МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ
Федеральное государственное автономное образовательное учреждение высшего образования
«СЕВЕРО-КАВКАЗСКИЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»
Пятигорский институт (филиал) СКФУ
Колледж Пятигорского института (филиал) СКФУ

МДК.02.06 «Безопасность программного обеспечения»
МЕТОДИЧЕСКИЕ УКАЗАНИЯ ДЛЯ ПРАКТИЧЕСКИХ ЗАНЯТИЙ

Специальность СПО

09.02.11 Разработка и управление программным обеспечением

Пятигорск 2026

Методические указания для практических занятий по дисциплине «Безопасность программного обеспечения» составлены в соответствии с требованиями ФГОС СПО к подготовке выпуска для получения квалификации. Предназначены для студентов, обучающихся по специальности 09.02.11 Разработка и управление программным обеспечением.

Практическая работа 1
Настройка безопасной лабораторной среды. Установка и конфигурация виртуальных машин.

Цель работы:

Изучить процессы настройки безопасной лабораторной среды для проведения анализа уязвимостей и тестирования программного обеспечения. Понять, как создать виртуальные машины для безопасного тестирования и работы с потенциально опасным ПО, изолировав их от основной системы.

Теоретические сведения

Безопасная лабораторная среда — это специально подготовленная изолированная среда, в которой можно безопасно тестировать программное обеспечение, исследовать уязвимости, без риска для основной операционной системы или сети. В таких средах можно моделировать различные сценарии атак и защищенности, не опасаясь распространения угроз.

Виртуализация — это технология, которая позволяет создавать несколько виртуальных машин (ВМ) на одном физическом компьютере. Каждая виртуальная машина работает как отдельный компьютер, имеет свою операционную систему и ресурсы. Виртуализация позволяет изолировать различные операционные системы и приложения друг от друга, что делает ее незаменимой для безопасного тестирования ПО.

Преимущества виртуализации для безопасности:

Изоляция: Каждая виртуальная машина изолирована от других, что предотвращает распространение угроз.

Мгновенное восстановление: В случае компрометации системы можно быстро восстановить исходное состояние, создав снимок виртуальной машины (snapshot).

Конфигурация и тестирование: В виртуальных машинах можно настроить различные операционные системы и конфигурации для тестирования на уязвимости.

Процесс настройки лабораторной среды:

Установка гипервизора: Гипервизор (например, VMware Workstation, Oracle VirtualBox, Hyper-V) позволяет создавать и управлять виртуальными машинами.

Создание виртуальных машин: На гипервизоре создаются ВМ с установленной операционной системой. Это могут быть как Windows, так и различные дистрибутивы Linux.

Настройка сетевой безопасности: Важной частью настройки является создание сетевых мостов и внутренних сетей, которые ограничивают доступ виртуальных машин друг к другу и внешней сети.

Снимки состояния: Виртуальные машины должны быть настроены таким образом, чтобы их состояние можно было легко сохранить и восстановить в случае возникновения проблем.

Инструменты для тестирования:

Важно также установить набор инструментов для тестирования безопасности, таких как:

Wireshark — для анализа сетевого трафика.

Metasploit — для проведения тестов на проникновение.

OWASP ZAP — для анализа уязвимостей веб-приложений.

Burp Suite — для тестирования безопасности веб-приложений.

Модели виртуальных машин:

При создании безопасной лабораторной среды следует учитывать различные модели виртуальных машин:

Изолированная среда: виртуальная машина, которая не подключена к внешней сети.

Лабораторная сеть: группа виртуальных машин, между которыми разрешено взаимодействие.

Среда с реальными сетями: виртуальные машины, подключенные к реальной сети, но с ограниченным доступом.

Важность мониторинга и логирования: для успешного тестирования безопасности крайне важно отслеживать все действия в лабораторной среде. Включение журналов событий, использование инструментов для мониторинга и анализа системных процессов поможет своевременно обнаружить и устранить угрозы.

Контрольные вопросы

1. Что такое безопасная лабораторная среда и какие преимущества она дает при тестировании программного обеспечения?
2. Какие основные функции выполняет гипервизор при создании виртуальных машин?
3. В чем заключается процесс создания виртуальных машин и какие операционные системы могут быть установлены на них?
4. Почему важно изолировать виртуальные машины друг от друга и от основной операционной системы?
5. Какие инструменты можно установить для анализа безопасности в виртуальной лабораторной среде? Приведите примеры.
6. Какова роль снимков состояния виртуальной машины в процессе тестирования безопасности?
7. Какие модели виртуальных машин можно использовать для безопасного тестирования ПО и какие особенности каждой модели?
8. Почему важно использовать мониторинг и логирование в процессе тестирования безопасности, и какие данные важно отслеживать?

Практическая работа 2
Моделирование угроз с использованием STRIDE.

Цель работы:

Ознакомление с методом моделирования угроз STRIDE, который используется для идентификации различных типов угроз на каждом этапе разработки программного обеспечения. Изучение принципов работы модели STRIDE и её применения в реальных проектах для повышения безопасности.

Теоретические сведения

Моделирование угроз — это процесс идентификации, анализа и оценки возможных угроз безопасности, с которыми может столкнуться система. Моделирование угроз позволяет заранее предусмотреть потенциальные уязвимости и разработать меры защиты. Одним из самых известных и эффективных методов моделирования угроз является методология **STRIDE**.

Методология STRIDE — это метод для анализа угроз в системе, разработанный Microsoft. Она состоит из шести основных типов угроз, каждый из которых ориентирован на определенный аспект безопасности. Название STRIDE является аббревиатурой, где каждая буква обозначает конкретную угрозу:

S — Spoofing (Подделка) — угроза, связанная с подменой личности. Это может включать в себя подделку учетных данных, чтобы выдать себя за другого пользователя.

T — Tampering (Изменение) — угроза, связанная с изменением данных или кода с целью вредоносных действий. Это может включать вмешательство в данные на диске или в процессе передачи.

R — Repudiation (Отказ от действия) — угроза, связанная с тем, что участник системы может отрицать выполнение какого-либо действия, не оставляя следов или логов.

I — Information Disclosure (Раскрытие информации) — угроза, при которой конфиденциальные данные могут быть раскрыты неавторизованным лицам.

D — Denial of Service (DoS) (Отказ в обслуживании) — угроза, связанная с намеренным лишением системы или её пользователей доступа к важным ресурсам.

E — Elevation of Privilege (Повышение привилегий) — угроза, при которой пользователь или процесс получает доступ к функционалу, к которому у него не должно быть прав.

Применение STRIDE на практике:

Методология STRIDE используется в процессе разработки и анализа программных систем для оценки возможных угроз и уязвимостей. На каждом этапе жизненного цикла системы (проектирование, разработка, тестирование и эксплуатация) STRIDE помогает выявить угрозы и предложить способы защиты.

Пример использования STRIDE:

Рассмотрим пример для веб-приложения:

Spoofing: возможно использование слабых механизмов аутентификации, что позволяет злоумышленнику подделать свою личность.

Tampering: возможность изменения данных через недостаточно защищенные API.

Repudiation: отсутствие механизмов логирования и аудита, что позволяет пользователю отрицать выполнение действий.

Information Disclosure: утечка конфиденциальной информации из-за недостаточно строгих прав доступа к данным.

Denial of Service: атака на сервер с целью перегрузки и недоступности сервиса для пользователей.

Elevation of Privilege: возможность обычного пользователя получить привилегии администратора из-за ошибок в реализации механизма контроля доступа.

Методы и инструменты для моделирования угроз:

Диаграммы потоков данных (DFD) — использование диаграмм потоков данных для отображения всех возможных путей атаки.

Техника "Что, если" — анализ разных сценариев возможных атак.

Инструменты моделирования угроз — такие как Threat Dragon (OWASP), Microsoft Threat Modeling Tool, которые могут помочь в создании и анализе угроз в системе.

Зачем использовать STRIDE?

Помогает идентифицировать слабые места в проектировании системы.

Повышает осведомленность о потенциальных угрозах среди разработчиков и архитекторов.

Позволяет заранее планировать защиту от возможных атак.

Улучшает качество безопасности системы на всех этапах её разработки.

Контрольные вопросы

1. Что такое моделирование угроз и какие цели оно преследует в контексте разработки программного обеспечения?
2. Что означает аббревиатура STRIDE, и как каждое из ее значений помогает в анализе угроз?
3. Как методология STRIDE может быть применена на разных этапах жизненного цикла разработки программного обеспечения?
4. Какие типы угроз входят в STRIDE и как они могут проявляться в реальных системах?
5. Приведите примеры угроз для веб-приложений по каждому из пунктов STRIDE.
6. Как диаграммы потоков данных (DFD) помогают в моделировании угроз в программных системах?
7. Какие инструменты и методы могут быть использованы для анализа угроз с помощью STRIDE?
8. Почему важно использовать метод STRIDE при проектировании безопасных программных решений, и какие преимущества он дает?

Практическая работа 3
Статический анализ кода (SAST) на Python/Java

Цель работы:

Ознакомление с процессом статического анализа кода для выявления уязвимостей на уровне исходного кода программного обеспечения. Исследование инструментов для выполнения анализа безопасности в программных проектах на языках Python и Java.

Теоретические сведения

Что такое статический анализ кода (SAST)?
Статический анализ кода (Static Application Security Testing — SAST) — это процесс анализа исходного кода программы без её исполнения. Этот подход позволяет выявлять ошибки, уязвимости и потенциальные проблемы безопасности на ранних стадиях разработки, еще до выполнения программы.

Как работает статический анализ?
Статический анализатор кода работает путём сканирования исходного кода программы, без её фактического запуска. Это позволяет обнаружить:

Проблемы с безопасностью, такие как SQL-инъекции, небезопасные методы аутентификации, переполнения буфера и другие.

Ошибки в коде, такие как потенциальные утечки памяти, неправильное управление ресурсами, ненужные зависимости и неэффективное использование памяти.

Проблемы с поддерживаемостью и читаемостью кода, такие как дублирование кода, сложные структуры данных или плохо документированные функции.

Преимущества статического анализа:

Раннее выявление уязвимостей: Ошибки и уязвимости обнаруживаются до того, как приложение запустится, что позволяет избежать эксплуатации этих уязвимостей в рабочей среде.

Автоматизация: Статический анализ можно автоматизировать, что позволяет интегрировать его в процессе непрерывной интеграции (CI) и получить результаты с минимальными усилиями.

Экономия времени и ресурсов: Статический анализ позволяет выявлять большинство ошибок в процессе разработки, а не во время тестирования или после выпуска.

Какие уязвимости можно найти с помощью SAST?

SQL-инъекции: неправильная обработка пользовательских данных, которая позволяет злоумышленнику внедрить вредоносные SQL-запросы.

XSS (Cross-Site Scripting): неправильная обработка ввода пользователя, что позволяет внедрить вредоносный JavaScript-код.

Отсутствие проверок на доступ: методы, которые не проверяют права пользователя перед выполнением действий.

Небезопасные криптографические методы: использование устаревших или ненадежных криптографических алгоритмов.

Утечки памяти: неправильное управление памятью, что может привести к утечкам и сбоям.

Инструменты для статического анализа: Существуют различные инструменты для статического анализа кода, которые можно использовать для выявления уязвимостей:

SonarQube: Это один из самых популярных инструментов для статического анализа кода, который поддерживает множество языков программирования, включая Java и Python. SonarQube анализирует качество кода, находит уязвимости и предоставляет подробную отчетность.

Checkmarx: Один из мощных инструментов для анализа безопасности кода, ориентированный на поиски уязвимостей и потенциальных рисков.

PMD (для Java): Это инструмент для анализа кода Java, который находит уязвимости и дает рекомендации по улучшению кода.

Bandit (для Python): Это инструмент, специально предназначенный для поиска уязвимостей в коде Python, таких как неправильное использование криптографических алгоритмов, небезопасная обработка пользовательских данных и другие ошибки.

Интеграция статического анализа в процесс разработки: Статический анализ можно интегрировать в процесс разработки через систему непрерывной интеграции (CI/CD). Например, с помощью Jenkins, GitLab CI или Travis CI можно настроить запуск статического анализатора на каждом коммите или pull request, чтобы выявлять ошибки в коде на ранних этапах разработки.

Примеры инструментов для статического анализа:

SonarQube — для анализа качества и безопасности кода.

Checkmarx — для поиска уязвимостей безопасности.

PMD — для Java, находит потенциальные ошибки и плохие практики.

Bandit — для Python, находит уязвимости в коде.

Контрольные вопросы

1. Что такое статический анализ кода (SAST) и как он помогает в обеспечении безопасности ПО?
2. Чем статический анализ отличается от динамического анализа?
3. Какие основные уязвимости можно выявить с помощью статического анализа на языках программирования Python и Java?
4. Какие инструменты для статического анализа кода вы знаете? Приведите примеры и особенности каждого инструмента.
5. Как интегрировать статический анализ в процесс разработки ПО с помощью CI/CD?
6. Что такое SQL-инъекция и как её можно выявить с помощью статического анализа кода?
7. Каковы преимущества статического анализа кода по сравнению с другими методами тестирования безопасности?
8. Как можно настроить инструменты для статического анализа, чтобы они эффективно выявляли уязвимости, специфичные для вашего проекта?

Практическая работа 4
Динамический анализ кода (DAST). Сканирование веб-приложения для автоматического поиска уязвимостей

Цель работы:

Ознакомление с процессом динамического анализа кода для выявления уязвимостей веб-приложений при их выполнении. Изучение методов сканирования веб-приложений для поиска таких уязвимостей, как SQL-инъекции, XSS и другие проблемы безопасности.

Теоретические сведения

Что такое динамический анализ кода (DAST)?

Динамический анализ (Dynamic Application Security Testing, DAST) — это процесс анализа работы приложения в реальном времени, когда оно уже выполняется. В отличие от статического анализа, который проверяет код на стадии разработки, динамический анализ ориентирован на выявление уязвимостей, которые могут быть использованы в эксплуатации при взаимодействии приложения с внешними системами.

Как работает DAST?

DAST работает путем тестирования работающего приложения на предмет уязвимостей через интерфейсы, с которыми оно взаимодействует, такие как HTTP, REST API и другие. Это включает в себя:

Отправку различных входных данных в приложение.

Мониторинг его реакции на эти данные, чтобы выявить ошибки, такие как утечки данных, возможности для выполнения SQL-инъекций, XSS-атак и другие уязвимости.

Преимущества динамического анализа:

Тестирование в реальных условиях: DAST помогает выявить уязвимости, которые не видны на стадии разработки, и которые могут быть использованы только в работающем приложении.

Автоматизация: Инструменты для динамического анализа могут автоматически сканировать веб-приложение на наличие уязвимостей, что ускоряет процесс тестирования.

Выявление проблем в реальных сценариях использования: DAST позволяет оценить, как приложение будет реагировать на реальные атаки в продуктивной среде.

Типы уязвимостей, выявляемых с помощью DAST:

SQL-инъекции (SQLi): Возможность для злоумышленника вставить вредоносные SQL-запросы через пользовательский ввод.

XSS (Cross-Site Scripting): Внедрение вредоносного скрипта в веб-страницу, который может быть выполнен на стороне клиента.

CSRF (Cross-Site Request Forgery): Атака, при которой злоумышленник может заставить пользователя выполнить нежелательные действия в веб-приложении.

Необработанные ошибки и уязвимости сессий: Проблемы с управлением сессиями и обработкой ошибок могут привести к утечкам данных.

Инструменты для динамического анализа: Существует несколько популярных инструментов для динамического анализа, которые автоматизируют процесс сканирования веб-приложений:

OWASP ZAP (Zed Attack Proxy): Открытый инструмент, предназначенный для тестирования веб-приложений на уязвимости. Он включает в себя функции автоматического сканирования, ручного тестирования, а также позволяет выявлять уязвимости через интерфейсы веб-приложений.

Burp Suite: Популярный инструмент для тестирования безопасности веб-приложений, который включает в себя прокси-сервер для перехвата трафика, а также инструменты для автоматического сканирования.

Acunetix: Платный инструмент для автоматизированного сканирования веб-приложений и обнаружения уязвимостей, таких как SQL-инъекции и XSS.

Процесс сканирования веб-приложения с использованием DAST:

Настройка сканера: Конфигурация сканера для проверки на определенные типы уязвимостей, такие как SQL-инъекции, XSS, или CSRF.

Запуск сканирования: Запуск сканирования веб-приложения с последующим анализом отчетов.

Анализ результатов: Изучение отчетов с результатами сканирования для выявления уязвимостей, таких как SQL-инъекции, неправильное управление сессиями, ошибки конфигурации.

Пример использования OWASP ZAP для сканирования веб-приложения:

Шаг 1: установите OWASP ZAP и настройте прокси-сервер для перехвата трафика.

Шаг 2: Прокачайте приложение, которое хотите протестировать, через прокси ZAP, чтобы собирать все запросы и ответы.

Шаг 3: Используйте инструмент для автоматического сканирования на предмет уязвимостей, таких как SQLi, XSS и CSRF.

Шаг 4: Проанализируйте отчеты и устраните выявленные проблемы.

Контрольные вопросы

1. Что такое динамический анализ кода (DAST) и чем он отличается от статического анализа (SAST)?
2. Как работает динамический анализ кода и какие уязвимости он помогает выявить?
3. Какие типы уязвимостей могут быть обнаружены с помощью DAST в веб-приложениях?
4. Каковы основные преимущества динамического анализа для выявления уязвимостей в реальных приложениях?
5. Какие инструменты для динамического анализа вы знаете и в чем их особенности? Приведите примеры.
6. Как настроить инструмент для динамического анализа, чтобы он выявил уязвимости, такие как SQL-инъекции или XSS?
7. Какие шаги включает в себя процесс сканирования веб-приложения с использованием DAST?
8. Как анализировать результаты сканирования веб-приложения для выявления уязвимостей?

Практическая работа 5
Анализ конфигурации веб-сервера. Аудит конфигурации nginx/apache на предмет ошибок безопасности

Цель работы: Ознакомление с процессом анализа конфигурации веб-сервера и аудитом настроек серверов Apache и Nginx с целью выявления уязвимостей и неправильных конфигураций, которые могут быть использованы для атак.

Теоретические сведения

Веб-серверы и их роль в обеспечении безопасности:
Веб-серверы, такие как Apache и Nginx, являются основными компонентами, которые управляют запросами от пользователей и обеспечивают доступ к веб-приложениям. Конфигурация этих серверов напрямую влияет на безопасность системы, поскольку неправильные настройки могут привести к уязвимостям, таким как утечка данных, атаки отказа в обслуживании (DoS) или возможность выполнения вредоносных запросов.

Основные типы угроз, связанные с конфигурацией веб-серверов:

Утечка конфиденциальной информации: Неправильная настройка прав доступа может привести к тому, что конфиденциальные данные будут доступны для неавторизованных пользователей.

Атаки на доступность (DoS): Неправильная настройка может привести к перегрузке сервера и недоступности веб-приложения для пользователей.

Вредоносные HTTP-запросы: Ошибки конфигурации могут позволить атакующим использовать сервер для отправки вредоносных запросов (например, SQL-инъекции или XSS).

Уязвимости в модулях и компонентах: Ненадежные или устаревшие модули могут стать точками входа для атак.

Управление правами доступа: неверно настроенные права доступа к важным файлам и папкам на сервере могут позволить злоумышленникам получить доступ к конфиденциальной информации.

Общие практики безопасной конфигурации веб-серверов:

Отключение неиспользуемых модулей и сервисов: это снижает поверхность атаки, убирая лишние компоненты, которые могут быть использованы злоумышленниками.

Настройка прав доступа: Установка ограничений на доступ к важным конфиденциальным данным и настройка прав для разных пользователей и групп.

Использование SSL/TLS для шифрования данных: Включение HTTPS и правильная настройка SSL/TLS для защиты данных, передаваемых между клиентом и сервером.

Защита от атак через HTTP-заголовки: Настройка безопасных HTTP-заголовков для защиты от атак, таких как XSS и Clickjacking.

Обновление и патчинг: Регулярное обновление веб-сервера и его компонентов для защиты от известных уязвимостей.

Аудит конфигурации Apache:
Веб-сервер Apache широко используется и обладает множеством настроек безопасности, которые могут быть отрегулированы:

Access Control: Установка ограничений на доступ к конфиденциальным файлам с использованием директив, таких как Allow, Deny и Require.

Logs: Настройка журналов для отслеживания подозрительных действий и предупреждения о возможных атаках.

Modules: Отключение неиспользуемых модулей, таких как mod_status, mod_autoindex, чтобы уменьшить поверхность атаки.

Аудит конфигурации Nginx:
Веб-сервер Nginx также имеет свои особенности конфигурации для обеспечения безопасности:

SSL/TLS: Настройка безопасных протоколов и шифров для HTTPS.

Rate Limiting: Ограничение количества запросов от одного клиента, чтобы предотвратить атаки DoS.

Access Control: Отключение доступа к файлам конфиденциальной информации и защита от обхода ограничений через HTTP-заголовки.

Обновления и патчи: как и для Apache, регулярное обновление Nginx и его модулей для защиты от новых уязвимостей.

Инструменты для анализа конфигурации:

Nessus: Один из популярных инструментов для сканирования и анализа конфигурации серверов на наличие уязвимостей.

Nikto: Открытый инструмент для сканирования веб-серверов, который может помочь в обнаружении ошибок конфигурации и уязвимостей.

OpenVAS: Полноценная система сканирования безопасности, которая может быть использована для проведения аудита конфигурации серверов.

Проверка и исправление ошибок безопасности в конфигурации:

Проверка конфиденциальных файлов: убедитесь, что конфиденциальные файлы, такие как .htaccess, имеют правильные права доступа.

Проверка SSL/TLS: убедитесь, что используется только безопасный протокол и что сертификат актуален.

Проверка прав доступа: убедитесь, что доступ к важным системным файлам и базам данных ограничен только авторизованными пользователями.

Контрольные вопросы

1. Как неправильная конфигурация веб-сервера может повлиять на безопасность системы?
2. Какие основные угрозы связаны с неправильной настройкой веб-сервера Apache и Nginx?
3. Какие методы могут быть использованы для защиты веб-сервера от атак через неправильные конфигурации?
4. Что такое SSL/TLS, и почему его использование критично для безопасности веб-приложений?
5. Как настроить веб-сервер для защиты от атак SQL-инъекций и XSS через неправильные HTTP-заголовки?
6. Почему важно отключать неиспользуемые модули на веб-сервере, и как это влияет на безопасность?
7. Каковы основные шаги аудита конфигурации веб-сервера Apache?
8. Какие инструменты можно использовать для анализа и тестирования конфигурации веб-сервера на наличие уязвимостей?

9. Как правильно настроить журналы событий веб-сервера для мониторинга безопасности?
10. Почему важно регулярно обновлять веб-сервер и его компоненты?

Практическая работа 6
Реализация безопасной аутентификации. Написание кода регистрации/входа с хешированием паролей и защитой от брутфорса

Цель работы: Изучение процесса создания безопасной системы аутентификации, включая использование хеширования паролей, защиту от атак брутфорс и использование современных методов безопасности для регистрации и входа в систему.

Теоретические сведения

Что такое аутентификация и почему она важна?

Аутентификация — это процесс проверки подлинности пользователя с целью обеспечения доступа к защищенным ресурсам системы. Безопасная аутентификация играет важнейшую роль в защите данных и приложений от несанкционированного доступа.

Методы аутентификации:

Базовая аутентификация (с использованием логина и пароля) — наиболее часто используемый метод, при котором пользователь вводит свои учетные данные, которые проверяются сервером.

Многофакторная аутентификация (MFA) — использование двух или более факторов для подтверждения личности пользователя (например, пароль + OTP, биометрия).

Проблемы с безопасностью при аутентификации:

Перехват паролей: если пароли передаются в открытом виде или хранятся без должной защиты, злоумышленник может их перехватить.

Атаки брутфорс: при недостаточной защите паролей (например, использовании слабых паролей или отсутствии ограничений на количество попыток ввода) злоумышленник может получить доступ через автоматический перебор всех возможных вариантов.

Слабо защищенные пароли: хранение паролей в открытом виде или использование простых хеш-функций без соли может привести к утечке паролей в случае компрометации базы данных.

Хеширование паролей:

Хеширование — это процесс преобразования пароля в строку фиксированной длины с использованием криптографического алгоритма. Пароли должны хешироваться с добавлением соли (random salt), чтобы предотвратить атаку методом подбора хешей.

Алгоритмы хеширования: SHA-256, bcrypt, Argon2.

Соль (salt): случайная строка, которая добавляется к паролю перед его хешированием, чтобы предотвратить использование заранее подготовленных хеш-таблиц (rainbow tables) для взлома.

Принципы безопасного хеширования паролей:

Использование сильных алгоритмов хеширования (например, bcrypt или Argon2).

Добавление соли к паролю перед хешированием.

Использование «костных» алгоритмов, которые требуют вычислительных ресурсов для хеширования (например, bcrypt или scrypt), чтобы затруднить атаки на пароли.

Защита от атак брутфорс:

Атака брутфорс — это метод подбора пароля, при котором перебираются все возможные комбинации. Для защиты от таких атак можно использовать:

Ограничение на количество попыток: после нескольких неудачных попыток входа аккаунт блокируется на некоторое время.

Использование CAPTCHA: для предотвращения автоматических атак.

Многофакторная аутентификация: добавление дополнительных шагов проверки (например, OTP через SMS).

Пример безопасной аутентификации:

Регистрация пользователя: Пользователь вводит имя пользователя и пароль. Сервер генерирует соль, добавляет её к паролю, хеширует результат и сохраняет хеш и соль в базе данных.

Вход в систему: Пользователь вводит свои данные для входа. Сервер извлекает соль из базы данных, добавляет её к введенному паролю, хеширует результат и сравнивает с хешем, сохраненным в базе данных.

Инструменты для реализации безопасной аутентификации:

bcrypt: Криптографический алгоритм для хеширования паролей, который автоматически использует соль и позволяет настроить «костность» хеширования.

Argon2: Современный алгоритм хеширования, обеспечивающий безопасность и устойчивость к атакам на пароли.

Google Authenticator: Для реализации многофакторной аутентификации с использованием OTP.

reCAPTCHA: Сервис от Google для защиты от автоматических атак брутфорс.

Пример реализации безопасной аутентификации на Python (с использованием bcrypt):

```python
import bcrypt

# Регистрация пользователя
password = b"securepassword123"
salt = bcrypt.gensalt()
hashed_password = bcrypt.hashpw(password, salt)

# Сохранение хешированного пароля и соли в базе данных
# В реальной системе пароль и соль сохраняются в базе данных
print(f"Хешированный пароль: {hashed_password}")

# Вход в систему
user_input = b"securepassword123"
if bcrypt.checkpw(user_input, hashed_password):
    print("Аутентификация успешна!")
else:
    print("Неверный пароль!")
```

Контрольные вопросы

1. Что такое аутентификация и почему она критична для безопасности приложений?
2. Какие проблемы могут возникнуть при неправильно организованной аутентификации в системе?
3. Что такое хеширование паролей и как оно помогает в защите паролей пользователей?
4. В чем заключается важность соли при хешировании паролей?
5. Какие алгоритмы хеширования паролей считаются наиболее безопасными?
6. Как защищать систему от атак брутфорс при аутентификации?
7. Какие методы можно использовать для защиты от подбора паролей в случае утечки базы данных?
8. Как работает процесс регистрации пользователя с безопасным хешированием пароля?
9. Почему важно использовать многофакторную аутентификацию для повышения безопасности системы?
10. Как можно применить CAPTCHA для защиты от брутфорс-атак?

Практическая работа 7
Обход WAF. Использование различных техник кодирования и обфускации для обхода простых правил межсетевого экрана уровня приложения.

Цель работы: Ознакомление с методами обхода веб-аппликационных межсетевых экранов (WAF), используя техники кодирования и обфускации для скрытия атак и обхода правил защиты веб-приложений.

Теоретические сведения

Что такое WAF (Web Application Firewall)?

WAF — это средство защиты веб-приложений, которое фильтрует, мониторит и блокирует HTTP-трафик, поступающий в веб-приложение или исходящий из него. WAF используется для защиты от атак, таких как SQL-инъекции, XSS, атаки на сессии и другие уязвимости, специфичные для веб-приложений.

Типичные задачи WAF:

Защита от несанкционированного доступа.

Защита от SQL-инъекций, XSS и других распространенных уязвимостей.

Фильтрация вредоносного трафика.

Типы атак, защищаемых WAF:

SQL-инъекция (SQLi): Вставка вредоносных SQL-запросов в поля ввода, которые затем выполняются в базе данных.

Cross-Site Scripting (XSS): Внедрение скриптов в веб-страницу, выполняемых на стороне клиента.

Cross-Site Request Forgery (CSRF): Атака, при которой злоумышленник заставляет пользователя выполнить нежелательные действия на веб-сайте, на котором он авторизован.

Command injection: Внедрение вредоносных команд в систему через уязвимости в приложении.

Обфускация и кодирование как методы обхода WAF:

Обфускация — это техника изменения структуры данных или кода для того, чтобы скрыть реальные намерения атакующего от систем защиты. Например, при использовании обфускации SQL-запроса можно заменять части кода на эквивалентные, но менее очевидные варианты.

Кодирование — процесс преобразования данных в формат, который будет трудным для распознавания системой защиты, но при этом остаётся читаемым для конечной системы. Примеры включают URL-кодирование, Base64-кодирование.

Как работают WAF и как их можно обойти:
WAF используют различные методы фильтрации, чтобы блокировать известные угрозы:

Белые списки: позволяют разрешить определенные запросы, соответствующие известным паттернам.

Черные списки: блокируют известные вредоносные паттерны.

Однако WAF имеют свои слабые места, например:

Ложные срабатывания: Некоторые легитимные запросы могут быть заблокированы из-за неправильно настроенных правил.

Обфускация и кодирование: если злоумышленник обфусцирует свой запрос или закодирует его, WAF может не распознать атаку.

Неадекватное обновление: Некоторые WAF используют старые базы данных для фильтрации атак, что делает их уязвимыми к новым методам атак.

Примеры обхода WAF:

SQL-инъекция с использованием кодирования: Злоумышленник может закодировать часть вредоносного запроса (например, использовать CHAR(39) вместо простого '), чтобы обойти фильтрацию.

XSS с использованием обфускации: Использование эквивалентных записей для символов, например, заменив < на < или <, чтобы скрыть инъекцию скрипта.

Использование URL-кодирования: Кодирование вредоносных запросов с использованием URL-энкодинга (%20 для пробела, %27 для одинарной кавычки) может позволить обходить фильтрацию по ключевым словам.

Примеры атак и их обход через WAF:

SQL-инъекция: Пример запроса, который может быть заблокирован WAF:

```
SELECT * FROM users WHERE username = 'admin' AND password = 'password';
```

Для обхода WAF можно закодировать запрос:

```
SELECT * FROM users WHERE username = 'admin' AND password = CHAR(39) OR 1=1--';
```

XSS: Пример атаки, который может быть заблокирован WAF:

```
<script>alert('XSS Attack');</script>
```

Для обхода WAF можно закодировать:

```
<scr<script>ipt>alert('XSS Attack');</scr<script>ipt>
```

Методы защиты от обхода WAF:

Обновление правил фильтрации: Регулярное обновление базы данных известных угроз, а также настройка системы для защиты от новых атак.

Использование многоуровневой защиты: Комбинирование различных методов защиты, таких как WAF, IDS/IPS, и проверка на уровне приложений.

Контекстуальная проверка запросов: Вместо простого анализа на наличие известных угроз можно внедрять алгоритмы, которые оценивают запросы в контексте их возможных действий на сервере.

Контрольные вопросы:

1. Что такое WAF и какие задачи он решает при защите веб-приложений?
2. Какие основные типы угроз блокируются WAF?
3. Что такое обфускация и как она помогает обходить защиту WAF?
4. Как кодирование может быть использовано для обхода WAF?
5. Какие уязвимости могут возникнуть при использовании WAF?
6. Приведите пример SQL-инъекции и как её можно обойти с помощью кодирования.
7. Как можно обойти WAF, используя методы обфускации XSS-атаки?
8. Какие основные методы защиты от обхода WAF существуют?
9. Почему важно регулярно обновлять правила фильтрации в WAF?
10. Как можно использовать многоуровневую защиту для улучшения эффективности WAF?

Практическая работа 8
Анализ безопасности Android-приложения. Декомпиляция APK и анализ манифеста и кода на наличие небезопасных практик.

Цель работы: Ознакомление с процессом анализа безопасности Android-приложений, включая декомпиляцию APK-файлов и исследование манифеста и исходного кода для выявления небезопасных практик, таких как утечка данных, незащищенные API, использование слабых криптографических методов и другие уязвимости.

Теоретические сведения

Android-приложения и безопасность:

Android-приложения часто подвергаются атакам, связанным с уязвимостями в коде, неправильной конфигурацией и недостаточной защитой данных. Одним из первых шагов для анализа безопасности Android-приложений является декомпиляция APK. Этот процесс позволяет извлечь исходный код из APK-файла, чтобы выявить потенциальные уязвимости и небезопасные практики.

Что такое декомпиляция APK?

APK (Android Package) — это формат файла, в котором распространяются Android-приложения. Для анализа безопасности необходимо декомпилировать APK-файл в исходный код или на уровне байт-кода, чтобы проанализировать работу приложения.

Декомпиляция — это процесс преобразования байт-кода обратно в исходный код, с возможной потерей некоторых структур данных, но с возможностью изучить логику приложения.

Для декомпиляции используется инструмент APKTool, который позволяет извлекать и пересобирать ресурсы APK-файла, а также Jadx, который преобразует DEX (Dalvik Executable) код в более понятный Java исходный код.

Анализ манифеста приложения:

AndroidManifest.xml — это файл, содержащий информацию о приложении, такую как разрешения, компоненты, действия и конфигурации. Он является ключевым файлом для анализа безопасности:

Разрешения: Важно проанализировать разрешения, которые запрашивает приложение. Например, разрешения на доступ к данным пользователя или камере могут указывать на возможные риски.

API-ключи и чувствительные данные: Некоторые приложения могут содержать ключи API или другие конфиденциальные данные прямо в манифесте, что представляет угрозу для безопасности.

Безопасность данных: Неправильное управление данными (например, использование незашифрованных данных или незащищенных сетевых соединений) также должно быть проанализировано.

Часто встречающиеся уязвимости в Android-приложениях:

Отсутствие шифрования: Приложение может передавать или хранить данные без использования криптографических методов.

Хардкодинг ключей и токенов: Многие приложения хранят ключи или токены доступа прямо в исходном коде, что может привести к утечке этих данных.

Неправильные разрешения: Приложения могут запрашивать разрешения, которые не соответствуют их функционалу, что может быть использовано злоумышленниками.

Ошибки в обработке пользовательского ввода: Отсутствие валидации данных может привести к уязвимостям, таким как SQL-инъекции или XSS.

Инструменты для анализа безопасности Android-приложений:

JADX: Декомпилятор DEX-кода в Java, который позволяет получить исходный код приложения для его анализа.

APKTool: Инструмент для декомпиляции APK-файлов и их повторной сборки, полезен для анализа ресурсов и конфигураций.

Drozer: Платформа для тестирования безопасности Android-приложений, которая может быть использована для анализа уязвимостей на уровне взаимодействия с другими компонентами Android-системы.

QARK (Quick Android Review Kit): Инструмент для анализа Android-приложений, который помогает находить и исправлять уязвимости безопасности.

Примеры уязвимостей, которые можно обнаружить в процессе анализа:

Hardcoded API Keys: API-ключи или другие конфиденциальные данные, встроенные в код приложения, могут быть извлечены через декомпиляцию.

Незашифрованные данные: Отсутствие шифрования данных при их передаче по сети или при хранении.

Уязвимости в разрешениях: Приложение запрашивает разрешения, которые не требуются для его основной функциональности, например, доступ к контактам, если они не используются.

Пример декомпиляции APK:

Скачиваем APK-файл.

Используем APKTool для декомпиляции:

```
apktool d appname.apk
```

Получаем доступ к манифесту и другим ресурсам приложения.

Используем Jadx для декомпиляции DEX-кода:

```
jadx appname.apk
```

Контрольные вопросы:

1. Что такое декомпиляция APK и как она используется для анализа безопасности Android-приложений?
2. Какие основные компоненты конфигурации Android-приложения содержатся в файле AndroidManifest.xml и как они могут повлиять на безопасность?
3. Что такое хардкодинг в контексте Android-приложений и почему это представляет угрозу безопасности?
4. Какие уязвимости могут быть связаны с неправильной обработкой пользовательского ввода в Android-приложениях?
5. Как можно использовать инструмент **Jadx** для декомпиляции Android-приложения?
6. Какие ошибки безопасности могут быть связаны с отсутствием шифрования данных в Android-приложениях?

7. Что такое **APKTool** и как он может быть использован для анализа конфигурации приложения?
8. Каковы основные шаги при анализе безопасности Android-приложений?
9. Какие инструменты можно использовать для нахождения уязвимостей в Android-приложениях, помимо **Jadx** и **APKTool**?
10. Почему важно проводить регулярный аудит безопасности Android-приложений и какие данные стоит искать при анализе?

Практическая работа 9
Инфраструктура и DevOps: Защита контейнеров и API

Цель работы: Ознакомление с методами обеспечения безопасности контейнеризованных приложений и защитой API с использованием инструментов DevOps. Изучение принципов защиты контейнеров, их образов и взаимодействия с API для предотвращения атак и обеспечения надежности системы.

Теоретические сведения

Что такое контейнеризация и как она влияет на безопасность? Контейнеризация — это метод виртуализации, при котором приложения и их зависимости упаковываются в единый контейнер, который может быть развернут на любом сервере с поддержкой контейнеров. Примером таких технологий является Docker и Kubernetes. Контейнеры позволяют изолировать приложения друг от друга, что делает их более удобными для масштабирования и управления.

Преимущества контейнеризации:

Легкость в развертывании и масштабировании приложений.

Изоляция приложений, что позволяет снижать риски, связанные с конфликтацией зависимостей.

Упрощение DevOps процессов и автоматизации развертывания приложений.

Угрозы безопасности контейнеров: Контейнеры, несмотря на свою изоляцию, могут иметь несколько уязвимых точек:

Неавторизованный доступ: Злоумышленники могут попытаться получить доступ к контейнерам через уязвимости в образах или API.

Неизолированные данные: Если контейнеры неправильно настроены, они могут делиться данными с другими контейнерами или с хост-системой.

Уязвимости в образах: Использование устаревших или небезопасных образов контейнеров может привести к эксплуатации уязвимостей.

Необработанные API-запросы: API, через которые контейнеры взаимодействуют, могут быть уязвимыми для атак.

Защита контейнеров:

Использование безопасных образов: Образы контейнеров должны регулярно обновляться, чтобы исключить уязвимости, связанные с устаревшими компонентами.

Минимизация прав: Контейнеры должны работать с минимальными правами доступа, чтобы снизить потенциальный ущерб от атаки.

Изоляция сетевых слоев: Использование сетевых правил и политик для предотвращения нежелательных сетевых взаимодействий между контейнерами.

Проверка уязвимостей образов: Использование инструментов для проверки безопасности образов, таких как Trivy и Clair.

Инструменты для защиты контейнеров:

Docker и Kubernetes: Используются для развертывания контейнеров и их оркестрации.

Trivy: Инструмент для сканирования контейнерных образов на наличие известных уязвимостей.

Clair: Инструмент для анализа уязвимостей в контейнерных образах.

Sysdig Falco: Инструмент для мониторинга безопасности контейнеров в реальном времени.

Aqua Security: Решение для обеспечения безопасности контейнеров, которое включает в себя сканирование уязвимостей, управление политиками безопасности и мониторинг.

Что такое API и почему его защита важна?
API (Application Programming Interface) — это интерфейс, через который приложения взаимодействуют с другими приложениями или компонентами. API является важной частью современной инфраструктуры приложений, и его защита критична для предотвращения атак, таких как:

SQL-инъекции: Уязвимости в API, которые позволяют злоумышленникам манипулировать запросами к базе данных.

DDoS-атаки: Атаки, направленные на перегрузку API, чтобы сделать его недоступным для пользователей.

Неавторизованный доступ: Атаки, при которых злоумышленник получает доступ к чувствительным данным через API.

Методы защиты API:

Аутентификация и авторизация: Использование безопасных методов аутентификации, таких как OAuth, JWT (JSON Web Token), чтобы гарантировать, что только авторизованные пользователи могут взаимодействовать с API.

Ограничение частоты запросов (Rate Limiting): Ограничение количества запросов, которые могут быть выполнены с одного IP-адреса или пользователя, чтобы предотвратить атаки типа DoS.

Шифрование данных: Использование HTTPS для шифрования всех данных, передаваемых через API, чтобы предотвратить перехват информации.

Валидация входных данных: Проверка всех данных, поступающих в API, чтобы предотвратить атаки через небезопасные входные данные, такие как SQL-инъекции или XSS.

Инструменты для защиты API:

API Gateway: Использование API-шлюза для управления трафиком API, а также обеспечения безопасности через аутентификацию, авторизацию и управление трафиком.

OWASP API Security Top 10: Рекомендации и стандарты от OWASP для защиты API от распространенных уязвимостей.

Rate limiting tools: Инструменты, такие как Kong или Nginx, для ограничения частоты запросов.

JWT (JSON Web Token): Стандарт для безопасной передачи информации между клиентом и сервером, обеспечивающий аутентификацию.

Контрольные вопросы

1. Что такое контейнеризация и как она влияет на безопасность приложений?
2. Какие основные угрозы безопасности существуют для контейнеров и как их можно минимизировать?
3. Как можно защитить контейнеры с помощью безопасных образов и минимизации прав?
4. Какие инструменты используются для защиты контейнеров и анализа их безопасности?
5. Почему важно защищать API и какие уязвимости могут возникнуть при неправильно настроенных API?

6. Как можно предотвратить атаки на API, такие как SQL-инъекции или DDoS?
7. Что такое Rate Limiting и как оно помогает в защите API от атак?
8. Какие методы аутентификации и авторизации лучше всего использовать для защиты API?
9. Как использовать **Trivy** и **Falco** для обеспечения безопасности контейнеров?
10. Как можно защитить API с помощью шлюза и какие правила следует применить для предотвращения атак?

Практическая работа 10
Реверс-инжиниринг и защита кода. Методы обфускации, проверки целостности кода, борьба с отладчиками

Цель работы: Изучение методов реверс-инжиниринга и защиты программного кода, включая использование обфускации, проверку целостности и техники защиты от отладчиков, чтобы предотвратить несанкционированное вмешательство и анализ кода.

Теоретические сведения

Реверс-инжиниринг (Reverse Engineering):

Реверс-инжиниринг — это процесс анализа программного обеспечения с целью извлечения его исходного кода, структуры и логики работы. Это может быть сделано для различных целей, включая:

Разбор защищенного программного кода.

Анализ уязвимостей в ПО.

Создание незаконных копий ПО.

Важно понимать, что реверс-инжиниринг сам по себе может быть использован как для улучшения безопасности, так и для ее нарушения, поэтому защита от реверс-инжиниринга критична для защиты интеллектуальной собственности и предотвращения использования уязвимостей.

Обфускация кода:

Обфускация — это метод защиты кода, при котором структура исходного кода изменяется таким образом, что его анализ становится сложным для человека, но функциональность программы не изменяется.

Цели обфускации:

Защита от реверс-инжиниринга.

Усложнение понимания кода злоумышленниками.

Снижение риска кражи интеллектуальной собственности.

Методы обфускации:

Изменение имен переменных и функций на бессмысленные или труднопонимаемые.

Удаление или изменение строк с комментариями.

Применение сложных конструкций, которые трудно анализировать.

Пример: преобразование кода:

```
def calculate_area(radius):
    return 3.14 * radius * radius
```

в

```
def a1(a2):
    return 3.14 * a2 * a2
```

Защита кода от отладчиков:

Отладчики — это инструменты, которые позволяют исследовать работу программы, пошагово выполняя её код. Злоумышленники могут использовать отладчики для анализа и взлома программного обеспечения.

Защита от отладчиков включает в себя методы, которые мешают нормальной работе отладчиков, затрудняя анализ программы:

Проверка на наличие отладчика: Программы могут регулярно проверять, работает ли отладчик (например, через системные вызовы).

Использование таймеров: Задержки или "ловушки времени", которые усложняют использование отладчиков.

Шифрование данных: Использование зашифрованных данных или методов защиты, которые требуют специфической логики для дешифровки.

Использование anti-debugging техник: Методы, которые позволяют программе обнаружить наличие отладчика и вывести ошибку или изменить поведение программы.

Проверка целостности кода:

Целостность кода — это механизм, который обеспечивает защиту программы от модификаций и атак, направленных на изменение кода в процессе выполнения.

Методы проверки целостности:

Контрольные суммы и хеши: Для проверки целостности файла или данных можно использовать контрольные суммы или хеши (например, MD5, SHA-256). Эти хеши сравниваются с оригинальными значениями, чтобы убедиться, что данные не были изменены.

Цифровая подпись: Программное обеспечение может быть подписано с использованием цифровой подписи для проверки подлинности и целостности.

Измерение контрольных точек: Программы могут проверять критические участки кода на наличие изменений, чтобы предотвратить вмешательство.

Инструменты для реверс-инжиниринга и защиты кода:

IDA Pro: Один из самых популярных инструментов для реверс-инжиниринга, который позволяет анализировать двоичные файлы и исходный код.

Ghidra: Открытый инструмент для реверс-инжиниринга, разработанный NSA, который позволяет анализировать программы и выявлять уязвимости.

ProGuard: Инструмент для обфускации Java-кода, который делает код сложным для анализа.

Dotfuscator: Инструмент для обфускации .NET-программ, который позволяет защитить код от реверс-инжиниринга.

Anti-debugging tools: Инструменты, такие как OllyDbg и x64dbg, могут быть использованы для отладки приложений. Защита от них включает проверку наличия отладчика и использование методов, мешающих их работе.

Пример обфускации кода:

Пример использования обфускации в Python:

Оригинальный код:

```
def encrypt_data(data):
    key = "secret"
    return "".join([chr(ord(c) + 3) for c in data])
```

Обфусцированный код:

```
def a1(b1):
    c1 = "secret"
    return "".join([chr(ord(d1) + 3) for d1 in b1])
```

Примеры защиты кода от отладчиков:

Проверка на отладчик: В коде можно встроить проверки на наличие отладчика с помощью системных вызовов. Например, в Windows можно проверить наличие

отладчика с помощью API-функций IsDebuggerPresent или CheckRemoteDebuggerPresent.

Пример на Python:

```python
import ctypes
def is_debugger_present():
    return ctypes.windll.kernel32.IsDebuggerPresent()

if is_debugger_present():
    print("Debugger detected! Exiting...")
    exit()
```

Применение защиты кода:

Важно понимать, что реверс-инжиниринг и защита от отладчиков — это не абсолютная защита. Программы можно декомпилировать или проанализировать с помощью современных инструментов, но защита кода может значительно усложнить эту задачу.

Контрольные вопросы

1. Что такое реверс-инжиниринг, и как он используется для анализа программного обеспечения?
2. Какие основные методы обфускации кода существуют и как они помогают защитить код от анализа?
3. Как работает защита от отладчиков, и какие техники можно использовать для обнаружения отладчиков?
4. Почему важно проверять целостность кода, и какие методы для этого используются?
5. В чем отличие между статической и динамической обфускацией кода?
6. Какую роль в защите кода играют контрольные суммы и хеши?
7. Какие инструменты можно использовать для реверс-инжиниринга программного обеспечения, и как они помогают в анализе безопасности?
8. Какие методы защиты кода от отладчиков можно использовать в программировании?
9. Почему важно применять обфускацию в процессах защиты программного обеспечения, и какие данные чаще всего подвергаются обфускации?
10. Как можно использовать цифровую подпись для защиты целостности программы?

Основная литература:

Практическая подготовка обучающихся специальности 10.05.01 Компьютерная безопасность : методические указания / составители И. В. Влацкая, С. А. Герасименко. — Оренбург : ОГУ, 2024. — 70 с. — Текст : электронный // Лань : электронно-библиотечная система. — URL: https://e.lanbook.com/book/503273 (дата обращения: 12.03.2026). — Режим доступа: для авториз. пользователей.

Череватова, Т. Ф. Нормативное обеспечение в сфере информационных технологий и систем : учебное пособие для СПО / Т. Ф. Череватова. — 2-е изд., стер. — Санкт-Петербург : Лань, 2024. — 84 с. — ISBN 978-5-507-47632-9. — Текст : электронный // Лань : электронно-библиотечная система. — URL: https://e.lanbook.com/book/398513 (дата обращения: 12.03.2026). — Режим доступа: для авториз. пользователей.

Баланов, А. Н. Защита информационных систем. Кибербезопасность : учебное пособие для вузов / А. Н. Баланов. — 3-е изд., стер. — Санкт-Петербург : Лань, 2026. — 280 с. — ISBN 978-5-507-56255-8. — Текст : электронный // Лань : электронно-библиотечная система. — URL: https://e.lanbook.com/book/514704 (дата обращения: 12.03.2026). — Режим доступа: для авториз. пользователей.

Дополнительная литература:

Федорова Г.И. Разработка, внедрение и адаптация программного обеспечения отраслевой направленности. Учебное пособие. Изд.: КУРС, Инфра-М. Среднее профессиональное образование. 2016 г. 336 стр.