-требований заказчика к конфиденциальности, целостности и доступности информации;
2) анализ нормативных требований и стандартов безопасности для данной предметной области;
3) выявление уязвимостей и угроз безопасности, актуальных для архитектуры системы;
4) разработка комплекса организационных и технических мер для удовлетворения выявленных требований к безопасности.

Тщательный анализ и сопоставление требований позволят сформировать эффективный план обеспечения безопасности разрабатываемого ПО.

Разберем процесс анализа и сопоставления требований к безопасности задач на примере проектирования веб-приложения электронной коммерции:

1. Сбор исходных данных:

– требование бизнеса: защита персональных данных клиентов согласно GDPR;
– нормативное требование: шифрование трафика по протоколу TLS;
– анализ угроз: возможность SQL-инъекций, межсайтового скриптинга.

2. Формализация требований:

– конфиденциальность персональных данных;
– защита трафика криптографией;
– предотвращение внедрения кода.

3. Сопоставление требований:

– TLS удовлетворяет требованиям GDPR по шифрованию;
– необходимы дополнительные меры против SQL-инъекций и XSS.

4. Разработка мер безопасности:

– применение TLS для защиты трафика;
– валидация и фильтрация ввода на стороне сервера;
– фильтрация вывода клиенту.

5. Утверждение плана мероприятий по обеспечению безопасности.

Таким образом, требования проанализированы и определен комплекс мер для защиты веб-приложения от типовых угроз.

Этап II. Проектирование системы с учетом требований безопасности. На этом этапе необходимо использование безопасных языков программирования, следование рекомендациям по безопасности и предотвращение потенциальных уязвимостей. Там, где это уместно, следует использовать безопасные методы кодирования: проверку ввода, проверку подлинности, шифрование и управление доступом.

На этапе проектирования необходимо реализовать меры безопасности, определенные в результате анализа требований. Это включает:

1) разработку безопасной архитектуры – модульность, минимизацию внешних интерфейсов, разделение среды исполнения;
2) выбор надежных криптографических алгоритмов и протоколов;
3) проектирование механизмов аутентификации и авторизации пользователей;
4) реализацию принципа наименьших привилегий для каждого компонента;
5) применение шифрования данных в хранилищах и каналах связи;
6) разработку подсистемы регистрации и аудита событий безопасности;
7) проектирование механизмов восстановления и резервного копирования;
8) разработку процесса управления уязвимостями и исправлениями.

Разберем пример реализации второго шага – проектирования с учетом безопасности – для веб-приложения электронной коммерции.

1. Разработка безопасной архитектуры:

– веб-приложение разделено на модули – frontend, backend, БД;
– backend предоставляет ограниченный набор API для frontend;
– доступ к БД только через ORM framework из backend.

2. Выбор криптографических средств:
– шифрование трафика между фронтендом и бэкендом по TLS;
– хранение паролей пользователей в виде хешей с солью.

3. Аутентификация и авторизация:
– аутентификация через JWT (JSON Web Token) для сессий пользователей;
– ролевая модель авторизации для доступа к API.

4. Минимизация привилегий:
– разграничение доступа к данным через настройки ORM;
– ограничение функционала API ролями пользователей.

5. Шифрование данных:
– шифрование персональных данных клиентов в БД;
– передача конфиденциальных данных по зашифрованным каналам.

Таким образом на этапе проектирования реализуются необходимые меры безопасности в соответствии с требованиями.

Этап III. Безопасная разработка кода.

На этапе реализации системы необходимо придерживаться следующих практик безопасного программирования:
1) использование проверенных библиотек и фреймворков;
2) валидация и санитизация ввода от внешних источников;
3) проверка границ массивов и строковых переменных;
4) использование проверенных криптографических алгоритмов;
5) отсутствие жестко запрограммированных учетных данных в коде;
6) обработка и логирование всех возможных ошибок и исключений;
7) отсутствие утечек конфиденциальных данных в логах и ответах;
8) шифрование конфиденциальных данных перед записью в БД;
9) минимизация привилегий для всех операций системы.

Соблюдение рекомендаций и стандартов безопасной разработки позволит существенно снизить количество уязвимостей в коде приложения.

Разберем пример реализации практик безопасного программирования на этапе разработки кода для веб-приложения электронной коммерции.

1. Валидация ввода на стороне сервера:

– для строковых параметров – проверка длины и формата регулярными выражениями;
– фильтрация опасных символов из строковых значений (скриптов, SQL-кода);
– проверка числовых значений на соответствие допустимым диапазонам.

2. Защита от атак на переполнение буфера:

– проверка в циклах обработки массивов: индекс не должен выходить за границы массива;
– обработка исключений при попытке обращения за границы массива.

3. Хранение паролей пользователей:

– пароли хранятся в виде хешей с солью, а не в открытом виде;
– для хеширования используется алгоритм bcrypt с рекомендуемыми параметрами.

4. Шифрование конфиденциальных данных:

– персональные данные клиентов шифруются перед записью в БД;
– используются надежные алгоритмы шифрования данных, такие как AES-256.

5. Обработка ошибок:

– перехват ошибок и исключений на всех уровнях приложения;
– в обработчиках ошибок не происходит утечки технических деталей.

6. Минимизация привилегий:

– невозможность запуска процессов приложения от имени непривилегированных пользователей;
– установка ограничений на чтение/запись файлов и БД.

Этап IV. Тестирование на проникновение и анализ уязвимостей.

На данном этапе проводятся следующие мероприятия:

1) сканирование кода на наличие уязвимостей с помощью SAST-инструментов;
2) анализ зависимостей приложения на предмет устаревших библиотек с известными уязвимостями;

3) функциональное тестирование критически важных механизмов безопасности;
4) тестирование производительности на предмет устойчивости к DoS-атакам;
5) тестирование на проникновение силами специализированных команд или аудиторов;
6) анализ трафика приложения на предмет ненадежных протоколов и уязвимых конфигураций;
7) сканирование инфраструктуры на предмет открытых удаленных уязвимостей.

Выявление и устранение недостатков безопасности на этапе тестирования позволяет предотвратить инциденты на стадии эксплуатации ПО.

Разберём пример реализации тестирования на проникновение для веб-приложения электронной коммерции.

1. SAST-сканирование кода:
– сканирование исходного кода приложения на наличие уязвимостей с помощью инструментов, таких как SonarQube;
– анализ отчетов и устранение обнаруженных проблем.

2. Анализ зависимостей:
– проверка актуальности версий библиотек и фреймворков приложения;
– обновление компонентов с известными уязвимостями.

3. Функциональное тестирование:
– тестирование корректности работы механизмов аутентификации, авторизации и управления сессиями;
– проверка алгоритмов шифрования и хеширования данных.

4. Тест на производительность:
– нагрузочное тестирование на предмет устойчивости к DDoS;
– проверка горизонтального масштабирования.

5. Тестирование на проникновение:
– аудит безопасности сторонними экспертами, имитация атак;
– анализ отчетов, устранение найденных уязвимостей.

6. Анализ трафика и конфигураций:
– проверка надежности используемых протоколов и параметров безопасности;

– анализ на предмет уязвимостей, связанных с неправильной конфигурацией.

При разработке ПО важно следовать рекомендациям по безопасному кодированию. Эти методы гарантируют, что код безопасен, устойчив к атакам и соответствует требованиям безопасности.

2. Основные принципы разработки безопасного ПО

Разработка безопасного ПО требует комплексного подхода, включающего архитектурное проектирование системы с учетом безопасности, использование надежных библиотек и фреймворков, применение практик безопасного кодирования, тестирование на проникновение и анализ уязвимостей.

Основные принципы разработки безопасного ПО:

1) минимизация поверхности атаки системы;
2) разделение прав и ограничение доступа;
3) валидация и санитизация вводимых данных;
4) управление исключениями и обработка ошибок;
5) криптографическая защита данных;
6) логирование и мониторинг безопасности.

Первый принцип – минимизация поверхности атаки системы – это сведение к минимуму количества компонентов системы, доступных для внешних запросов и потенциально подверженных атакам.

К сожалению, сегодня организации все чаще не могут определить истинный размер и сложность своей поверхности атаки, оставляя свои цифровые и физические активы беззащитными перед субъектами угроз.

На базовом уровне поверхность атаки можно определить как физические и цифровые активы, которыми владеет организация и которые могут быть скомпрометированы для облегчения кибератаки. Конечной целью злоумышленников, стоящих за этим, может быть что угодно – от развертывания программ-вымогателей и кражи данных до включения машин в ботнет, загрузки банковских троянов или установки вредоносных программ для крипто-майнинга. Суть в следующем: чем больше поверхность атаки, тем крупнее цель преступников.

Поверхность атаки делится на две основные категории – поверхность цифровой атаки и поверхность физической атаки.

Поверхность цифровой атаки

Здесь описывается все оборудование, программное обеспечение и компоненты организации, подключенные к сети. К ним относятся:

– приложения: уязвимости в приложениях являются обычным явлением и могут предоставить злоумышленникам полезную точку входа в критически важные ИТ-системы и данные;

– код: большая его часть компилируется из сторонних компонентов, которые могут содержать вредоносные программы или уязвимости;

– порты: злоумышленники всё чаще сканируют открытые порты и проверяют, прослушивают ли какие-либо сервисы определенный порт (например, TCP-порт 3389 для RDP). Если эти сервисы настроены неправильно или содержат ошибки, ими можно воспользоваться;

– серверы, которые могут быть атакованы с помощью уязвимостей или переполнены трафиком при DDoS-атаках;

– веб-сайты, подвергающиеся нескольким векторам атаки, включая недостатки кода и неправильную конфигурацию. Удавшаяся компрометация может привести к повреждению веб-сайта или внедрению вредоносного кода для мимоходных и других атак (то есть formjacking);

– сертификаты с истекшим сроком действия.

В исследовании 2022 года, посвященном фирмам из списка FTSE 30, содержатся ключевые выводы о масштабах цифровых атак на крупные компании:

– в среднем компании FTSE 30 сталкиваются с 390 000 кибератак в неделю – это огромное количество атак, направленных на крупный бизнес;
– 91 % этих атак приходится на автоматизированные инструменты и ботнеты – злоумышленники активно используют автоматизацию для массовых атак;

– наиболее распространены атаки методом подбора паролей, сканирования уязвимостей, DDoS-атаки и вредоносные вложения в почту;
– ущерб от успешных атак оценивается в среднем в 3,4 млн фунтов стерлингов для одной компании;
– время обнаружения и реагирования на инциденты в среднем составляет 146 дней (слишком долго);
– только 28 % компаний считают свою киберзащиту достаточно зрелой.

Эти данные свидетельствуют, что крупный бизнес сталкивается с огромным количеством изощренных атак и компаниям нужно значительно повышать уровень своей кибербезопасности, чтобы противостоять этим угрозам.

Поверхность физической атаки

Сюда входят все конечные устройства (рабочие места), к которым злоумышленник может получить физический доступ, например: настольные компьютеры, жесткие диски, ноутбуки, мобильные телефоны/устройства, флэш-накопители.

Сотрудники являются основной стороной поверхности физической атаки организации, поскольку ими можно манипулировать с помощью социальной инженерии (фишинг и его варианты) в ходе кибератаки. Они также несут ответственность за теневые ИТ, несанкционированное использование приложений и устройств в обход корпоративных мер безопасности. Используя эти неутвержденные – и часто недостаточно защищенные – инструменты для работы, они могут подвергнуть организацию дополнительным угрозам.

Расширение поверхности атак в условиях цифровой трансформации

Современная ИТ-инфраструктура переживает фундаментальную трансформацию: периметр корпоративной сети, ранее чётко очерченный брандмауэрами и контролируемыми точками входа, де-факто упразднён. Это следствие глобального перехода к гибридным и удалённым моделям работы, массовой миграции в облака, автоматизации жизненного цикла ПО (DevOps/CI/CD) и повсе-

местного внедрения IoT-устройств, ускоренного пандемией как катализатором цифровизации.

В результате поверхность атак – совокупность всех активов, интерфейсов, протоколов и данных, потенциально доступных злоумышленнику – не просто увеличилась количественно, но и качественно усложнилась. К числу ключевых векторов её расширения относятся:

1) наличие децентрализованных конечных точек, таких как персональные устройства сотрудников, функционирующие вне корпоративного доверенного периметра и часто не подпадающие под единые политики безопасности;

2) облачные среды (IaaS, PaaS, SaaS), где ответственность за безопасность распределена (shared responsibility model), а ошибки конфигурации становятся одним из главных источников инцидентов;

3) инфраструктура удалённого доступа (VPN, RDP, Zero Trust-платформы), превратившаяся в критически важную, но уязвимую точку соприкосновения пользователей и корпоративных ресурсов;

4) цепочки поставок ПО, включающие open-source-компоненты, сторонние библиотеки и автоматизированные пайплайны развертывания – каналы, через которые уязвимости и вредоносный код могут проникать на ранних этапах разработки;

5) киберфизические системы и 5G-инфраструктура, где каждое подключённое устройство – потенциальная точка компрометации.

Эта трансформация создаёт новые условия для реализации угроз:

– массовые кампании социальной инженерии (в том числе фишинг), эксплуатирующие человеческий фактор в распределённой среде;

– целенаправленную эксплуатацию уязвимостей в API, контейнерах и микросервисах;

– компрометацию учётных данных – включая кражу сессий, использование слабых или повторяющихся паролей, утечку TLS-сертификатов;

– некорректную настройку облачных ресурсов, ведущую к публичной экспозиции данных (например, открытые хранилища AWS S3);

– атаки на программные цепочки поставок (software supply chain attacks), как это было продемонстрировано в инцидентах типа SolarWinds.

Таким образом, цифровая зрелость организаций сегодня напрямую коррелирует не с сокращением рисков, а с увеличением сложности и динамичности угрозы. Это требует перехода от реактивной защиты к проактивной, встроенной в архитектуру и процессы – от «безопасности как контроля» к «безопасности как услуге».

Поверхность атаки имеет принципиальное значение для наилучшей практики кибербезопасности, поскольку понимание ее размера и принятие мер по ее сокращению или управлению ею является первым шагом на пути к упреждающей защите. Размер поверхности атаки определяется с помощью аудита активов и запасов, теста на проникновения, сканирования уязвимостей и многого другого.

Аудит активов и инвентаризация – составление полного перечня аппаратных и программных активов организации, которые требуют защиты. Помогает выявить все потенциально уязвимые компоненты.

Сканирование уязвимостей – использование автоматизированных инструментов, таких как Nessus, OpenVAS, Qualys, которые сканируют сети, серверы, ПО на предмет известных уязвимостей.

Тестирование на проникновение – этичный хакинг – позволяет найти уязвимости путем имитации действий злоумышленников. Может проводиться как вручную, так и с помощью инструментов (Metasploit, Burp и др.).

Анализ конфигураций – проверка безопасности настроек сетевых компонентов, серверов, ПО. Помогает выявить ошибки конфигурации, создающие уязвимости.

Анализ исходного кода (SAST) – статический анализ кода на наличие уязвимостей с помощью инструментов (Fortify, Checkmarx, SonarQube и др.).

Регулярное применение этих методов позволяет контролировать состояние защищенности компании и своевременно устранять уязвимости.

Корпоративная ИТ-среда находится в постоянном движении благодаря широкому использованию виртуальных машин, контей-

неров и микросервисов, постоянному прибытию и отъезду сотрудников, а также новому оборудованию и ПО. Это означает, что любые попытки управлять поверхностью атаки и понимать её должны предприниматься с помощью гибких интеллектуальных инструментов, работающих на основе данных в режиме реального времени, то есть наблюдения и контроля.

Следование перечисленным принципам и лучшим практикам позволяет существенно повысить защищенность разрабатываемых программных комплексов от атак и несанкционированного доступа.

Рекомендации по реализации:

1) выделить отдельный уровень внешних сервисов и API;
2) скрыть внутренние структуры данных и потоки выполнения;
3) использовать шаблон «Фасад» для внешних компонентов;
4) разрешить доступ только к необходимому функционалу;
5) применять валидацию данных на входе во внешние компоненты.

Второй принцип – разделение прав и ограничение доступа – направлен на предотвращение несанкционированного доступа к системе или данным, а также на минимизацию воздействия на систему в случае возникновения уязвимостей или атак.

Для понимания принципа разделения прав и ограничения доступа рассмотрим его применение на различных уровнях ПО.

На уровне ***операционной системы*** (ОС) разделение прав и ограничение доступа осуществляются путем использования механизмов аутентификации, авторизации и управления доступом. Аутентификация позволяет проверить подлинность пользователя, а авторизация определяет права доступа пользователя к системным ресурсам. Управление доступом определяет, какие ресурсы доступны для каких пользователей или групп пользователей. ОС также может использовать механизмы контроля доступа, такие как права доступа к файлам и каталогам, разрешения на выполнение программ и т. д., чтобы обеспечить ограничение доступа к системным ресурсам.

При разработке ***сетевых сервисов*** важно разделять права и ограничивать доступ к различным функциям и данным. Например, веб-приложение может иметь различные роли пользователей, каждая из которых имеет определенные права доступа к функциональности и данным приложения. Разделение прав и ограничение

доступа в сетевых сервисах достигается путем реализации механизмов аутентификации, авторизации и контроля доступа на уровне приложения.

Для обеспечения безопасности данных используется разделение прав и ограничение доступа на уровне ***базы данных***. БД могут предоставлять механизмы управления пользователями и ролями, которые определяют, какие данные могут быть просмотрены, изменены или удалены различными пользователями. Ограничение доступа в БД также может быть реализовано с помощью механизмов шифрования данных и механизмов аудита, чтобы отслеживать и контролировать доступ к данным.

При разработке ***приложений*** необходимо учитывать принцип наименьших привилегий, то есть предоставлять только необходимые права доступа для выполнения определенных функций. Это может быть достигнуто путем реализации принципа «отделения обязанностей» и применения привилегий на уровне кода, таких как принципы ограничения доступа к ресурсам операционной системы и БД.

В целом, разделение прав и ограничение доступа в ПО включает использование механизмов аутентификации, авторизации, управления доступом, контроля доступа и шифрования данных. Эти механизмы помогают предотвратить несанкционированный доступ, уменьшить воздействие атак и обеспечить конфиденциальность, целостность и доступность системы и данных. Правильная реализация и соблюдение этих принципов помогают создать безопасное ПО, устойчивое к различным угрозам и атакам.

Третий принцип – валидация и санитизация вводимых данных – направлен на предотвращение атак, связанных с обработкой некорректных, вредоносных или злоумышленных данных, которые могут привести к уязвимостям или нарушению безопасности системы. Валидация и санитизация данных в ПО выполняются на различных уровнях и в различных компонентах системы.

Одна из основных уязвимостей ПО – недостаточная валидация и санитизация ***входных данных***, поступающих от пользователей. Принцип валидации и санитизации данных подразумевает проверку и очистку вводимых пользователем данных, чтобы убедиться в их

корректности, целостности и безопасности. Это включает проверку наличия обязательных полей, проверку формата данных (например, адрес электронной почты или номер телефона), фильтрацию нежелательных символов (например, HTML-теги или SQL-инъекции), а также ограничение размеров данных для предотвращения переполнения буфера или других атак.

Следует предусмотреть безопасность ***сетевого взаимодействия***. При передаче данных по сети необходимо проверять и фильтровать вводимые значения, чтобы предотвратить атаки, такие как межсетевая инъекция (например, SQL-инъекции или команды операционной системы), межсайтовый сценарий (XSS) или подделка запроса междоменного происхождения (CSRF). Производится валидация форматов данных, используются безопасные протоколы передачи данных (например, HTTPS), а также необходима проверка и фильтрация данных, поступающих от удаленных источников.

При ***сохранении*** данных в БД необходимо проверять и фильтровать входные значения, чтобы предотвратить атаки, такие как SQL-инъекции или внедрение скриптов. При этом нужно использовать параметризованные запросы или подготовленные выражения для предотвращения выполнения вредоносного кода в запросах к БД.

В ходе обработки данных ***внутри системы***, например при работе с файлами, необходимо проверять и фильтровать входные значения, чтобы предотвратить атаки, такие как внедрение исполняемого кода или чтение чувствительных данных. В этом случае валидация и санитизация включают проверку расширений файлов, проверку размеров файлов и использование безопасных методов обработки файлов.

При разработке ПО следует учитывать следующие меры для валидации и санитизации данных:

– проверку формата (соответствия вводимых данных определенному формату). Например, если пользователь должен ввести адрес электронной почты, то необходимо проверить, что введенное значение содержит символ «@» и имеет правильную структуру электронной почты;

– фильтрацию нежелательных символов. Потенциально опасные или нежелательные для обработки символы удаляют или экранируют (фильтрация HTML-тегов, чтобы предотвратить межсайтовые сценарии (XSS), или экранирование символов, которые могут быть использованы для инъекций SQL);

– ограничение размера данных для предотвращения переполнения буфера или других видов атак, связанных с переполнением;

– параметризацию запросов: использование параметризованных запросов при взаимодействии с БД предотвращает SQL-инъекции, поскольку параметры запроса отделены от самого запроса и не могут быть интерпретированы как код или команды;

– валидацию на стороне сервера: дополнительная валидация данных на стороне сервера проводится, чтобы убедиться в их корректности и принимать решения на основе безопасных данных. Нельзя полагаться только на валидацию, производимую на стороне клиента, так как она может быть обойдена или изменена злоумышленником;

– использование белых списков (whitelisting): ограничение вводимых данных только до определенных допустимых значений или шаблонов позволяет исключить нежелательные или потенциально опасные данные;

– применение шифрования в случае хранения или передачи чувствительных данных.

Применение этих мер безопасности в различных компонентах ПО, начиная от пользовательского интерфейса и заканчивая БД и серверной обработкой, помогает предотвратить уязвимости, связанные с обработкой некорректных данных. Это важно для защиты системы от атак, основанных на некорректной обработке вводимых данных.

Четвертый принцип – управление исключениями и обработка ошибок для предотвращения уязвимостей, обеспечения надлежащей работы системы и защиты ее от потенциальных атак.

В рамках этого принципа предусмотрены:

– обработка ошибок на всех уровнях системы, начиная от пользовательского интерфейса и заканчивая бэкенд-компонентами, которая позволяет предотвратить неожиданное завершение работы

программы и предоставить пользователю информативные сообщения об ошибках вместо отображения технических деталей, которые могут быть использованы злоумышленниками для анализа и эксплуатации системы;

– защита от информационных утечек: критически важно избегать отображения конфиденциальной информации или подробностей, которые могут быть использованы хакерами для анализа или атаки на систему. Вместо этого следует предоставлять общие и информативные сообщения, которые помогут пользователям понять причину ошибки, но не будут раскрывать чувствительную информацию;

– логирование ошибок, их регистрация, которые являются важным инструментом для отслеживания и анализа проблем в ПО. Грамотное управление журналами ошибок предусматривает логирование только необходимой информации, избегание регистрации конфиденциальных данных, таких как пароли или персональная информация пользователей, а также обеспечение защиты журналов от несанкционированного доступа;

– предотвращение ошибок взлома: необходимо исключить возможность злоумышленников использовать ошибки для получения несанкционированного доступа к системе. Для этого применяется проверка прав доступа, аутентификация и авторизация пользователей, а также принципы обеспечения безопасности – защита от подделки запросов междоменного происхождения (CSRF) и предотвращение межсайтовых сценариев (XSS);

– резервное копирование и восстановление данных на случай возникновения серьезных ошибок или сбоев в системе. Регулярное создание резервных копий и тестирование процедур восстановления помогут минимизировать потерю данных и обеспечить бесперебойную работу системы;

– тестирование и отладка ПО на предмет выявления и исправления ошибок. Регулярное проведение тестирования поможет обнаружить потенциальные уязвимости и ошибки в обработке исключений. Необходимо убедиться, что обработка исключений осуществляется корректно и безопасно.

Для реализации принципа управления исключениями и обработки ошибок ПО можно использовать различные языки программирования и платформы.

Java предоставляет мощные средства для обработки исключений. В языке Java есть ключевые слова try, catch и finally, которые позволяют перехватывать и обрабатывать исключения. Java также предлагает иерархию классов исключений, которую можно использовать для различной обработки ошибок.

C# также обладает обширными средствами для обработки исключений. Он использует блоки try, catch и finally для перехвата и обработки исключений. Язык C# также предоставляет иерархию классов исключений и возможность создания пользовательских исключений.

Python поддерживает обработку исключений с помощью блоков try и except. Он предоставляет различные типы исключений, которые можно перехватывать и обрабатывать, а также возможность создания пользовательских исключений.

В языке C++ исключения обрабатываются с помощью блоков try, catch и throw. Язык предоставляет возможность определить типы исключений и перехватывать их с помощью соответствующих catch-блоков.

Фреймворк .NET, включающий C# и другие языки, предлагает механизмы для обработки исключений, в том числе стандартную библиотеку исключений и классы, такие как Exception. Он также предоставляет возможность создания пользовательских исключений и управления процессом обработки ошибок.

В целом, выбор конкретного языка и платформы зависит от требований проекта, предпочтений разработчиков и характеристик системы, для которой разрабатывается ПО. Важно выбрать язык и платформу, которые обеспечат максимальную гибкость, безопасность и эффективность при обработке исключений и ошибок.

Пятый принцип – криптографическая защита данных – подразумевает использование криптографических методов и алгоритмов для обеспечения конфиденциальности, целостности и аутентификации данных. Криптографическая защита данных играет важную роль в обеспечении безопасности информации в приложениях

для хранения паролей, системах электронной коммерции, мессенджерах и др.

Реализация криптографической защиты данных может использоваться:

– для обеспечения конфиденциальности данных путем их шифрования. Примером может служить использование алгоритма AES (Advanced Encryption Standard) для шифрования конфиденциальной информации, такой как пароли пользователей, данные кредитных карт и другие чувствительные данные. Таким образом, даже если злоумышленник получит доступ к зашифрованным данным, он не сможет прочитать их без соответствующего ключа;

– для обеспечения целостности данных: криптографическая хеш-функция преобразует входные данные в уникальную строку фиксированной длины, называемую хеш-значением. Если данные изменяются, даже незначительно, хеш-значение также изменится. Примером является использование алгоритма SHA-256 для генерации хеш-значений и проверки целостности файлов или сообщений. При получении данных можно вычислить хеш и сравнить его с оригинальным значением, чтобы убедиться, что данные не были изменены;

– для аутентификации и обеспечения подлинности данных. Например, цифровая подпись создается с использованием приватного ключа и позволяет проверить идентичность отправителя данных. При получении данных можно проверить цифровую подпись с использованием публичного ключа отправителя для подтверждения, что данные не были изменены и что они действительно были отправлены ожидаемым отправителем;

– для управления ключами. Ключи шифрования и дешифрования, ключи подписи и ключи аутентификации должны быть сохранены и передаваться в защищенном виде. Ключевое управление предусматривает генерацию безопасных ключей, их хранение в защищенном виде, регулярное обновление и установление строгих политик доступа к ключам;

– для защиты от атак на криптографию. При реализации криптографической защиты данных необходимо учитывать возможные атаки на криптографические алгоритмы и протоколы.

Некорректная реализация криптографии может привести к уязвимостям и компрометации данных. Поэтому важно использовать проверенные и надежные криптографические библиотеки и следить за обновлениями и рекомендациями в области безопасности.

Для реализации криптографической защиты данных могут использоваться различные языки программирования и библиотеки.

В Java доступны классы и методы из пакета javax.crypto. Например, для шифрования данных с использованием алгоритма AES можно использовать классы Cipher и SecretKey из этого пакета.

В C# в пространстве имен System.Security.Cryptography доступны несколько классов и методов. Например, для шифрования данных с использованием алгоритма AES можно использовать классы AesManaged и ICryptoTransform.

В Python доступны библиотеки cryptography и pycryptodome. Например, с использованием библиотеки cryptography можно реализовать шифрование данных с помощью алгоритма AES и генерацию цифровых подписей.

В C++ можно использовать библиотеки OpenSSL или Crypto++. Например, с использованием библиотеки OpenSSL можно реализовать шифрование данных с помощью алгоритма AES и генерацию хеш-значений.

Важно отметить, что реализация криптографической защиты данных требует глубоких знаний в области криптографии и безопасности. Неправильная реализация может привести к уязвимостям и компрометации данных. Поэтому рекомендуется обратиться к документации и руководствам по безопасности для выбранного языка программирования и использовать проверенные и рекомендованные методы и алгоритмы.

Шестой принцип – логирование и мониторинг безопасности – подразумевает создание и поддержку системы логирования, которая записывает события безопасности и другую релевантную информацию о работе приложения.

Логирование и мониторинг безопасности позволяют обнаруживать потенциальные инциденты безопасности, анализировать атаки и незначительные нарушения, а также осуществлять отслеживание действий пользователей и администраторов.

Для реализации принципа логирования и мониторинга безопасности необходимы следующие мероприятия.

Логирование событий. Приложение должно записывать события безопасности, такие как попытки неудачной аутентификации, доступ к защищенным ресурсам, изменение настроек безопасности и другие события, которые могут указывать на потенциальные угрозы. Логи должны содержать достаточно информации для понимания происходящего, включая метки времени, идентификаторы пользователей или аккаунтов, IP-адреса и другую сетевую информацию.

Хранение и защита логов. Система логирования должна храниться в безопасном и надежном месте, чтобы предотвратить несанкционированный доступ и изменение логов злоумышленниками. Логи должны быть защищены от удаления или модификации, чтобы обеспечить целостность данных. Рекомендуется использовать централизованную систему хранения логов, которая обеспечивает централизованный и безопасный доступ к логам.

Мониторинг и анализ логов. Система логирования должна быть интегрирована с механизмом мониторинга и анализа, чтобы обнаруживать потенциальные угрозы и атаки. Можно использовать системы SIEM (Security Information and Event Management) для сбора, анализа и корреляции данных логов. Это позволяет выявлять аномальное поведение, связывать события и создавать предупреждения на основе заданных правил или шаблонов.

Уведомления и реагирование на инциденты. При обнаружении потенциальных угроз или атак система мониторинга должна предоставлять уведомления администраторам или ответственным лицам. Уведомления могут быть отправлены по электронной почте, в виде SMS или другими способами связи. При получении уведомления администраторы должны принять меры для реагирования на инцидент и отслеживать его развитие.

Аудит и проверка соответствия. Логи безопасности могут использоваться для аудита и проверки соответствия нормам безопасности, таким как PCI DSS, HIPAA, GDPR и др. Логи могут быть предоставлены сторонним аудиторам и регуляторам для проверки соответствия требованиям безопасности и обнаружения нарушений.

Для реализации логирования и мониторинга безопасности можно использовать различные инструменты и технологии:

– логирование на уровне приложения: в большинстве современных языков программирования существуют библиотеки и инструменты для регистрации событий и ошибок на уровне приложения. Например, в языке Python популярными инструментами являются logging и loguru. Они позволяют указывать уровни логирования, форматировать и сохранять логи в файлы или БД (рис. 3);

– системы управления журналами (Log Management Systems) – специализированные инструменты, которые помогают собирать, хранить и анализировать логи с нескольких источников. Например, Elasticsearch, Logstash и Kibana (известные вместе как ELK-стек) и Splunk позволяют централизованно собирать и анализировать логи, создавать дашборды и предупреждения;

– системы SIEM (Security Information and Event Management), которые объединяют в себе функциональность сбора, анализа и корреляции данных логов, а также обнаружения и реагирования на угрозы. Они позволяют выявлять аномальное поведение, связывать события и создавать предупреждения на основе заданных правил или шаблонов (например, IBM QRadar, Splunk Enterprise Security и LogRhythm);

– мониторинг сетевой активности, используемый для обнаружения внешних атак и аномальной активности в сети (Network Security Monitoring, NSM). Он позволяет анализировать сетевой трафик, обнаруживать атаки и вторжения, а также регистрировать события безопасности. Примеры NSM-систем: Suricata, Bro и Snort;

– автоматизацию и машинное обучение. Для обнаружения сложных атак и аномалий можно применять методы машинного обучения и алгоритмы анализа данных. Например, можно использовать нейронные сети для обнаружения аномалий в логах или алгоритмы классификации для идентификации подозрительной активности. Такие методы могут помочь автоматизировать процесс обнаружения инцидентов безопасности и реагирования на них.

```
import logging

# Настройка логгера
logging.basicConfig(filename='app.log', level=logging.INFO)

# Запись событий
logging.info('Сообщение информационного уровня')
logging.warning('Предупреждение')
logging.error('Ошибка')
```

Рис. 3. Пример использования библиотеки logging в Python

Важно отметить, что реализация логирования и мониторинга безопасности должна быть адаптирована к конкретным потребностям и особенностям приложения или системы. Следует учитывать требования безопасности, регуляторные нормы и лучшие практики при разработке и настройке системы логирования и мониторинга.

Контрольные вопросы

1. Какие меры безопасности должны быть реализованы на этапе проектирования?
2. Какие задачи выполняются на этапе внедрения (разработки) ПО с точки зрения безопасности?
3. Почему важен контроль конфигураций?
4. Какие действия проводятся на этапе тестирования безопасности?
5. Какие аспекты безопасности нужно учитывать при выпуске и обслуживании ПО?

Тесты для самоконтроля

1. На этапе проектирования нужно учитывать

а) выбор цветовой гаммы

б) моделирование угроз

в) написание юнит-тестов

г) создание презентаций

2. Безопасное управление конфигурацией – это
а) хранение конфигураций в виде кода
б) использование жёстко заданных паролей
в) прямая запись данных в базу
г) использование незащищённых API

3. На этапе тестирования используется
а) ручное написание кода
б) пентестирование
в) установка ОС
г) архивирование файлов

4. При настройке доступа применяется принцип
а) максимальных привилегий
б) минимальных привилегий
в) случайных прав
г) открытого доступа

5. Обновления и исправления следует проводить
а) раз в год
б) по мере выхода новых версий и уязвимостей
в) только при необходимости
г) после каждого запуска программы

Рекомендуемая литература

1. Бабушкин, В. М. Разработка защищенных программных средств информатизации производственных процессов предприятия : учеб. пособие / В. М. Бабушкин, М. В. Тумбинская. – Москва [и др.] : Инфра-Инженерия, 2024. – 257 с. – URL: znanium.ru/catalog/document?id=451746 (дата обращения: 04.06.2025). – Режим доступа: по подписке. – ISBN 978-5-9729-1618-4.
2. Безопасность разработки в Agile-проектах : Обеспечение безопасности в конвейере непрерывной поставки / Л. Белл, М. Брантон-Сполл, Р. Смит, Д. Бэрд ; пер. с англ. А. А. Слинкин. – Москва : ДМК Пресс, 2018. – 448 с. – ISBN 978-5-97060-648-3.

Тема 5. ТЕСТИРОВАНИЕ БЕЗОПАСНОСТИ ПРИЛОЖЕНИЙ: МЕТОДЫ И ИНСТРУМЕНТЫ

Форма проведения занятия – лекция.

Оборудование к занятию: проектор, ноутбук.

Цель занятия – представление различных методов и инструментов, используемых при тестировании безопасности приложений. Будут рассмотрены основные подходы, их преимущества и недостатки, методы и инструменты, помогающие обнаруживать уязвимости и повышать уровень безопасности приложений.

Вопросы для обсуждения

1. Динамическое тестирование безопасности приложений (DAST).
2. Инструменты статического тестирования безопасности приложений (SAST).
3. Пентестирование.
4. Анализ состава ПО (SCA).
5. Интерактивное тестирование безопасности приложений (IAST).

Методические указания по проведению занятия

При освоении темы необходимо:

1) изучить учебный материал по теме 5;
2) акцентировать внимание на основных понятиях и определениях;
3) ответить на контрольные вопросы по теме 5;
4) выполнить тест по теме 5.

Методические материалы к занятию

Основные методы и инструменты обеспечения безопасности приложений (AppSec)

В практике безопасной разработки используется не универсальное средство, а многоуровневая стратегия, в которой каждый инструмент AppSec (Application Security) решает конкретную задачу на определённом этапе жизненного цикла приложения. Эффективная защита достигается за счёт сочетания автоматизации и экспертного анализа, с учётом критичности системы, частоты изменений

и уровня риска. Существуют следующие ключевые методы, которые требуют рационального применения.

1. Статический анализ (SAST – Static Application Security Testing).

SAST сканирует исходный код, байт-код или двоичные файлы без запуска приложения, выявляя потенциальные уязвимости на ранних этапах разработки. Этот подход позволяет «смещать безопасность влево» (shift left), интегрируя проверки прямо в IDE или CI/CD-пайплайн. SAST особенно ценен для выявления типовых ошибок, таких как SQL-инъекции, XSS или небезопасная работа с памятью, и поддерживает культуру ответственности за безопасность у самих разработчиков.

2. Динамический анализ (DAST – Dynamic Application Security Testing).

DAST тестирует запущенное приложение «снаружи», имитируя действия злоумышленника. Он не требует доступа к исходному коду и эффективен при поиске уязвимостей, проявляющихся только в runtime (например, неправильная конфигурация сервера или утечки через API). DAST оптимален для приложений с низким и средним уровнем риска. Для критически важных систем его обязательно дополняют ручным тестированием безопасности веб-интерфейсов, чтобы охватить логические и контекстно-зависимые уязвимости.

3. Интерактивный анализ (IAST – Interactive Application Security Testing).

IAST объединяет преимущества SAST и DAST: инструмент работает внутри запущенного приложения, используя агенты или датчики для мониторинга реального поведения кода при выполнении тестовых сценариев. Это обеспечивает высокую точность (низкий уровень ложных срабатываний) и возможность автоматической верификации уязвимостей в DevOps-конвейерах. IAST особенно эффективен в средах с частыми релизами, где требуется быстрая и достоверная обратная связь.

4. Анализ состава ПО (SCA – Software Composition Analysis).

SCA фокусируется на рисках, связанных со сторонними компонентами: open-source-библиотеками, фреймворками и контейнерными образами. Инструмент сканирует зависимости на наличие известных уязвимостей (CVE), проверяет лицензионную совместимость и помогает управлять «техническим долгом безопасности».

В современных приложениях доля стороннего кода – включая open-source-библиотеки и зависимости – может достигать 90 %, поэтому Software Composition Analysis (SCA) стал неотъемлемым элементом любой зрелой AppSec-стратегии.

5. Пенетрационное тестирование (Penetration Testing).

В отличие от автоматизированных методов, пентест – это целенаправленная имитация атаки, выполняемая экспертами. Он фокусируется на бизнес-логике, цепочках эксплуатации и нестандартных векторах, недоступных для сканеров. Пентест незаменим для критически важных систем, особенно при значительных изменениях архитектуры или перед запуском в production. Это «финальная проверка» зрелости защиты.

1. Динамическое тестирование безопасности приложений (DAST)

После того как ПО было полностью разработано, оно должно пройти через ряд сценариев кибератак, с которыми оно может столкнуться при развертывании. Этот процесс тестирования операционного ПО известен как динамическое тестирование безопасности приложений (DAST – Dynamic Application Security Testing).

DAST исследует функциональную устойчивость ПО. При правильном выполнении DAST обнаружит все уязвимости безопасности, которые появляются только при использовании ПО. Это важная практика безопасного кодирования, которая должна быть интегрирована во все жизненные циклы разработки ПО.

Инструменту DAST не требуется никаких сведений о приложении, например о том, какой язык программирования использовался для реализации приложения. Таким образом, можно повысить безопасность своих приложений даже при использовании нишевых языков программирования.

Для оценки функциональности метода тестирования безопасности важно учитывать его возможности и ограничения.

Динамическое тестирование безопасности приложений (DAST) – это метод тестирования черного ящика, который сканирует приложения во время выполнения. Он применяется позже в конвейере CI. DAST является хорошим методом предотвращения регрессий и не зависит от конкретного языка программирования.

DAST, как и интерактивное тестирование безопасности приложений (IAST), фокусируется на поведении приложения во время выполнения. Но анализ IAST, скорее, основан на сочетании тестирования черного ящика, сканирования и анализа внутренних потоков приложений. Преимущество IAST заключается в его способности связывать результаты, подобные DAST, с исходным кодом, таким как SAST. Недостатком IAST является то, что он зависит от языка программирования и может быть выполнен только позже, в конвейере CI.

Анализ состава ПО (Software Composi-tion Analysis, SCA) фокусируется на зависимостях стороннего кода, которые используются в приложении. SCA очень эффективен в приложениях, использующих множество библиотек с открытым исходным кодом. Этот метод также зависит от языка программирования.

DAST лучше всего подходит для методов тестирования безопасности приложений, основанных на статических проверках, таких как SAST и SCA, поскольку он предоставляет дополнительные аналитические сведения во время выполнения для анализа статического исходного кода.

Часто инструменты SAST просматривают только отдельные фрагменты кода и не учитывают их контекст. Код в одном файле можно рассматривать как проблему, даже если код из другого файла, который его использует, решает все проблемы. Безопасность, внедренная в свою систему, может работать даже на совершенно другом компьютере. Например, инструмент SAST идентифицирует неочищенный ввод как проблему, поскольку он не может соотнести его с санитарией, которая происходит на сервере непосредственно перед использованием данных.

Инструменты SAST идут в направлении сквозного тестирования. Они не знают, соответствует ли клиентский код рекомендациям по очистке входных данных. Инструменты DAST будут просматривать входные и выходные данные вашего приложения. Если выход продезинфицирован, им все равно, где конкретно в архитектуре произошла санитария.

Инструменты DAST изучают исходный код и проверяют наличие проблемного поведения, которое останется незамеченным

при использовании только инструмента DAST. Например, преобразование типа может иногда привести к сбою или нежелательным результатам. Тестовый пример DAST, который охватывает только 10 входных данных, может сообщить, что всё в порядке, но инструмент SAST будет знать, что метод преобразования может не работать при определенных значениях, которые вы даже не рассматривали при реализации своего приложения.

Чтобы определить, подходит ли DAST для конкретного программного проекта, нужно оценить преимущества и недостатки его инструментов.

К преимуществам относится:

– меньшее количество ложных срабатываний, поскольку DAST не сканирует всё приложение. Это позволяет гораздо быстрее проверить, реальна ли уязвимость, и посмотреть, можно ли ее предотвратить в дальнейшем;

– независимость от языка программирования, так как DAST не просматривает исходный код, байт-код или ассемблерный код, а просто проверяет входы и выходы системы. Если приложение реализовано с помощью нишевого языка программирования, DAST может быть единственным вариантом;

– быстрое повторное тестирование исправленных уязвимостей. DAST контролирует регрессии. Если уязвимость в системе безопасности обнаружена и воспроизведена, ее можно автоматизировать и добавить в набор тестов DAST. Это означает, что каждый последующий выпуск будет включать взаимодействия, которые привели к прошлым проблемам. Если эти проблемы каким-то образом возвращаются снова, DAST обнаруживает их до того, как они будут выпущены.

К недостаткам инструментов DAST можно отнести:

– отсутствие аналитических сведений о коде: DAST смотрит только на входы и выходы вашей системы. Это делает невозможным соотнесение обнаруженных уязвимостей со строками кода;

– более медленный процесс тестирования. Требование запуска и использования ПО может замедлить процесс тестирования даже при использовании автоматических методов тестирования. Щел-

чок по процессу регистрации с несколькими перестановками ввода просто требует времени;

– обнаружение результатов на поздних этапах конвейера CI/CD: DAST находится в крайнем правом углу конвейера CI/CD, так как выполнение приложения требуется для того, чтобы инструменты DAST выполняли свою работу. Это может занять довольно много времени, особенно если приложение со временем выросло;

– необходимость ручного тестирования. Если по какой-то причине невозможно автоматизировать выполнение и использование приложения, придется полностью удалить DAST из конвейера CI/CD и тестировать приложение вручную для каждого выпуска.

Так как DAST зависит от выполнения приложения, добавить его в конвейер тестирования не так просто, как добавить SAST. Чтобы сделать DAST автоматизированным процессом, часть, которая будет автоматизирована, должна быть сначала записана. После добавления инструмента DAST в конвейер необходимо следовать определенным правилам.

1. Общайтесь с пользователями.

Хорошей отправной точкой для реализации DAST является общение с пользователями и ведение хроники того, как они используют приложение. В дополнение к записи их действий попросите их объяснить, что они делают.

Пользователи, как правило, забывают, на что они на самом деле нажимают в приложении: действия становятся подсознательными. С одной стороны, это помогает пользователям сосредоточиться на своих задачах, с другой стороны, это может привести к проблеме.

2. Автоматизируйте взаимодействие с пользователем.

Следующим шагом является использование инструмента автоматизации для написания сценариев действий пользователя. Это может быть более простым делом для приложений CLI и API, чем для графического интерфейса, но, вообще говоря, это возможно для всех из них.

3. Добавьте тестовые скрипты в конвейер CI/CD.

Когда наиболее важные варианты использования охвачены автоматическим взаимодействием, вы можете запускать эти сценарии для своего приложения, пока инструмент DAST сканирует

его. После первого запуска DAST можно приступить к устранению уязвимостей безопасности.

4. Добавьте регрессионные тесты в набор тестов.

При обнаружении уязвимости безопасности в повседневном использовании вашего приложения вы можете добавить определенные сценарии использования в свой набор тестов. Это гарантирует, что проблемы не вернутся в будущем.

2. Инструменты статического тестирования безопасности приложений (SAST)

Цель каждого разработчика – обеспечить безопасность своего исходного кода. Но разработчики часто не имеют опыта в области безопасности, не знают, как использовать безопасные API или обнаруживать межфайловые проблемы, которые включают несколько частей приложения, разработанного несколькими командами.

Поэтому рекомендуется проводить статическое тестирование безопасности приложений (SAST) как часть общего решения и стратегии безопасности приложений. SAST анализирует исходный код на наличие уязвимостей безопасности.

SAST – это метод сканирования уязвимостей, который фокусируется на исходном коде, байт-коде или ассемблерном коде. Сканер может работать на ранних этапах конвейера CI или даже в качестве подключаемого модуля IDE во время кодирования. Инструменты SAST отслеживают код, обеспечивая защиту от проблем безопасности, таких как сохранение пароля в виде открытого текста или отправка данных по незашифрованному соединению.

Использование SAST на ранней стадии конвейера непрерывной интеграции (CI) или в интегрированной среде разработки (IDE) с помощью подключаемого модуля во время кодирования позволяет инструменту проверять код в режиме реального времени и предотвращать попадание проблем безопасности в кодовую базу.

Комплексный анализ статического кода (SAST) предоставляет разработчикам ряд **преимуществ** в SDLC, которые способствуют улучшению качества кода и снижению затрат и усилий, связанных с обеспечением безопасности приложений:

– обнаружение уязвимостей на ранних этапах: SAST позволяет обнаружить потенциальные уязвимости и слабые места в коде

на ранних этапах разработки, еще до компиляции или выпуска приложения. Это позволяет быстро выявлять проблемы и вносить исправления, что снижает риски и затраты на исправление уязвимостей в более поздних стадиях разработки;

– улучшение качества кода: SAST помогает выявить ошибки, несоответствия стандартам кодирования, потенциально опасные конструкции и другие проблемы, которые могут привести к уязвимостям или неправильному функционированию приложения. Исправление этих проблем приводит к повышению качества кода и уменьшению количества ошибок, что в конечном итоге положительно сказывается на безопасности и надежности приложения;

– сокращение затрат и усилий на обеспечение безопасности: поскольку SAST позволяет обнаруживать уязвимости на ранних этапах разработки, это позволяет снизить затраты и усилия, связанные с обеспечением безопасности приложений. Раннее обнаружение и исправление проблем стоит гораздо меньше, чем позднее внесение изменений в уже разработанный и выпущенный продукт;

– соблюдение стандартов и регуляций: SAST может помочь в обеспечении соблюдения стандартов безопасности и требований регуляций. Инструменты SAST могут проверять код на соответствие набору правил и стандартов, что позволяет разработчикам обнаруживать и исправлять нарушения, связанные с безопасностью, еще на ранней стадии разработки;

– автоматизацию и масштабируемость: SAST-инструменты могут автоматизировать процесс анализа кода и интегрироваться в CI/CD-пайплайны, что позволяет выполнять проверки безопасности на регулярной основе в автоматическом режиме. Это упрощает интеграцию безопасности в процесс разработки и позволяет масштабировать проверки на большие проекты.

В целом, использование SAST-инструментов приводит к повышению качества кода, снижению уязвимостей и общих затрат и усилий на обеспечение безопасности приложений в SDLC. Это важный компонент в создании безопасного и надежного ПО.

При наличии неоспоримых преимуществ статическое тестирование безопасности приложений также имеет **ограничения**:

– ложноположительные и ложноотрицательные результаты: инструменты SAST интерпретируют исходный код и должны применять определенные допущения, следовательно, могут найти проблемы, которые не соответствуют действительности, что называется ложным срабатыванием – ложным результатом. Устаревшие инструменты SAST могут иметь частоту ложных срабатываний от 50 до 80 %, что затрудняет поиск сигнала в шуме, а рентабельность инвестиций в SAST сомнительна, поэтому важно использовать современный SAST с большей точностью;

– отсутствует контекст: неочищенный пользовательский ввод представляет собой огромную угрозу безопасности и должен быть исправлен при каждом входе в программный компонент. Неочищенный ввод во внешнем интерфейсе часто фиксируется на сервере, что снижает риск. Это происходит из-за того, что код фронтенда и бэкенда не всегда находится в одном и том же репозитории, а это означает, что инструмент SAST не обнаружит очистку и не предложит разработчику исправить проблему;

– языковая зависимость: SAST имеет сильную зависимость от кода. Для распространенных языков программирования (например, Java и C#) доступно множество инструментов SAST, но для более нишевых языков (например, ReScript и Nim) инструментов SAST очень мало.

SAST имеет существенные **отличия** от других инструментов Application Security.

Так, DAST подразумевает тестирование приложения во время его выполнения, а не анализ исходного кода. В отличие от SAST, DAST сканирует приложение в рабочем состоянии и ищет уязвимости, основываясь на фактическом взаимодействии с ним. По сравнению с SAST, DAST может обнаружить уязвимости, связанные с конфигурацией, сетевыми настройками и другими аспектами, которые не могут быть выявлены только анализом кода. Однако DAST не может обнаружить уязвимости, связанные с ошибками в коде и некорректным использованием API.

Инструменты OSS сканируют сторонние компоненты, используемые в приложении, для обнаружения известных уязвимостей и проблем безопасности. Они анализируют открытый исходный

код, который входит в состав приложения, и предупреждают о наличии уязвимостей, связанных с использованием устаревших или уязвимых версий компонентов. В отличие от SAST, инструменты OSS анализируют не сам исходный код приложения, а только используемые зависимости.

Интерактивное тестирование приложений (IAST) предоставляет возможность тестирования приложения во время его выполнения и анализа исходного кода. IAST может обнаруживать уязвимости в режиме реального времени, основываясь на взаимодействии с приложением, а также на статическом анализе кода. Это позволяет обнаруживать и анализировать уязвимости, связанные как с конфигурацией и сетевыми настройками, так и с ошибками в коде.

Каждый из этих инструментов имеет свои преимущества и ограничения, и часто их использование в комбинации может быть наиболее эффективным для обеспечения безопасности приложений. SAST является мощным инструментом для обнаружения уязвимостей, связанных с ошибками в коде, и может быть особенно полезным на ранних стадиях разработки. Однако в зависимости от конкретных потребностей и контекста проекта может потребоваться комбинированное использование нескольких инструментов AppSec для достижения наилучших результатов.

При выборе инструмента SAST для защиты SDLC нужно искать инструмент со следующими характеристиками:

1) с удобным для разработчиков и пользователей интерфейсом, который прост в использовании и понимании даже для сотрудников, не связанных с безопасностью;
2) с возможностью быстрого сканирования, чтобы не замедлять процесс разработки;
3) с низким уровнем ложных срабатываний, так как это сокращает время и усилия, необходимые разработчикам для просмотра и проверки результатов вручную;
4) с простой интеграцией в существующий конвейер CI/CD.

С помощью этих типов инструментов SAST организации могут гарантировать, что их ПО разрабатывается с учетом безопасности, снижая риск уязвимостей и повышая общую безопасность приложений.

3. Пентестирование

Пентестирование – процесс активного исследования системы с целью выявления уязвимостей и проверки ее защищенности. Пентестеры, также известные как этические хакеры, имитируют атаки злоумышленников, чтобы помочь организациям улучшить безопасность своих систем.

Цели пентестирования:

– идентификация уязвимостей: их обнаружение и документирование;
– оценка уровня безопасности: пентест позволяет оценить эффективность существующих механизмов защиты и выявить слабые места;
– проверка соответствия требованиям.

Виды пентестирования:

– черный ящик (Black Box): тестеру предоставляется минимальная информация о системе, и он проводит атаки с позиции внешнего злоумышленника;
– белый ящик (White Box): тестер получает полный доступ к системе, включая исходный код и документацию;
– серый ящик (Gray Box): тестеру предоставляется частичная информация о системе, чтобы смоделировать позицию внутреннего пользователя или злоумышленника с некоторым уровнем привилегий.

Методология пентестирования

На подготовительном этапе важно полностью понять потребности и цели клиента перед началом пентеста. Для этого необходимо собрать информацию о целевой системе, включая IP-адреса, домены, сетевую инфраструктуру и другие факторы.

Второй этап включает анализ уязвимостей. Для этого с использованием специализированных инструментов проводится сканирование системы, выявляющее потенциальные уязвимости. После сканирования проводится ручная проверка системы для выявления уязвимостей, которые могут быть упущены автоматическими инструментами.

Затем выявленные уязвимости эксплуатируются. Пентестер использует обнаруженные уязвимости для проведения контролируе-

мых атак на систему. Целью пентестера является получение контроля над системой или доступа к конфиденциальным данным.

На этапе постэксплуатации пентестер старается сохранить доступ к системе для дальнейшего исследования и документирования. По завершении тестирования пентестер составляет подробный отчет, в котором описываются используемые методы, обнаруженные уязвимости и рекомендации по их устранению.

Лучшие практики и рекомендации

1. Соблюдение этики: при проведении пентеста необходимо соблюдать этические принципы и действовать в рамках законодательства.

2. Сотрудничество с командой безопасности: важно установить сотрудничество с внутренней командой безопасности организации для обмена информацией и координации действий.

3. Обновление знаний: пентестирование – это постоянно развивающаяся область, поэтому важно поддерживать актуальные знания и навыки с помощью обучения и профессионального развития.

4. Обработка результатов: полученные в результате пентеста данные и уязвимости должны быть обработаны и исправлены совместными усилиями команды безопасности и разработчиков.

Пентестирование должно проводиться регулярно, чтобы поддерживать высокий уровень безопасности и предотвращать возможные атаки злоумышленников.

4. Анализ состава программного обеспечения (SCA)

Код, лежащий в основе многих (на самом деле большинства) приложений сегодня, включает компоненты с открытым исходным кодом. Но открытый исходный код может содержать критические уязвимости, такие как недавно обнаруженный эксплойт Log4Shell.

Анализ состава ПО – лучший выбор для поиска уязвимостей в пакетах с открытым исходным кодом и изучения того, как их исправить. Он дает возможность защитить код и работоспособность приложений.

Software Composition Analysis (SCA) – это методология безопасности приложений для управления компонентами с открытым исходным кодом. Используя SCA, команды разработчиков могут

быстро отслеживать и анализировать любой компонент с открытым исходным кодом, добавленный в проект.

Инструменты SCA могут обнаруживать все связанные компоненты, поддерживающие их библиотеки, а также их прямые и косвенные зависимости. Инструменты SCA также могут обнаруживать лицензии на ПО, устаревшие зависимости, а также уязвимости и потенциальные эксплойты. В процессе сканирования создается спецификация материалов (BOM), обеспечивающая полную инвентаризацию программных активов проекта.

SCA сам по себе не является чем-то новым, но растущее внедрение открытого исходного кода за последние несколько лет сделало его ключевым процессом для обеспечения безопасности приложений. В результате инструменты SCA получили широкое распространение. Но не все решения SCA одинаково полезны. Современные методы разработки ПО, включая понятие DevSecOps, требуют, чтобы SCA был ориентирован на разработчиков, предоставляя командам удобные инструменты, а группам безопасности – возможность направлять разработчиков, чтобы они могли использовать безопасность во всем SDLC.

Компоненты с открытым исходным кодом становятся основными строительными блоками в ПО практически во всех вертикалях. Инструменты SCA помогают отслеживать используемые приложениями компоненты с открытым исходным кодом, что очень важно как с точки зрения производительности, так и с точки зрения безопасности.

По оценкам Gartner, более 70 % приложений содержат недостатки, связанные с использованием открытого исходного кода. Открытый исходный код составляет до 90 процентов состава кода приложений. Конечно, приложения состоят не только из открытого исходного кода. Одна из проблем, с которыми сталкиваются организации, пытающиеся защитить свою кодовую базу, заключается в том, что приложения собираются из разных строительных блоков, которые необходимо защитить, чтобы иметь возможность эффективно управлять рисками и снижать их.

Трудно переоценить роль, которую играет открытый исходный код в стимулировании цифровой трансформации. Вместе с обла-

ком и DevOps открытый исходный код является одним из ключевых факторов, помогающих компаниям оцифровывать свои услуги и использовать свои технологии для лучшей конкуренции на сегодняшнем рынке ПО.

Использование пакетов с открытым исходным кодом, которые предоставляют точно такую же функциональность, помогает снизить эти затраты. Открытый исходный код по самой своей природе очень гибкий и при необходимости может быть легко настроен. При поддержке сообщества открытый исходный код часто безопаснее, поскольку он проверяется более тщательно. Он бесплатный, а также помогает организациям избежать привязки к поставщику.

Все эти преимущества приводят к повышению эффективности и объясняют высокий уровень внедрения открытого исходного кода в организациях, стремящихся ускорить выход на рынок.

Современные цепочки поставок программного обеспечения

Открытый исходный код – это лишь одна часть головоломки, включающей современное облачное приложение. Приложения сегодня больше собираются, чем создаются. Кроме пакетов с открытым исходным кодом они включают проприетарный код, контейнеры и инфраструктуру в виде кода, и это лишь некоторые из строительных блоков, используемых в рамках этой новой цепочки поставок ПО, являющихся потенциальной точкой входа для злоумышленников.

Уязвимость, эксплуатируемая в одной части цепочки поставок, может быть использована для заражения всего приложения, тем самым расширяя поверхность атаки, требующую защиты. Например, Octopus Scanner GitHub обнаружил вредоносный код, предназначенный для перечисления и бэкдора среды IDE NetBeans с открытым исходным кодом Apache. Метод атаки здесь – воздействие на цепочку поставок путем злоупотребления процессом сборки и распространения полученных артефактов, при этом затронутые проекты могут быть клонированы, разветвлены и использованы многими различными системами – вот что сделало эту атаку интересной, но, к сожалению, не уникальной. Недавняя атака SolarWinds, на этот раз нацеленная на проприетарное ПО, еще раз

демонстрирует растущий риск, который современная цепочка поставок ПО представляет для организаций.

По определению, проекты с открытым исходным кодом являются общедоступными и видимыми для всех, включая злоумышленников. Любая уязвимость, обнаруженная и исправленная в них, неявно открыта для злоумышленников. Чем популярнее проект с открытым исходным кодом, тем привлекательнее будет пакет, поскольку влияние атаки шире.

Как определено выше, SCA – это общий термин для методологий и инструментов безопасности приложений, которые сканируют приложения (например, SAST) как правило во время разработки для сопоставления компонентов с открытым исходным кодом, используемых в приложении, и последующего выявления уязвимостей безопасности и проблем с лицензиями на ПО, которые они вызывают. Чтобы успешно снижать риски, связанные с этими компонентами, организации внедряют методологии и инструменты SCA. При этом они сталкиваются с рядом проблем, связанных с тем, как открытый исходный код используется для создания современных приложений.

Инструменты SCA предназначены для идентификации и анализа компонентов и зависимостей, используемых в разрабатываемом ПО. Они помогают выявить сторонние компоненты, открытые и закрытые уязвимости, лицензионные проблемы и другие потенциальные риски в составе ПО. Основные инструменты SCA:

1) Black Duck от Synopsys предоставляет возможности сканирования и анализа состава ПО, а также обнаружения уязвимостей и отслеживания лицензий. Он интегрируется с различными инструментами разработки и предоставляет обширную базу данных о компонентах и их уязвимостях;

2) Nexus Lifecycle от Sonatype является платформой для анализа состава ПО и управления зависимостями. Он автоматически сканирует проекты и предоставляет информацию о компонентах, уязвимостях и лицензиях. Также интегрируется с различными инструментами разработки;

3) WhiteSource предлагает SCA-инструменты для анализа и управления составом ПО. Он обнаруживает сторонние компонен-

ты, уязвимости и проблемы с лицензиями, предоставляя интеграцию со средами разработки и инструментами непрерывной интеграции;

4) OWASP Dependency-Check – это бесплатный инструмент, разработанный для обнаружения уязвимостей в сторонних зависимостях. Он интегрируется с различными средами разработки и предоставляет отчеты о найденных уязвимостях;

5) Snyk предлагает набор инструментов для анализа и ремедиации уязвимостей в сторонних компонентах. Он интегрируется с различными инструментами разработки и предоставляет информацию о зависимостях, уязвимостях и лицензиях.

Это лишь некоторые из множества доступных инструментов анализа состава ПО. Выбор конкретного инструмента может зависеть от требований и особенностей проекта. Важно выбрать инструмент, который наилучшим образом соответствует потребностям по обнаружению уязвимостей, управлению лицензиями и другим аспектам безопасности ПО.

5. Интерактивное тестирование безопасности приложений

Ни один метод проверки приложений не может гарантировать безопасность на 100 %, но необходимо прилагать усилия для обеспечения безопасности приложений и стремиться ее повысить.

Интерактивное тестирование безопасности приложений (Interactive Application Security Testing, IAST) – это метод, который проверяет приложение на наличие уязвимостей во время фактического использования приложения (либо реальным пользователем, либо автоматическим средством выполнения тестов). Некоторые инструменты IAST даже поставляются с интеграцией IDE, что позволяет запускать анализ безопасности при разработке приложения.

Ядром инструмента IAST являются сенсорные модули, программные библиотеки, включенные в код приложения. Они отслеживают поведение приложения во время выполнения интерактивных тестов. При обнаружении уязвимости будет отправлено оповещение. Примерами таких уязвимостей могут быть жесткое кодирование ключей API в открытом виде, отсутствие очистки вводимых пользователем данных или использование соединений без шифрования SSL.

Чтобы определить, подходит ли конкретный метод тестирования безопасности приложений, важно учитывать его возможности и ограничения.

Статическое тестирование безопасности приложений (SAST) фокусируется на коде, работая на ранних этапах конвейера CI и сканируя исходный код, байт-код или двоичный код, чтобы выявить проблемные шаблоны кодирования, которые противоречат рекомендациям. SAST зависит от языка программирования.

Динамическое тестирование безопасности приложений (DAST) – это метод тестирования черного ящика, который сканирует приложения во время выполнения. Он применяется позже в конвейере CI. DAST является хорошим методом предотвращения регрессий и не зависит от конкретного языка программирования.

Метод IAST похож на DAST в том, что он фокусируется на поведении приложений во время выполнения. Но анализ IAST скорее основан на сочетании тестирования черного ящика, сканирования и анализа внутренних потоков приложений. Преимущество IAST заключается в его способности связывать результаты, подобные DAST, с исходным кодом, таким как SAST. Недостатком этого подхода является то, что IAST зависит от языка программирования и может быть выполнен только позже в конвейере CI.

SCA фокусируется на зависимостях стороннего кода, которые используются в приложении. SCA очень эффективен в приложениях, использующих множество библиотек с открытым исходным кодом. Этот метод также зависит от языка программирования.

IAST – это, по сути, комбинация SAST и DAST. Метод IAST анализирует только код, выполняемый в тестах, например DAST, но также определяет точное место в коде, где была обнаружена уязвимость, как и в случае с SAST. Но у метода IAST есть существенные **преимущества**.

1. IAST сканирует код в продакшене, то есть код, который фактически используется в рабочей среде. Инструменты SAST, как правило, перегружают разработчиков ложными срабатываниями. Иногда строка кода может указывать на проблему безопасности, которая была решена в другой части кодовой базы. IAST фокусируется на вопросах, которые действительно имеют значение.

2. IAST сканирует код, находящийся в разработке. Некоторые инструменты IAST поставляются с интеграцией IDE, чтобы дать инженерам быструю обратную связь о функциях, которые они реализуют. Это смещает проверки безопасности влево в жизненном цикле разработки, когда их дешевле исправить.

3. Быстрое исправление: IAST в отличие от DAST связывает проблемы с расположением кода. Он позволяет просматривать приложение, чтобы найти проблемы, и предоставляет рекомендации по быстрому их устранению, что очень удобно для разработчиков в условиях ограниченного времени.

Есть у IAST и свои **недостатки**. Во-первых, он зависит от языка программирования.

Во-вторых, трудоемкость: метод IAST требует сборки и выполнения приложения (чего нельзя сказать о SAST) и, следовательно, значительных затрат времени в долгосрочной перспективе. Это может быть неважно для проблем, обнаруженных в разработке при использовании подключаемых модулей IDE, из-за быстрой обратной связи. Но при создании больших наборов тестов, которые должны работать во всех производственных выпусках, процесс может замедлиться.

В-третьих, не дает 100 % покрытия кода. Сканируется только тот код, который фактически выполняется, то есть IAST не сканирует весь код. Несмотря на то, что он удаляет множество ложных срабатываний, он также игнорирует код, который контроль качества забыл запустить в тесте.

Как и в случае с любым другим методом тестирования безопасности приложений, важно проанализировать стек технологий и процессы, прежде чем выбрать один из них. В зависимости от выбранного языка программирования IAST может даже не подойти. В таких случаях придется прибегнуть к DAST, который проверяет только входные и выходные данные приложения и не сканирует код.

IAST может предоставить критически важную информацию о безопасности приложения, которая не может быть получена с помощью подходов SAST, но при этом IAST также может значительно замедлить конвейер CI. Таким образом, решения на основе SAST могут быть лучшим вариантом для приложения.

Проведение тестирования безопасности включает семь этапов.

1. Определение целей и требований. Это может включать проверку соответствия определенным стандартам безопасности, обнаружение уязвимостей или оценку общей безопасности системы.

2. Планирование. План тестирования безопасности должен определить методы, инструменты и ресурсы, необходимые для проведения тестирования, с учетом различных аспектов (сетевая безопасность, безопасность приложений, физическая безопасность и т. д.).

3. Идентификация уязвимостей: использование сканеров уязвимостей для сканирования сети, веб-приложений, аудита конфигурации и других методов и техники.

4. Анализ результатов после обнаружения уязвимостей. Нужно оценить серьезность и потенциальные последствия каждой уязвимости. Это поможет определить приоритеты и разработать план действий по устранению уязвимостей.

5. Эксплуатация уязвимостей – это более глубокий анализ уязвимостей путем проверки их на практике. Важно при этом соблюдать этические принципы и получить соответствующее разрешение от владельцев системы.

6. Разработка рекомендаций по устранению обнаруженных уязвимостей на основе результатов тестирования; изменение конфигурации, обновление ПО, применение патчей или другие меры безопасности.

7. Повторное тестирование после выполнения рекомендаций по устранению уязвимостей. Необходимо убедиться, что проблемы были успешно исправлены и система осталась безопасной.

Тестирование безопасности должно проводиться регулярно, а не только в начале разработки. Это поможет обнаруживать и устранять новые уязвимости, связанные с изменениями в системе или обновлениями ПО.

Инструменты тестирования безопасности

Существует широкий спектр инструментов – платных и бесплатных, предназначенных для тестирования безопасности ПО.

Burp Suite – один из наиболее популярных инструментов для тестирования безопасности веб-приложений. Он предоставляет

возможности для сканирования уязвимостей, перехвата и изменения трафика, анализа сессий и других функций, необходимых для обнаружения и устранения уязвимостей.

OWASP ZAP – бесплатный и открытый инструмент, разработанный сообществом OWASP (Open Web Application Security Project) для тестирования безопасности веб-приложений. Он предлагает функционал для сканирования уязвимостей, анализа трафика, инъекций, кросс-сайтового скриптинга и других типов атак.

Nessus – коммерческий инструмент, предоставляющий возможности для сканирования сетей и обнаружения уязвимостей в системах. Он осуществляет сканирование на основе сигнатур, анализирует уязвимости и предоставляет отчеты о найденных проблемах.

Nikto – бесплатный инструмент, предназначенный для сканирования веб-серверов на наличие уязвимостей. Он идентифицирует известные уязвимости: отсутствие патчей, настройку безопасности и другие проблемы, которые могут быть использованы злоумышленниками.

Wireshark – бесплатный инструмент для анализа сетевого трафика. Он позволяет перехватывать и анализировать пакеты данных, что может быть полезно при обнаружении аномалий, атак или уязвимостей в сети.

Важно выбирать подходящий инструмент или комбинацию инструментов в зависимости от требований и целей тестирования безопасности.

Контрольные вопросы

1. Какие виды тестирования безопасности вы знаете?
2. В чём различие между DAST и IAST?
3. Какие инструменты используются при динамическом тестировании?
4. Что включает в себя пентестирование?
5. Какие метрики можно использовать для оценки эффективности тестирования безопасности?

Тесты для самоконтроля

1. Для тестирования работающего приложения используется метод
а) SAST
б) DAST
в) IAST
г) SCA

2. Для динамического тестирования используется инструмент
а) Bandit
б) OWASP ZAP
в) Dependency-Check
г) Ansible

3. Пентестирование типа Black Box даёт
а) полный доступ к системе
б) минимальную информацию о системе
в) доступ через API
г) тестирование только UI

4. Для автоматизации процесса поиска и эксплуатации уязвимостей SQL-инъекций в веб-приложениях используется
а) SQLmap
б) OWASP ZAP
в) Burp Suite
г) Nmap

5. Этот подход позволяет обнаружить уязвимости в зависимостях.
а) статический анализ кода
б) анализ состава ПО (SCA)
в) пентестирование
г) функциональное тестирование

Рекомендуемая литература

1. Бабушкин, В. М. Разработка защищенных программных средств информатизации производственных процессов предприятия : учеб. пособие / В. М. Бабушкин, М. В. Тумбинская. – Москва [и др.] : Инфра-Инженерия, 2024. – 257 с. – URL: znanium.ru/catalog/document?id=451746 (дата обращения: 04.06.2025). – Режим доступа: по подписке. – ISBN 978-5-9729-1618-4.
2. Безопасность разработки в Agile-проектах : Обеспечение безопасности в конвейере непрерывной поставки / Л. Белл, М. Брантон-Сполл, Р. Смит, Д. Бэрд ; пер. с англ. А. А. Слинкин. – Москва : ДМК Пресс, 2018. – 448 с. – ISBN 978-5-97060-648-3.

Практическая работа 1
Разработка веб-приложения для экспорта заметок в различные форматы

Цель – освоить принципы разработки веб-приложений с использованием современных технологий (Flask, PostgreSQL, HTML/CSS) и реализовать функционал экспорта данных.

Сведения, необходимые для выполнения работы

Веб-приложение – это программное обеспечение, работающее в браузере и состоящее из клиентской (Frontend) и серверной (Backend) частей. В данной работе используется:

– Frontend: HTML (структура), CSS (стилизация);
– Backend: Python + Flask (обработка запросов), PostgreSQL (хранение данных).

Дополнительные технологии: ReportLab (генерация PDF), статический анализ кода (Bandit).

Задание

1. Разработать веб-приложение с возможностью:
   • создания, просмотра, редактирования и удаления заметок;
   • экспорта заметок в форматы PDF и TXT.
2. Реализовать взаимодействие с БД PostgreSQL.
3. Провести статический анализ кода с помощью инструментов для выявления уязвимостей.

Вопросы для самоконтроля

1. Какие технологии используются для клиентской и серверной частей приложения?
2. Почему важно избегать хардкодинга паролей в коде?
3. Как работает механизм экспорта данных в PDF?

Практическая работа 2
Статический анализ кода на примере веб-приложения

Цель – научиться выявлять уязвимости в коде с помощью инструмента Bandit и устранять их.

Сведения, необходимые для выполнения работы

Статический анализ кода – это проверка исходного кода без его выполнения для выявления:

- уязвимостей безопасности (SQL-инъекции, XSS, hardcoded secrets);
- ошибок стиля и синтаксиса (PEP 8, несогласованность форматирования);
- потенциальных багов (утечки памяти, неиспользуемые переменные).

Популярные инструменты, используемые при разработке на Python

Инструмент	Основные функции	Плюсы	Минусы
Bandit	Поиск уязвимостей (CWE, OWASP Top 10)	Простота, интеграция с CI/CD	Ограниченная проверка логики
Pylint	Проверка стиля, сложности кода, ошибок	Поддержка кастомных правил	Высокий уровень «шума» (ложных срабатываний)
SonarQube	Комплексный анализ (security, bugs, code smells)	Мультиязычность, интеграция с GitHub	Требует сервер для запуска
MyPy	Проверка типов (type hints)	Помогает избежать runtime-ошибок	Не ищет security-проблемы

Универсальные (мультиязычные):

- **Semgrep**: шаблонный поиск уязвимостей (аналог grep для кода);
- **CodeQL** (GitHub): анализ через SQL-подобные запросы;
- **Checkmarx**: enterprise-решение для SAST (Static Application Security Testing).

Выбор зависит от целей и контекста проекта:

Критерий	Варианты	Пример выбора
Язык программирования	Python, Java, C++, JavaScript	Для Python: Bandit + Pylint
Тип проекта	Веб-приложение, IoT, мобильное приложение	Веб: SonarQube (покрывает OWASP риски)
Интеграция	CI/CD (GitHub Actions, GitLab CI), IDE	Для GitHub: CodeQL
Уровень детализации	Базовый (стиль кода), продвинутый (security)	Security: Bandit + Semgrep
Лицензия	Open-source (бесплатно), коммерческий	Стартап: SonarQube Open Edition

Задание

1. Проанализировать код приложения с помощью выбранного инструмента.
2. Устранить обнаруженные уязвимости.
3. Отключить режим debug.
4. Перенести пароли в переменные окружения (.env).
5. Провести повторный анализ для проверки исправлений.

Вопросы для самоконтроля

1. Какие уязвимости чаще всего обнаруживаются при статическом анализе кода?
2. Почему режим debug опасен в production?
3. Как организовать безопасное хранение паролей?

Практическая работа 3
Динамическое тестирование веб-приложения

Цель – освоить методы динамического тестирования безопасности веб-приложений.

Сведения, необходимые для выполнения работы

Динамическое тестирование (DAST) проверяет работающее приложение на уязвимости:

- межсайтовый скриптинг (XSS);
- отсутствие заголовков безопасности (CSP, X-Content-Type-Options);
- утечка информации (версия сервера).

Задание

1. Просканировать приложение с помощью OWASP ZAP.
2. Устранить найденные уязвимости:
   - добавить заголовки CSP и X-Content-Type-Options;
   - скрыть версию сервера (настроить Gunicorn);
   - проверить защиту от CSRF (Flask-WTF).

Вопросы для самоконтроля

1. Какие уязвимости выявляет OWASP ZAP?
2. Как заголовок CSP защищает от XSS?
3. Почему важно скрывать версию сервера?

Практическая работа 4
Защита от SQL-инъекций в веб-приложениях

Цель – изучить методы защиты от SQL-инъекций на примере Flask и PostgreSQL

Сведения, необходимые для выполнения работы

SQL-инъекции – один из наиболее опасных видов атак, позволяющих злоумышленникам:

1) обходить аутентификацию;
2) получать несанкционированный доступ к данным;
3) модифицировать или удалять информацию в БД.

Основные типы атак:

- классические SQL-инъекции;
- слепые SQL-инъекции (Boolean-based и Time-based);
- инъекции в хранимые процедуры;
- вторичные SQL-инъекции.

Задание

1. Добавьте в приложение форму авторизации, уязвимую к SQL-инъекциям.
2. Для каждой формы проведите следующие атаки:
   - классическая SQL-инъекция (используйте payload: ‘ OR ‘1’=’1);
   - слепая Boolean-based инъекция (используйте условия типа AND 1=1, AND 1=2);
   - Time-based инъекция (используйте функции задержки типа SLEEP(5));
   - попробуйте выполнить UNION-based атаку для извлечения данных.
3. Защита приложения. Для каждой уязвимой формы реализуйте защиту:
   - параметризованные запросы;
   - использование ORM (SQLAlchemy или Django ORM);
   - валидацию входных данных;
   - ограничение прав доступа к БД.

4. Тестирование защиты:
    - повторите все виды атак на защищенное приложение;
    - проведите автоматизированное тестирование с помощью SQLmap;
    - сравните результаты до и после внедрения защиты.

Вопросы для самоконтроля

1. Как работает SQL-инъекция?
2. Почему параметризованные запросы безопасны?
3. Какие ORM совместимы с Flask?
4. Какие типы атак имитирует SQLmap?
5. Как защищенное приложение реагирует на SQLmap?

Практическая работа 5
Настройка безопасности для СУБД

Цель – настроить защиту СУБД: аутентификацию, шифрование, права пользователей.

Сведения, необходимые для выполнения работы

Основные угрозы безопасности СУБД:

1. Несанкционированный доступ:

- вход в систему под чужими учетными данными;
- получение привилегий выше разрешенных.

2. Утечка данных:

- кража конфиденциальной информации (пароли, персональные данные);
- неправильная настройка прав доступа.

3. SQL-инъекции – внедрение вредоносного кода через уязвимые интерфейсы.

4. Отказ в обслуживании (DoS):

- перегрузка сервера запросами;
- блокировка критических таблиц.

5. Незащищенные резервные копии:

- доступ к бэкапам посторонних лиц;
- хранение резервных копий без шифрования.

Для защиты СУБД применяются следующие методы:

1. Аутентификация и авторизация:

- настройка сложных паролей:
  - минимальная длина (12+ символов);
  - обязательное использование спецсимволов и цифр;
- двухфакторная аутентификация (если поддерживается СУБД);
- ролевая модель доступа.

2. Шифрование данных:

- SSL/TLS для соединений;
- шифрование данных «на лету» (TDE – Transparent Data Encryption);
- хеширование паролей (использование алгоритмов bcrypt, Argon2).

3. Контроль доступа – брандмауэры: ограничение доступа к порту СУБД (по умолчанию: 5432 для PostgreSQL, 3306 для MySQL).

4. Резервное копирование:

- автоматизированные бэкапы;
- шифрование резервных копий.

5. Мониторинг и аудит:

- включение журналирования;
- анализ логов, инструменты: ELK-стек (Elasticsearch, Logstash, Kibana), Grafana.

Особенности безопасности для разных СУБД

СУБД	Особенности защиты
PostgreSQL	Гибкая система ролей, SSL, Row-Level Security
MySQL	Плагин validate_password, аудит через Enterprise Edition
MongoDB	RBAC, шифрование на уровне поля, TLS
SQLite	Ограниченные возможности (зависит от файловой системы)

Работа выполняется для одного из типов СУБД:

1) реляционных (SQL) – PostgreSQL;
2) документных (NoSQL) – MongoDB.

Задание 1. Настройка защиты от несанкционированного доступа.

1. Для PostgreSQL:

- измените файл `pg_hba.conf`, разрешив подключения только с локального адреса (127.0.0.1) и одного доверенного IP (например, 192.168.1.100);
- все остальные подключения должны быть заблокированы.

2. Для MongoDB:

- в файле `mongod.cfg` установите параметр `bindIp` только для локального адреса и доверенного IP;
- включите обязательную аутентификацию (`authorization: enabled`).

Задание 2. Настройка аутентификации и шифрования.

1. Для обеих СУБД:

- сгенерируйте SSL-сертификаты с помощью OpenSSL;
- настройте обязательное использование SSL/TLS для подключений.

2. Для PostgreSQL: в файле `postgresql.conf` включите SSL и укажите пути к сертификатам.

3. Для MongoDB: в конфигурационном файле включите режим `requireTLS` и укажите путь к PEM-файлу.

Задание 3. Ограничение прав пользователей.

1. Для PostgreSQL:

- создайте нового пользователя с ограниченными правами (только SELECT, INSERT);
- отзовите все права у роли PUBLIC.

2. Для MongoDB:

- создайте администратора с правами root;
- создайте пользователя приложения с ограниченными правами (только чтение/запись в нужные коллекции).

Задание 4. Настройка брандмауэра для обеих СУБД:

- создайте правило брандмауэра, разрешающее подключения только с доверенного IP;
- создайте правило, блокирующее все остальные подключения к порту СУБД.

Проверьте попытку подключения:

- с разрешенного IP, она должна быть успешной;
- с другого IP, она должна быть заблокирована.

Задание 5. Настройка резервного копирования и мониторинга.

1. Для PostgreSQL:

- создайте PowerShell-скрипт для выполнения полного дампа БД;
- настройте автоматическое выполнение скрипта ежедневно через Планировщик задач.

2. Для MongoDB:

- создайте bat-файл для выполнения mongodump;
- настройте его ежедневное выполнение.

3. Для обеих СУБД:

- настройте сбор и анализ логов подключений;
- реализуйте проверку целостности бэкапов (тестовое восстановление).

Вопросы для самоконтроля

1. Назовите три основные угрозы для безопасности БД.
2. Объясните, почему принцип минимальных привилегий критически важен при настройке доступа к СУБД.
3. Какие типы атак могут быть направлены на СУБД, кроме SQL-инъекций?
4. Какие методы аутентификации поддерживает PostgreSQL (или другая СУБД на ваш выбор)?
5. Как настроить ролевую модель доступа в СУБД? Приведите пример SQL-запроса.
6. Почему двухфакторная аутентификация (2FA) повышает безопасность? Поддерживается ли она в вашей СУБД?
7. Зачем нужно шифрование соединений (SSL/TLS) между клиентом и СУБД?
8. Как настроить SSL в PostgreSQL? Какие файлы для этого необходимы?
9. В чём разница между шифрованием «на лету» (TDE) и шифрованием отдельных полей?
10. Как ограничить доступ к СУБД по IP-адресам? Приведите пример настройки pg_hba.conf.
11. Какие порты используются по умолчанию в PostgreSQL, MySQL и MongoDB?
12. Почему важно отключать удаленный доступ root/администратора к СУБД?
13. Какие методы резервного копирования вы знаете? Как обеспечить безопасность бэкапов?
14. Как настроить автоматическое логирование подключений и запросов в СУБД?
15. Какие инструменты можно использовать для анализа логов СУБД?

Библиографический список

1. Авдошин, С. М. Технологии и продукты Microsoft в обеспечении информационной безопасности : учеб. пособие / С. М. Авдошин, А. А. Савельева, В. А. Сердюк. – 4-е изд. стер. электрон. – Москва : Интернет-Университет Информационных Технологий [и др.], 2025. – 431 с. – URL: www.iprbookshop.ru/146405.html (дата обращения: 04.06.2025). – Режим доступа: по подписке. – ISBN 978-5-4497-0935-6.
2. Бабушкин, В. М. Разработка защищенных программных средств информатизации производственных процессов предприятия : учеб. пособие / В. М. Бабушкин, М. В. Тумбинская. – Москва [и др.] : Инфра-Инженерия, 2024. – 257 с. – URL: znanium.ru/catalog/document?id=451746 (дата обращения: 04.06.2025). – Режим доступа: по подписке. – ISBN 978-5-9729-1618-4.
3. Безопасность разработки в Agile-проектах : Обеспечение безопасности в конвейере непрерывной поставки / Л. Белл, М. Брантон-Сполл, Р. Смит, Д. Бэрд ; пер. с англ. А. А. Слинкин. – Москва : ДМК Пресс, 2018. – 448 с. – ISBN 978-5-97060-648-3.
4. Зиновьева, О. М. Интегрированные системы управления безопасностью : Разработка и аудит : практикум / О. М. Зиновьева, А. М. Меркулова, Н. А. Смирнова. – Москва : МИСиС, 2021. – 84 с. – URL: e.lanbook.com/book/238376 (дата обращения: 04.06.2025). – Режим доступа: по подписке.
5. Positive Technologies: какие уязвимости будут главными угрозами в 2023 году // InformationSecurity : информационный бюллетень : [сайт]. – URL: www.itsec.ru/news/positive-technologies-kakiye-uyazvimosti-budut-glavnimi-ugrozami-v-2023-godu (дата обращения: 16.05.2025).

Глоссарий

AppSec (Application Security) – меры и процессы, направленные на защиту приложений от угроз за счет поиска, исправления и предотвращения уязвимостей на всех этапах жизненного цикла ПО.

Authentication (аутентификация) – процесс проверки подлинности пользователя, устройства или системы. Обычно реализуется через логин/пароль, двухфакторную аутентификацию и т. д.

Authorization (авторизация) – определение прав доступа после успешной аутентификации: что может делать пользователь в системе.

Automated Testing (автоматизированное тестирование) – использование инструментов для автоматической проверки функциональности и безопасности приложения.

Bandit – инструмент статического анализа кода для Python, используемый для поиска уязвимостей безопасности.

Black Box Testing (тестирование «вслепую») – тестирование без знания внутренней структуры приложения, имитирующее действия злоумышленника.

Brute Force Attack (атака «грубой силой») – атака, при которой злоумышленник пытается подобрать учетные данные методом перебора.

Burp Suite – коммерческий инструмент для тестирования безопасности веб-приложений, включающий прокси, сканер уязвимостей и другие функции.

CI/CD (Continuous Integration / Continuous Delivery) – совокупность практик и инструментов, позволяющих часто и надежно доставлять изменения в код. Безопасность может быть интегрирована в эти процессы (DevSecOps).

Content Security Policy (CSP) – HTTP-заголовок, ограничивающий источники, из которых браузер может загружать ресурсы, чтобы предотвратить XSS.

Cross-Site Request Forgery (CSRF) – атака, при которой пользователь неосознанно отправляет запрос от своего имени, например через вредоносный сайт.

Cross-Site Scripting (XSS) – уязвимость, позволяющая внедрять на веб-страницу вредоносный JavaScript, который выполняется в браузере жертвы.

DAST (Dynamic Application Security Testing) – динамическое тестирование безопасности работающего приложения с целью выявления уязвимостей.

Data Encryption (шифрование данных) – преобразование данных в недоступный для чтения формат с помощью криптографических алгоритмов.

Debug Mode (режим отладки) – режим, используемый разработчиками для диагностики ошибок. В боевой среде должен быть отключен, так как может раскрыть конфиденциальную информацию.

DevOps – подход, объединяющий разработку (Dev) и эксплуатацию (Ops) для более быстрой и качественной доставки ПО.

DevSecOps – расширение DevOps, включающее безопасность (Sec) в каждый этап CI/CD.

Error Handling (обработка ошибок) – корректная обработка исключений и ошибок в коде, чтобы избежать утечки информации и сбоев в работе приложения.

Exception (исключение) – событие, возникающее во время выполнения программы, которое нарушает нормальный поток выполнения.

Firewall (брандмауэр) – система, контролирующая входящий и исходящий сетевой трафик на основе заданных правил.

Flask-WTF – расширение Flask для работы с формами, включая защиту от CSRF.

Gunicorn – WSGI-сервер для развертывания веб-приложений на Python.

Hashing (хеширование) – преобразование данных в уникальную строку фиксированной длины, используется для хранения паролей.

HTTPS (HyperText Transfer Protocol Secure) – протокол передачи данных с шифрованием, обеспечивающий безопасность соединения между клиентом и сервером.

IAST (Interactive Application Security Testing) – интерактивное тестирование безопасности, сочетающее элементы SAST и DAST.

Input Validation (проверка пользовательского ввода) – процесс фильтрации и очистки данных, полученных от пользователя, для предотвращения инъекций и других атак.

Injection (инъекция) – уязвимость, позволяющая злоумышленнику внедрять и выполнять нежелательный код (например, SQL-инъекция).

Least Privilege (принцип наименьших привилегий) – назначение пользователям минимально необходимых прав для выполнения своих задач.

Logging (логирование) – запись событий и действий в системе для последующего анализа и реагирования на инциденты.

Man-in-the-Middle (MITM) – атака, при которой злоумышленник перехватывает и модифицирует данные между двумя сторонами.

MyPy – инструмент для проверки типов в Python, помогает избежать ошибок, связанных с неправильным использованием данных.

OWASP (Open Web Application Security Project) – открытая некоммерческая организация, занимающаяся вопросами безопасности веб-приложений. Известна списком OWASP Top 10.

OWASP ZAP (Zed Attack Proxy) – бесплатный инструмент для тестирования безопасности веб-приложений, основанный на прокси-сервере.

ORM (Object-Relational Mapping) – технология, позволяющая работать с БД через объекты, вместо использования SQL-запросов напрямую. Помогает предотвратить SQL-инъекции.

Pentesting (тестирование на проникновение) – симуляция атак на систему с целью выявления уязвимостей.

PEP 8 – стандарт оформления кода на Python.

Penetration Testing (пентест) – целенаправленное тестирование безопасности системы с попыткой взлома.

Permissions (права доступа) – уровень доступа, предоставляемый пользователям или процессам к ресурсам системы.

Phishing (фишинг) – мошенническая атака, при которой злоумышленник выдает себя за доверенный источник, чтобы получить конфиденциальные данные.

PostgreSQL – реляционная система управления БД, поддерживающая широкие возможности по защите данных.

Role-Based Access Control (RBAC) – модель контроля доступа, основанная на ролях пользователей.

ReportLab – библиотека Python для генерации PDF-документов.

SAST (Static Application Security Testing) – статический анализ кода на наличие уязвимостей без запуска приложения.

Secure SDLC (Secure Software Development Life Cycle) – жизненный цикл разработки ПО, в котором безопасность интегрирована на каждом этапе.

Security Headers (заголовки безопасности) – HTTP-заголовки, повышающие уровень безопасности веб-приложения (например, CSP, X-Content-Type-Options).

Session Management (управление сеансами) – контроль за созданием, хранением и завершением сеансов пользователей.

SQL Injection (SQL-инъекция) – атака, при которой злоумышленник внедряет вредоносный SQL-код через поля ввода.

SQLmap – инструмент автоматизации тестирования SQL-инъекций.

SSL/TLS – протоколы шифрования, обеспечивающие безопасную передачу данных между клиентом и сервером.

Static Code Analysis (статический анализ кода) – анализ исходного кода без его выполнения с целью выявления ошибок и уязвимостей.

SonarQube – платформа для статического анализа кода, поддерживающая множество языков программирования.

Semgrep – инструмент поиска уязвимостей в коде по шаблонам, аналог grep для кода.

SCA (Software Composition Analysis) – анализ состава ПО с целью выявления уязвимостей в зависимостях.

Secrets in Code (Hardcoded Secrets) – хранение секретов (токенов, паролей) прямо в коде, опасная практика, ведущая к утечкам.

Threat Modeling (моделирование угроз) – процесс выявления возможных угроз и уязвимостей в архитектуре приложения.

TLS (Transport Layer Security) – современная версия протокола SSL для безопасной передачи данных.

Two-Factor Authentication (2FA) – метод аутентификации, требующий два разных фактора (например, пароль и SMS-код).

Type Hinting (подсказки типов) – возможность указания типов переменных и функций в Python, улучшающая читаемость и безопасность кода.

User Input (пользовательский ввод) – данные, введённые пользователем, которые должны быть проверены и очищены перед использованием.

Validation (проверка данных) – процесс фильтрации и проверки корректности вводимых данных.

Vulnerability (уязвимость) – ошибка в проектировании или реализации ПО, которую можно использовать для совершения атаки.

Web Application Firewall (WAF) – специализированный брандмауэр для защиты веб-приложений от распространённых уязвимостей, таких как XSS и SQL-инъекции.

Ответы на тесты для самоконтроля

Тема 1. Введение в безопасность при разработке программного обеспечения

1 – в. Упаковка дисков – это этап распространения программного продукта, не является частью классического жизненного цикла разработки ПО (например, Waterfall или Agile).

2 – б. Тестирование на проникновение (Penetration Testing) – это один из ключевых методов тестирования безопасности, позволяющий имитировать реальные атаки на систему.

3 – б. Принцип минимальных привилегий означает, что пользователь или процесс получает только те права, которые действительно необходимы для выполнения его задач. Это ключевой принцип информационной безопасности, минимизирующий риск компрометации.

4 – б. На этапе тестирования проводятся проверки всей системы, включая функциональное и нагрузочное тестирование, а также активную проверку безопасности (в том числе поиск уязвимостей), включая автоматические и ручные методы анализа.

5 – а. SDLC – это жизненный цикл разработки ПО, который включает все этапы – от идеи до вывода из эксплуатации.

Тема 2. Безопасное кодирование: основы, стандарты и методы

1 – б. Параметризованные запросы отделяют данные от кода SQL, что предотвращает внедрение вредоносных команд.

2 – а. Static Application Security Testing, метод анализа безопасности приложений на основе статического анализа исходного кода без выполнения программы.

3 – в. Snyk – инструмент, специально предназначенный для обнаружения уязвимостей в зависимостях (библиотеках, пакетах) различных языков программирования и управления ими. У остальных инструментов другое назначение.

4 – б. Валидация ввода – это процесс проверки корректности данных, введённых пользователем. Валидация проверяет, что введённые пользователем данные соответствуют ожидаемому формату, типу и ограничениям.

5 – б. ISO/IEC 27002 – международный стандарт, дающий рекомендации по управлению информационной безопасностью, включая использование криптографически стойких алгоритмов и практик.

Тема 3. Безопасность приложений AppSec

1 – б. DAST (Dynamic Application Security Testing) – динамический анализ безопасности – проводится на работающем приложении, имитируя атаки и проверяя его поведение без доступа к исходному коду.

2 – б. Пентестирование предполагает ручное исследование логических ошибок и проверку сценариев атак, что позволяет выявлять ошибки логики, которые автоматизированные методы могут пропустить.

3 – а. Burp Suite – популярный инструмент для IAST (Interactive Application Security Testing), позволяющий тестировать приложения вручную для анализа и эксплуатации уязвимостей.

4 – б. Content Security Policy.

5 – б. OWASP Dependency-Check – инструмент для анализа состава ПО (Software Composition Analysis, SCA), выявляет уязвимости в сторонних библиотеках и компонентах.

Тема 4. DevSecOps: интеграция безопасности в современные методы разработки ПО

1 – б. Анализ возможных угроз и рисков должен учитываться на ранних этапах разработки. Поэтому проектирование должно включать моделирование угроз, чтобы создать защищенную архитектуру системы.

2 – а. Безопасное управление конфигурацией подразумевает версионный контроль и защиту чувствительных данных через подход Infrastructure as Code (IaC).

3 – б. Пентестирование – это один из ключевых методов тестирования безопасности, позволяющий имитировать реальные атаки на систему, который проводится на этапе тестирования для выявления уязвимостей.

4 – б. Минимизация привилегий снижает риск повреждений в случае компрометации учетной записи. Это фундаментальный принцип информационной безопасности.

5 – б. Обновления необходимы регулярно, но после каждого запуска нецелесообразны. Поэтому обновления по мере выхода новых версий и уязвимостей – правильный подход, позволяющий своевременно закрывать обнаруженные уязвимости и использовать улучшения.

Тема 5. Тестирование безопасности приложений: методы и инструменты

1 – б. DAST (Dynamic Application Security Testing) анализирует приложение в режиме реального времени, без доступа к исходному коду.

2 – б. OWASP ZAP – инструмент для динамического тестирования безопасности веб-приложений, позволяет сканировать и находить уязвимости во время работы приложения.

3 – б. При пентестировании типа Black Box тестер не имеет никаких предварительных данных о системе, имитируя внешнего злоумышленника.

4 – а. SQLmap – это специализированный инструмент для автоматизации поиска и эксплуатации уязвимостей SQL-инъекций.

5 – б. SCA (Software Composition Analysis) помогает выявлять уязвимости в библиотеках и компонентах с открытым исходным кодом, используемых в проекте.

Содержание

Учебное издание

Раченко Татьяна Александровна

ОБЕСПЕЧЕНИЕ БЕЗОПАСНОСТИ ПРИ РАЗРАБОТКЕ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

Учебно-методическое пособие

Редактор *О.И. Елисеева*
Технический редактор *Н.П. Крюкова*
Компьютерная верстка: *Л.В. Сызганцева*
Дизайн обложки: *Г.В. Карасева*

При оформлении обложки использована иллюстрация, созданная с помощью искусственного интеллекта Qwen Chat, а также изображение от starline на Freepik (сайт https://ru.freepik.com).

Подписано в печать 11.12.2025. Формат 60×80/16.
Печать оперативная. Усл. п. л. 6,91.
Тираж 100 экз. Заказ 1-13-25.

Издательство Тольяттинского государственного университета
445020, г. Тольятти, ул. Белорусская, 14,
тел. 8 (8482) 44-91-47, www.tltsu.ru