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

ПМ.02 РАЗРАБОТКА И ИНТЕГРАЦИЯ МОДУЛЕЙ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ
МДК.02.06 БЕЗОПАСНОСТЬ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ
МЕТОДИЧЕСКИЕ УКАЗАНИЯ ДЛЯ ЛАБОРАТОРНЫХ РАБОТ

Специальности СПО
09.02.11 Разработка и управление программным обеспечением

Пятигорск 2026 г.

Методические указания к выполнению лабораторных работ по дисциплине ПМ.02 Разработка и интеграция модулей программного обеспечения составлены в соответствии с требованиями ФГОС СПО. Предназначены для студентов, обучающихся по специальности 09.02.11 Разработка и управление программным обеспечением.

ПОЯСНИТЕЛЬНАЯ ЗАПИСКА

Дисциплина «Безопасность программного обеспечения» направлена на формирование у студентов компетенций в области разработки безопасного кода, выявления и устранения уязвимостей, а также применения стандартов и методов защиты программных продуктов на всех этапах жизненного цикла.

В результате освоения учебной дисциплины обучающийся должен:

знать:

основные угрозы безопасности программного обеспечения и методы их классификации;

типовые уязвимости программного кода (OWASP Top 10, CWE Top 25);

методы статического и динамического анализа кода;

принципы безопасного программирования и криптографической защиты данных;

стандарты и нормативные документы в области безопасности ПО.

уметь:

выявлять уязвимости в исходном коде программ;

применять методы и средства защиты приложений от типовых атак;

использовать инструменты анализа защищенности ПО;

разрабатывать рекомендации по устранению обнаруженных уязвимостей;

документировать результаты анализа безопасности.

В результате освоения учебной дисциплины студент должен овладевать:

Общими компетенциями:

ОК 01. Выбирать способы решения задач профессиональной деятельности применительно к различным контекстам

ОК 02. Использовать современные средства поиска, анализа и интерпретации информации, и информационные технологии для выполнения задач профессиональной деятельности

ОК 03. Планировать и реализовывать собственное профессиональное и личностное развитие, предпринимательскую деятельность в профессиональной сфере, использовать знания по правовой и финансовой грамотности в различных жизненных ситуациях

ОК 04. Эффективно взаимодействовать и работать в коллективе и команде

ОК 05. Осуществлять устную и письменную коммуникацию на государственном языке Российской Федерации с учетом особенностей социального и культурного контекста

ОК 06. Проявлять гражданско-патриотическую позицию, демонстрировать осознанное поведение на основе традиционных российских духовно-нравственных ценностей, в том числе с учетом гармонизации межнациональных и межрелигиозных отношений, применять стандарты антикоррупционного поведения

ОК 07. Содействовать сохранению окружающей среды, ресурсосбережению, применять знания об изменении климата, принципы бережливого производства, эффективно действовать в чрезвычайных ситуациях

ОК 08. Использовать средства физической культуры для сохранения и укрепления здоровья в процессе профессиональной деятельности и поддержания необходимого уровня физической подготовленности

ОК 09. Пользоваться профессиональной документацией на государственном и иностранном языках.

Профессиональными компетенциями:

ПК 2.1. Проектировать модули программного обеспечения

ПК 2.2. Разрабатывать модули программного обеспечения

ПК 2.3. Выполнять интеграцию модулей и компонентов программного обеспечения

ПК 2.4. Выполнять тестирование и отладку программного обеспечения

ПК 2.5. Осуществлять документирование программных модулей программного обеспечения

ОБЩИЕ МЕТОДИЧЕСКИЕ УКАЗАНИЯ К ВЫПОЛНЕНИЮ ЛАБОРАТОРНЫХ РАБОТ

Варианты лабораторных работ по дисциплине «Безопасность программного обеспечения» для каждого обучающегося – индивидуальные.

Задачи и ответы на вопросы, выполненные не по своему варианту, не засчитываются.

Лабораторная работа выполняется в отдельной тетради. Условия задачи и формулировки вопросов переписываются полностью. Формулы, расчеты, ответы на вопросы пишутся ручкой, а чертежи, схемы и рисунки выполняются карандашом, на графиках и диаграммах указывается масштаб. Вначале задача решается в общем виде, затем делаются расчёты по условию задания. Решение задач обязательно ведется в Международной системе единиц (СИ).

При выполнении лабораторной работы необходимо следовать методическим указаниям: повторить краткое содержание теории, запомнить основные формулы и законы, проанализировать пример выполнения аналогичного задания, затем преступить непосредственно к решению задачи. К зачету допускаются студенты, получившие положительные оценки по всем лабораторным работам.

Правила выполнения лабораторных работ.

Студент должен прийти на лабораторную работу подготовленным к выполнению лабораторной работы.

Каждый студент после проведения работы должен представить отчет о проделанной работе с анализом полученных результатов и выводом по работе.

Таблицы и рисунки следует выполнять с помощью чертежных инструментов (линейки, циркуля, и т.д.) карандашом с соблюдением ЕСКД.

Расчет следует проводить с точностью до двух значащих цифр.

Исправления проводить на обратной стороне листа. При мелких исправлениях неправильное слово (буква, число и т.п.) аккуратно зачеркивается и над ним пишут правильное пропущенное слово (букву, число и т.п.).

Вспомогательные расчеты можно выполнять на отдельных листах, а при необходимости на листах отчета.

Если студент не выполнит лабораторную работу или часть работы, то он выполнит ее во внеурочное время, согласованное с преподавателем.

Оценку по лабораторной работе студент получает с учетом срока выполнения работы, если;

- расчеты выполнены правильно и в полном объеме;

сделан анализ проделанной работы и вывод по результатам работы;

студент может пояснить выполнение любого этапа работы;

отчет выполнен в соответствии с требованиями к выполнению работы.

Лабораторная работа №1.
Введение в безопасность ПО. Основные понятия и угрозы

Цель: изучить основные понятия безопасности программного обеспечения, классификацию угроз и уязвимостей, жизненный цикл уязвимости.

Краткие теоретические сведения

Безопасность программного обеспечения — это свойство ПО функционировать без проявления вредоносных последствий для системы и пользователей при возникновении различных дестабилизирующих факторов.

Основные понятия:

Уязвимость — недостаток или слабость в проектировании, реализации или конфигурации системы, которые могут быть использованы для нарушения политики безопасности.

Угроза — потенциальная возможность нарушения безопасности информации.

Атака — действие, реализующее угрозу путем использования уязвимости.

Риск — мера, сочетающая вероятность реализации угрозы и размер ущерба.

Жизненный цикл уязвимости:

Появление уязвимости (на этапе разработки)

Обнаружение уязвимости (исследователем или злоумышленником)

Публичное раскрытие (disclosure)

Выпуск патча (исправления)

Установка патча пользователями

Классификация угроз (по происхождению):

Естественные (природные катаклизмы)

Техногенные (сбои оборудования)

Антропогенные (действия человека): преднамеренные и непреднамеренные

Задание:

1. Изучить базы данных уязвимостей CVE (Common Vulnerabilities and Exposures) и CWE (Common Weakness Enumeration).
2. Найти в базе CVE три уязвимости за последний год. Для каждой уязвимости определить:

идентификатор CVE;

описание уязвимости;

затронутое ПО/версии;

тип уязвимости по CWE;

уровень критичности (CVSS score);

наличие исправления.

3. Составить отчет, содержащий анализ каждой уязвимости и рекомендации по защите от подобных угроз.

Контрольные вопросы:

1. Чем отличается уязвимость от угрозы?
2. Что такое вектор атаки?
3. Какие существуют стадии жизненного цикла уязвимости?
4. Для чего нужен стандарт CWE?
5. Что означает показатель CVSS?

Лабораторная работа №2.
Переполнение буфера: принципы атаки и методы защиты

Цель: изучить механизм возникновения уязвимости переполнения буфера, освоить методы эксплуатации и способы защиты.

Краткие теоретические сведения

Переполнение буфера (buffer overflow) — одна из наиболее опасных уязвимостей, возникающая при отсутствии контроля границ массивов в языках C/C++.

Принцип возникновения:

При записи данных в буфер (массив) без проверки размера, данные могут выйти за его границы и перезаписать соседние области памяти, включая:

локальные переменные;

адрес возврата функции (return address);

указатель на предыдущий кадр стека.

Методы защиты:

1. Использование безопасных функций: strncpy(), strncat(), snprintf() вместо strcpy(), strcat(), sprintf().
2. Защиты компилятора:

Stack Canary (GS) — случайное значение, помещаемое перед адресом возврата; при переполнении оно изменяется, и программа завершается.

DEP/NX (Data Execution Prevention) — запрет выполнения кода в стеке и куче.

ASLR (Address Space Layout Randomization) — случайное размещение областей памяти.

RELRO — защита от перезаписи секций GOT/PLT.

Задание:

1. Написать программу на C, уязвимую для переполнения буфера (функция vuln с использованием strcpy для копирования пользовательского ввода в локальный буфер 64 байта).
2. Скомпилировать программу с отключенными защитами (-fno-stack-protector -z execstack -no-pie).
3. С помощью отладчика (gdb) определить смещение до адреса возврата.
4. Создать эксплойт (на Python с использованием pwntools или struct), изменяющий поток выполнения на другую функцию (например, win или shell).
5. Скомпилировать программу со включенными защитами (canary, NX) и продемонстрировать их работу.

Контрольные вопросы:

1. Как располагаются данные в стеке при вызове функции?

2. Что произойдет при переполнении буфера при использовании защитного механизма Stack Canary?
3. Чем отличается атака на переполнение буфера в стеке от атаки на кучу (heap overflow)?
4. Как работает ASLR и почему она затрудняет эксплуатацию?
5. Почему использование gets() считается небезопасным?

Лабораторная работа №3.
SQL-инъекции: выявление и предотвращение

Цель: изучить механизм SQL-инъекций, освоить методы их обнаружения и защиты.

Краткие теоретические сведения

SQL-инъекция — это тип атаки, при котором злоумышленник внедряет (инжектит) произвольный SQL-код в запрос, выполняемый приложением. Причина — некорректная обработка пользовательского ввода при формировании SQL-запросов.

Типы SQL-инъекций:

Классическая (внедрение в параметры) — манипуляция условиями WHERE.

Union-based — использование оператора UNION для объединения результатов с другими таблицами.

Error-based — получение информации через сообщения об ошибках СУБД.

Boolean-based (слепая) — анализ поведения приложения при истинных/ложных условиях.

Time-based (слепая) — анализ задержек ответа при использовании функций ожидания.

Задание:

1. Развернуть локально (или использовать онлайн-песочницу) базу данных MySQL и написать простое веб-приложение на PHP/Python/Java с формой авторизации, использующее конкатенацию строк для формирования запроса.
2. Продемонстрировать обход авторизации с использованием SQL-инъекции.
3. Продемонстрировать получение данных из других таблиц (union-based инъекция).
4. Переписать код с использованием параметризованных запросов (PDO для PHP, cursor.execute с параметрами для Python, PreparedStatement для Java).
5. Убедиться, что инъекции больше не работают.

Контрольные вопросы:

1. Каков основной принцип работы SQL-инъекции?
2. Чем параметризованные запросы лучше экранирования?
3. Что такое слепая SQL-инъекция и как ее можно обнаружить?

4. Какие типы данных в пользовательском вводе требуют особой проверки?
5. Может ли использование хранимых процедур полностью защитить от SQL-инъекций?

Лабораторная работа №4.
Межсайтовый скриптинг (XSS): типы и методы защиты

Цель: изучить механизм XSS-атак, научиться выявлять уязвимости и применять методы защиты.

Краткие теоретические сведения

XSS (Cross-Site Scripting) — тип атаки, при котором злоумышленник внедряет в веб-страницу вредоносный скрипт, который выполняется в браузере жертвы.

Типы XSS:

1. **Reflected XSS (отраженный)** — вредоносный код передается в запросе и сразу же отображается в ответе (например, в параметре URL).
2. **Stored XSS (хранимый)** — вредоносный код сохраняется на сервере (в БД, в комментариях, профилях) и отображается всем посетителям.
3. **DOM-based XSS** — уязвимость на стороне клиента, когда JavaScript изменяет DOM страницы, используя непроверенные данные из источника (location.hash, document.URL).

Задание:

1. Создать веб-страницу с формой обратной связи (поле для комментария), которая выводит комментарий без обработки.
2. Внедрить через форму JavaScript-код (<script>alert(document.cookie)</script>).
3. Создать страницу, где параметр из URL (например, ?name=...) выводится без экранирования. Продемонстрировать рефлекторную XSS.
4. Реализовать защиту:

   Для HTML-контекста: замена спецсимволов на HTML-сущности.

   Для JavaScript-контекста: использование JSON.stringify и дополнительное экранирование.
5. Настроить заголовок CSP: Content-Security-Policy: default-src 'self' и проверить, что inline-скрипты блокируются.

Контрольные вопросы:

1. Чем отличается хранимая XSS от отраженной?
2. Какие символы необходимо экранировать при выводе в HTML?
3. Как защищает флаг HttpOnly для cookies?
4. Что такое CSP и как он помогает бороться с XSS?
5. Почему нельзя полностью полагаться только на валидацию ввода на стороне клиента?

Лабораторная работа №5.
Безопасное хранение паролей: хеширование

Цель: изучить современные методы безопасного хранения паролей, освоить использование адаптивных хеш-функций.

Краткие теоретические сведения

Хранение паролей в открытом виде недопустимо. Даже хеширование обычными криптографическими функциями (MD5, SHA-1, SHA-256) недостаточно из-за возможности радужных таблиц (rainbow tables) и высокой скорости вычислений.

Требования к безопасному хранению паролей:

Использование **адаптивных (медленных) хеш-функций**, специально разработанных для паролей:

bcrypt — основан на алгоритме Blowfish, имеет параметр стоимости (work factor).

scrypt — требует большого объема памяти, что затрудняет аппаратный подбор.

PBKDF2 (Password-Based Key Derivation Function 2) — многократное применение HMAC.

Argon2 — победитель конкурса Password Hashing Competition (рекомендуется к использованию).

Процесс хранения пароля:

1. Генерация случайной последовательности (например, 16-32 байта).
2. Вычисление хеша: hash = slow_hash_function(password + salt, cost_factor).
3. Сохранение в БД: salt:hash (или в одном поле с разделителем).

Процесс проверки пароля:

1. Извлечение соли и хеша для пользователя.
2. Вычисление хеша для введенного пароля с той же солью.
3. Сравнение полученного хеша с сохраненным (безопасное сравнение, без раннего выхода).

Задание:

1. Разработать скрипт регистрации и авторизации на Python/JavaScript/PHP, который:

   сначала хранит пароль в открытом виде (демонстрация уязвимости);

   затем хранит простой MD5-хеш (демонстрация уязвимости к радужным таблицам);

   наконец, реализует хранение с использованием bcrypt (или argon2).
2. Для каждого способа продемонстрировать примеры сгенерированных значений.
3. Сравнить время вычисления MD5, SHA-256 и bcrypt с разными факторами стоимости (cost).
4. Реализовать функцию безопасного сравнения строк, не подверженную timing-атакам.

Контрольные вопросы:

1. Почему MD5 и SHA-256 не подходят для хранения паролей?
2. Как работает bcrypt и что такое фактор стоимости (cost factor)?
3. Чем отличается атака по времени (timing attack) и как от нее защититься?
4. Что такое радужная таблица и как соль делает ее бесполезной?

Лабораторная работа №6.
Управление сессиями и безопасность cookies

Цель: изучить механизмы обеспечения безопасности сессий, атрибуты защиты cookies и методы предотвращения атак на сессии.

Краткие теоретические сведения

Сессия — это механизм, позволяющий серверу сохранять состояние между запросами от одного клиента. Обычно сессия идентифицируется по уникальному идентификатору (session ID), который передается в cookies или URL.

Атрибуты безопасности cookies:

Secure — cookie передается только по HTTPS.

HttpOnly — запрещает доступ к cookie из JavaScript (защита от XSS).

SameSite — ограничивает отправку cookie с межсайтовыми запросами:

Strict — только для запросов с того же сайта.

Lax — разрешены переходы по ссылкам и GET-запросы с внешних сайтов.

None — отправляется всегда (требует Secure).

Domain и **Path** — ограничивают область действия cookie.

Expires/Max-Age — время жизни cookie.

Основные угрозы сессиям:

1. **Перехват сессии (Session Hijacking)** — кража session ID (через XSS, сниффинг трафика).
2. **Фиксация сессии (Session Fixation)** — атакующий заставляет жертву использовать известный ему session ID.
3. **Подделка межсайтовых запросов (CSRF)** — выполнение действий от имени жертвы без ее ведома.

Методы защиты:

Регенерация session ID после успешной аутентификации.

Привязка сессии к IP-адресу и User-Agent (может создавать проблемы для мобильных пользователей).

Установка таймаутов сессии.

Использование CSRF-токенов для важных операций.

Корректная настройка атрибутов cookies.

Задание:

1. Разработать простое веб-приложение с аутентификацией (логин/пароль).
2. Проанализировать cookies сессии до и после установки атрибутов (через инструменты разработчика браузера).
3. Настроить cookie с флагами HttpOnly, Secure, SameSite=Lax.
4. Продемонстрировать уязвимость фиксации сессии:

   Получить session ID до логина.

   Передать его жертве (имитация).

   Выполнить вход — убедиться, что session ID не изменился.
5. Реализовать защиту: регенерация ID сессии после логина (session_regenerate_id() в PHP, request.session.cycle_key() в Django).
6. Реализовать защиту от CSRF (для формы изменения пароля): генерация и проверка CSRF-токена.

Контрольные вопросы:

1. Какие атрибуты cookies защищают от XSS-атак?
2. Для чего нужен флаг SameSite и какие значения он принимает?
3. В чем суть атаки фиксации сессии?
4. Что такое CSRF и как от него защититься?
5. Почему не рекомендуется хранить session ID в URL?

Лабораторная работа №7.
Статический анализ кода (SAST)

Цель: освоить использование инструментов статического анализа для выявления уязвимостей на ранних этапах разработки.

Краткие теоретические сведения

SAST (Static Application Security Testing) — метод тестирования безопасности, основанный на анализе исходного кода, байт-кода или бинарных файлов без выполнения программы.

Принцип работы:

Анализатор строит модель программы (абстрактное синтаксическое дерево, граф потоков данных и управления) и применяет набор правил для поиска паттернов, соответствующих уязвимостям.

Преимущества SAST: Обнаружение проблем на ранних стадиях (shift-left). Выявление точного места уязвимости в коде. Не требует работающего приложения.

Недостатки: может давать ложные срабатывания (false positives). Не находит ошибки, возникающие только во время выполнения (конфигурация, окружение). Требует доступа к исходному коду.

Типы обнаруживаемых проблем:

SQL-инъекции

XSS

Переполнение буфера

Небезопасное использование криптографии

Утечки памяти

Неправильная обработка ошибок

Инструменты SAST:

SonarQube — платформа для непрерывного анализа качества и безопасности кода.

PVS-Studio — анализатор для C/C++, C#, Java.

ESLint с плагинами безопасности — для JavaScript.

Bandit — для Python.

FindBugs/SpotBugs — для Java.

Fortify, Checkmarx — коммерческие решения.

Задание:

1. Получить от преподавателя (или найти в открытых источниках) репозиторий с учебным проектом, содержащим преднамеренные уязвимости (например, Damn Vulnerable Web Application, WebGoat, или специально подготовленный код).
2. Установить и настроить один из SAST-инструментов (например, SonarQube в Docker, или Bandit для Python).
3. Запустить анализ проекта.
4. Изучить отчет, классифицировать найденные уязвимости по:

   типу (CWE);

   критичности;

   точному местоположению в коде.
5. Выбрать 3-5 наиболее критичных уязвимостей и исправить их в коде.
6. Повторно запустить анализ и убедиться в устранении проблем.

Контрольные вопросы:

1. В чем основное отличие SAST от DAST?
2. Какие метрики используются для оценки работы SAST-инструментов (полнота, точность)?
3. Что такое ложное срабатывание (false positive) и как с ним бороться?

4. На каком этапе CI/CD целесообразно внедрять SAST?
5. Какие типы уязвимостей SAST не может обнаружить?

Лабораторная работа №8.
Динамический анализ кода (DAST) и фаззинг

Цель: изучить методы динамического анализа и фаззинг-тестирования для поиска ошибок во время выполнения программы.

Краткие теоретические сведения

DAST (Dynamic Application Security Testing) — метод тестирования безопасности, при котором анализ проводится на работающем приложении путем подачи на вход специально сформированных запросов и анализа ответов.

Особенности DAST:

Не требует доступа к исходному коду.

Обнаруживает проблемы, связанные с конфигурацией и окружением.

Позволяет тестировать приложение в реальных условиях.

Может пропускать уязвимости, скрытые глубоко в логике.

Фаззинг (Fuzzing) — техника тестирования, при которой на вход программы подаются случайные, непредусмотренные или семантически некорректные данные с целью вызвать аварийное завершение, зависание или недокументированное поведение.

Виды фаззинга:

1. **Мутационный (mutational)** — модификация корректных входных данных.
2. **Генерационный (generational)** — генерация данных на основе спецификации формата.
3. **С покрытием кода (coverage-guided)** — использование обратной связи от инструментированной программы для генерации данных, увеличивающих покрытие кода (AFL, libFuzzer).

Типы обнаруживаемых проблем:

Переполнение буфера

Use-after-free

Деление на ноль

Утечки памяти

Логические ошибки

Инструменты:

American Fuzzy Lop (AFL) — coverage-guided фаззер.

libFuzzer — in-process фаззер, интегрируется с LLVM.

OWASP ZAP — DAST-инструмент для веб-приложений.

Burp Suite — прокси для ручного и автоматизированного тестирования.

Задание:

1. Написать простую программу на C, обрабатывающую входные данные из файла (парсинг простого формата: "ключ=значение\n"). Внести уязвимость (переполнение буфера при копировании ключа фиксированной длины).
2. Скомпилировать программу с инструментированием для AFL (используя afl-gcc или afl-clang).
3. Создать начальный набор тестовых файлов (корпус).
4. Запустить фаззер (afl-fuzz) и дождаться обнаружения краша.
5. Проанализировать найденный входной файл, воспроизвести уязвимость.
6. Исправить уязвимость, перезапустить фаззер для подтверждения.
7. Для веб-версии: использовать OWASP ZAP для автоматического сканирования учебного веб-приложения и найти уязвимости.

Контрольные вопросы:

1. Чем фаззинг отличается от обычного тестирования?
2. Что такое coverage-guided фаззинг?
3. Какие ограничения имеет динамический анализ (DAST)?
4. Почему важно иметь начальный корпус тестовых данных для фаззинга?
5. Какие типы уязвимостей сложно найти с помощью фаззинга?

Лабораторная работа №9.
Анализ зависимостей и безопасность цепочки поставок ПО

Цель: научиться использовать инструменты для анализа зависимостей, выявления уязвимостей в сторонних библиотеках и управления рисками цепочки поставок.

Краткие теоретические сведения

Современная разработка ПО активно использует сторонние библиотеки и фреймворки. Это создает риски, связанные с уязвимостями в этих компонентах.

Управление зависимостями (Software Composition Analysis, SCA) — процесс идентификации всех компонентов с открытым исходным кодом в кодовой базе, анализ их лицензий и уязвимостей.

Основные понятия:

CVE (Common Vulnerabilities and Exposures) — общедоступная база данных уязвимостей.

CVSS (Common Vulnerability Scoring System) — система оценки критичности уязвимостей.

Транзитивные зависимости — зависимости, которые требуются вашим прямым зависимостям.

Supply Chain Attack — атака, при которой злоумышленник компрометирует один из компонентов цепочки поставок.

Инструменты анализа зависимостей:

OWASP Dependency Check — сканирует проект и сверяет зависимости с базой NVD.

Snyk — коммерческий инструмент с возможностью мониторинга и автоматического исправления.

npm audit / yarn audit — встроенные в экосистему Node.js.

Safety — для Python.

GitHub Dependabot — автоматическое создание PR для обновления уязвимых зависимостей.

Процесс управления зависимостями:

1. Инвентаризация всех используемых компонентов.
2. Регулярное сканирование на наличие известных уязвимостей.
3. Оценка критичности (учитывая, используется ли уязвимый функционал).
4. Обновление версий или применение патчей.
5. Мониторинг появления новых уязвимостей.

Задание:

1. Создать учебный проект с несколькими зависимостями (например, Node.js проект с express, lodash, и другими библиотеками, включая заведомо устаревшие версии).
2. Использовать встроенный инструмент (npm audit) для анализа.
3. Установить и запустить OWASP Dependency Check для этого же проекта.
4. Проанализировать отчеты: найти уязвимости, определить их CVE-идентификаторы, оценить критичность по CVSS.
5. Для одной из найденных уязвимостей определить, используется ли уязвимый функционал в проекте.
6. Обновить уязвимые зависимости до безопасных версий.
7. Настроить автоматическую проверку зависимостей через GitHub Dependabot (если есть возможность) или написать скрипт для регулярной проверки.

Контрольные вопросы:

1. Что такое транзитивные зависимости и чем они опасны?
2. Какая информация необходима для оценки критичности уязвимости в сторонней библиотеке?
3. Чем отличается прямое обновление библиотеки от патча через backporting?
4. Что такое цепочка поставок ПО (software supply chain)?
5. Как часто следует проводить анализ зависимостей в проекте?

Лабораторная работа №10.
Безопасность API (REST)

Цель: изучить основные угрозы безопасности API, освоить методы аутентификации, авторизации и защиты конечных точек.

Краткие теоретические сведения

API (Application Programming Interface) — интерфейс для взаимодействия программ. REST API стал стандартом для веб-сервисов, но часто содержит уязвимости, связанные с неправильной реализацией.

Основные угрозы безопасности API (OWASP API Security Top 10):

1. **Нарушение авторизации на уровне объектов (BOLA/IDOR)** — доступ к объектам, принадлежащим другим пользователям.
2. **Нарушение аутентификации** — слабые механизмы проверки подлинности.
3. **Чрезмерное раскрытие данных** — возврат большего объема данных, чем необходимо.
4. **Отсутствие ограничений на частоту запросов (Rate Limiting)** — позволяет проводить DoS-атаки и брутфорс.
5. **Нарушение авторизации на уровне функций** — доступ к функциям администратора обычными пользователями.
6. **Массовое присваивание (Mass Assignment)** — изменение не предназначенных для этого полей.

Методы аутентификации в API:

Basic Auth — передача логина и пароля в заголовке (небезопасно без HTTPS).

Token-based (JWT) — JSON Web Token, содержащий утверждения (claims) и подписанный сервером.

OAuth 2.0 / OpenID Connect — делегирование доступа сторонним приложениям.

API Keys — простые идентификаторы (не обеспечивают аутентификацию пользователя).

JWT (JSON Web Token) структура:

text

header.payload.signature

Header — алгоритм подписи (HS256, RS256).

Payload — данные (user_id, roles, exp и т.д.).

Signature — подпись, удостоверяющая целостность.

Меры защиты:

Использование HTTPS для всех запросов.

Короткое время жизни токенов (access token) и использование refresh token.

Проверка прав доступа на каждый запрос.

Валидация входных данных.

Rate limiting и ограничение размера запросов.

Задание:

1. Разработать простое REST API на любом языке/фреймворке (Flask, Express, Django REST) с ресурсами: пользователи, записи (posts).
2. Реализовать JWT-аутентификацию:

   Endpoint /login принимает логин/пароль, возвращает JWT.

   Middleware проверяет JWT для защищенных эндпоинтов.
3. Внести уязвимость IDOR: в эндпоинте /api/posts/<id> не проверять, принадлежит ли пост текущему пользователю.
4. Продемонстрировать атаку: один пользователь получает доступ к посту другого, подставив другой ID.
5. Исправить уязвимость, добавив проверку владельца.
6. Реализовать rate limiting (например, с помощью библиотеки express-rate-limit или flask-limiter).
7. Продемонстрировать защиту от массового присваивания (например, используя массовое обновление полей модели и показывая, как можно изменить роль пользователя, если не настроена защита).

Контрольные вопросы:

1. Чем отличается аутентификация от авторизации?
2. Какие поля обязательно должны быть в payload JWT?
3. Как защитить API от подбора паролей?
4. Что такое IDOR и как его предотвратить?
5. Почему нельзя использовать JWT как единственное средство

Лабораторная работа №11.
Межсайтовая подделка запроса (CSRF)

Цель: изучить механизм CSRF-атак, освоить методы защиты, включая CSRF-токены и SameSite cookies.

Краткие теоретические сведения

CSRF (Cross-Site Request Forgery) — тип атаки, при которой злоумышленник заставляет браузер жертвы отправить поддельный запрос к веб-приложению, где пользователь аутентифицирован. В результате приложение выполняет действие (смена пароля, перевод денег) от имени жертвы.

Методы защиты:

1. **CSRF-токены (синхронизатор токенов)**:
   Сервер генерирует уникальный токен и встраивает его в форму.
   При отправке запроса токен проверяется.
   Токен должен быть привязан к сессии пользователя.

Задание:

1. Разработать простое веб-приложение с функцией изменения email пользователя (POST-запрос), использующее cookie-аутентификацию.
2. Создать страницу злоумышленника (отдельный HTML-файл) со скрытой формой, автоматически отправляющей запрос на изменение email.
3. Продемонстрировать CSRF-атаку: будучи аутентифицированным, посетить страницу злоумышленника и увидеть, что email изменился.
4. Реализовать защиту с помощью CSRF-токенов:
   При GET-запросе формы генерировать токен, сохранять в сессии.
   При POST-запросе проверять совпадение токена.
5. Альтернативно (или дополнительно) настроить cookie с флагом SameSite=Lax или Strict и показать, что атака перестает работать.
6. Протестировать защиту.

Контрольные вопросы:

1. Какие условия необходимы для успешной CSRF-атаки?
2. Чем CSRF отличается от XSS?
3. Почему проверка Referer не является абсолютно надежной защитой?
4. Как работает защита с помощью CSRF-токенов?
5. Какие значения может принимать атрибут SameSite и как они влияют на защиту?

Лабораторная работа №12.
Безопасность контейнеров (Docker)

Цель: изучить принципы безопасной сборки и эксплуатации Docker-контейнеров, освоить методы сканирования образов и минимизации поверхности атаки.

Краткие теоретические сведения

Контейнеризация стала стандартом де-факто для развертывания приложений, но контейнеры не обеспечивают полной изоляции и требуют внимания к безопасности.

Основные риски контейнеров:

- Уязвимости в базовом образе.
- Запуск от root (по умолчанию).
- Неограниченный доступ к ресурсам хоста.
- Утечка секретов в образ.
- Небезопасные параметры запуска.

Лучшие практики безопасности Docker:

1. **Использование минимальных базовых образов** (alpine, distroless) для уменьшения поверхности атаки.
2. **Запуск от непривилегированного пользователя** — создание пользователя в Dockerfile и использование USER.
3. **Многоступенчатая сборка (multi-stage builds)** — финальный образ содержит только артефакты, не инструменты сборки.
4. **Не запускать контейнеры с привилегиями** (--privileged).
5. **Ограничение ресурсов** (CPU, память) через --memory, --cpus.
6. **Read-only root filesystem** — монтирование корневой ФС только для чтения.
7. **Не хранить секреты в образе** — использовать переменные окружения при запуске или Docker secrets.
8. **Сканирование образов на уязвимости** (Trivy, Docker Scout, Clair).

Dockerfile best practices:

```
dockerfile
FROM alpine:latest
RUN addgroup -g 1000 -S appgroup && adduser -u 1000 -S appuser -G appgroup
USER appuser
COPY --chown=appuser:appgroup app /app
WORKDIR /app
CMD ["./app"]
```

Задание:

1. Написать простое приложение (например, веб-сервер на Python/Node.js).
2. Создать Dockerfile, который:
   использует полный базовый образ (ubuntu:latest);
   копирует исходный код;
   устанавливает зависимости;
   запускает приложение от root.
3. Собрать образ и просканировать его с помощью Trivy (или docker scan).
4. Проанализировать отчет: найти критичные уязвимости.
5. Переработать Dockerfile:
   использовать минимальный базовый образ (alpine или distroless);
   добавить многоступенчатую сборку;
   создать непривилегированного пользователя и переключиться на него;
   минимизировать количество слоев.
6. Собрать новый образ, повторно просканировать и сравнить результаты.
7. Запустить контейнер и проверить, что приложение работает, но процессы идут от непривилегированного пользователя.

Контрольные вопросы:

1. Почему не рекомендуется запускать контейнеры от root?
2. Чем многоступенчатая сборка улучшает безопасность?
3. Какие преимущества дают минимальные образы (alpine)?
4. Что такое сканирование образов и как оно помогает?
5. Какие риски связаны с монтированием директорий хоста в контейнер?

Лабораторная работа №13.
Безопасность кода на Python: типовые уязвимости и защита

Цель: изучить типовые уязвимости Python-приложений и методы защиты на уровне кода.

Краткие теоретические сведения

Python, несмотря на свою безопасность по сравнению с C/C++, также подвержен ряду уязвимостей.

Типовые уязвимости Python:

Command Injection — использование os.system(), subprocess.Popen(shell=True) с непроверенным пользовательским вводом.

Pickle deserialization — загрузка данных из недоверенных источников с помощью pickle.load() может привести к выполнению произвольного кода.

Eval/exec injection — использование eval() с пользовательским вводом.

Path traversal — недостаточная проверка путей при работе с файлами.

SQL injection (если используется ручное формирование запросов).

YAML deserialization — уязвимости в PyYAML при загрузке из ненадежных источников (!!python/object).

Безопасные альтернативы:

Для команд: использовать subprocess.run() с массивом аргументов, shell=False.

Для сериализации: использовать json вместо pickle для данных от клиента.

Для eval: использовать ast.literal_eval() для безопасного разбора литералов.

Для файлов: использовать os.path.abspath и проверять, что путь начинается с ожидаемой директории.

Для YAML: использовать yaml.safe_load().

Пример уязвимости Command Injection:

```
user_input = request.GET['filename']
os.system(f"cat {user_input}")  # user_input может быть "; rm -rf /"
```

Защита:

```
import subprocess
user_input = request.GET['filename']
# Валидация: только буквы, цифры, точка
if not re.match(r'^[\w\.]+$', user_input):
    raise ValueError("Invalid filename")
subprocess.run(['cat', user_input], check=True)
```

Задание:

Контрольные вопросы:

1. Почему использование shell=True в subprocess опасно?
2. Чем ast.literal_eval() безопаснее eval()?
3. Какие риски связаны с десериализацией pickle?
4. Как защититься от path traversal?
5. Почему нельзя использовать yaml.load() без дополнительных параметров?

Лабораторная работа №14.
Валидация входных данных и обработка ошибок

Цель: изучить принципы безопасной валидации входных данных и обработки ошибок, исключающей утечку чувствительной информации.

Краткие теоретические сведения

Некорректная обработка входных данных — причина большинства уязвимостей. Валидация должна быть многоуровневой и выполняться на стороне сервера.

Принципы валидации:

Белый список (allowlist) предпочтительнее черного списка (denylist) — определять разрешенное, а не запрещенное.

Валидация на стороне сервера — клиентская валидация только для удобства.

Проверка типа, длины, формата, диапазона.

Канонизация — приведение данных к стандартному виду (например, пути, Unicode).

Типы валидации:

- Синтаксическая — соответствует ли формат (email, число, дата).
- Семантическая — имеет ли смысл в контексте (возраст не отрицательный).

Обработка ошибок:

Неправильная обработка ошибок может раскрыть:

- структуру БД (сообщения об ошибках SQL);
- пути к файлам;
- версии ПО;
- стек-трейсы с именами функций.

Принципы безопасной обработки ошибок:

1. Пользователю — минимальное, обобщенное сообщение.
2. В лог — подробная информация (время, IP, трассировка).
3. Логи должны быть защищены от несанкционированного доступа.
4. Не выводить детали исключений в production-среде.

Пример:

```
try:
    result = database.query(user_input)
except DatabaseError as e:
    # Плохо: return str(e)
    # Хорошо:
    logger.error(f"Database error for user {user_id}: {e}")
    return "Произошла внутренняя ошибка. Администратор уведомлен."
```

Задание:

1. Разработать веб-форму для регистрации с полями: имя, email, возраст.
2. Реализовать валидацию на стороне сервера (Python/Flask):
   имя: только буквы, минимум 2 символа;
   email: корректный формат;
   возраст: число от 18 до 120.
3. Внести ошибку в SQL-запрос, вызывающую исключение. Показать, что по умолчанию Flask выводит стек-трейс (в debug-режиме).
4. Настроить production-режим (debug=False) и показать, что пользователь видит общую ошибку 500.
5. Добавить логирование в файл с полной информацией об ошибке.
6. Написать функцию для канонизации путей файлов (защита от path traversal).
7. Проверить ввод на наличие потенциально опасных символов (в контексте вывода).

Контрольные вопросы:

1. Почему белые списки предпочтительнее черных?
2. Какие данные можно получить из подробного сообщения об ошибке SQL?
3. Где и как следует хранить логи безопасности?
4. Что такое канонизация и для чего она нужна?
5. В чем разница между валидацией на клиенте и на сервере?

Лабораторная работа №15.
Безопасность баз данных: шифрование и аудит

Цель: изучить методы защиты данных в СУБД: шифрование, маскирование данных, аудит доступа.

Краткие теоретические сведения

Базы данных хранят критически важную информацию, поэтому их защита является приоритетной.

Уровни защиты БД:

Сетевой уровень — TLS/SSL для шифрования трафика между приложением и БД.

Аутентификация — сильные пароли, интеграция с LDAP/Kerberos.

Авторизация — минимально необходимые привилегии (principle of least privilege).

Шифрование данных:

TDE (Transparent Data Encryption) — шифрование на уровне файлов БД.

Шифрование на уровне приложения — данные шифруются до отправки в БД (потеря возможности поиска).

Шифрование отдельных колонок — встроенными средствами СУБД.

Маскирование данных (data masking) — замена реальных данных на тестовые в не-production средах.

Аудит — логирование всех операций с данными.

Аудит в PostgreSQL:

Включение логирования всех запросов (log_statement = 'all') — только для отладки.

Использование расширения pgaudit для детального аудита.

Аудит в MySQL:

- General log — все запросы.
- Audit plugin (enterprise) или сторонние решения.

Защита от SQL-инъекций на уровне БД:

- Использование хранимых процедур.
- Минимальные привилегии для приложения (только SELECT, INSERT, UPDATE на нужные таблицы).

Задание:

1. Развернуть локально СУБД (MySQL или PostgreSQL).
2. Создать базу данных с таблицей users (id, username, email, password_hash, credit_card, created_at).
3. Настроить аутентификацию: создать пользователя БД для приложения с минимальными правами (только CRUD на таблицу users).
4. Настроить шифрование соединения (TLS) между приложением и БД.
5. Включить аудит:
   для PostgreSQL: установить и настроить pgaudit;
   для MySQL: включить general log (или использовать audit plugin).
6. Выполнить серию запросов и проанализировать логи аудита.
7. Реализовать шифрование данных на уровне приложения для колонки credit_card (например, с использованием библиотеки cryptography для Python).
8. Сравнить размер зашифрованных и открытых данных.
9. Настроить маскирование данных в тестовой среде: написать скрипт, заменяющий реальные email на test@example.com, а номера карт на звездочки.

Контрольные вопросы:

1. Чем TDE отличается от шифрования на уровне приложения?
2. Каковы преимущества и недостатки шифрования отдельных колонок?
3. Для чего нужен аудит базы данных?
4. Какой принцип следует применять при назначении прав доступа к БД?
5. Что такое маскирование данных и когда оно применяется?

Лабораторная работа №16.
Разработка безопасного приложения (итоговый проект)

Цель: применить все полученные знания для разработки безопасного веб-приложения и проведения его комплексного анализа.

Краткие теоретические сведения

Итоговая работа объединяет все темы курса и требует применения комплексного подхода к безопасности на всех этапах — от проектирования до развертывания.

План разработки безопасного приложения:

1. **Проектирование:**
   Моделирование угроз (STRIDE).
   Определение границ доверия.
   Выбор стека технологий с учетом безопасности.
2. **Разработка:**
   Безопасное кодирование (валидация ввода, параметризованные запросы).
   Аутентификация и управление сессиями.
   Контроль доступа (ролевая модель).
   Защита от XSS, CSRF, SQLi.
   Безопасная обработка ошибок.
3. **Тестирование:**
   Статический анализ (SAST).
   Динамический анализ (DAST, ручное тестирование).
   Сканирование зависимостей (SCA).
   Фаззинг (по возможности).
4. **Развертывание:**
   Контейнеризация с соблюдением best practices.
   Настройка HTTPS (Let's Encrypt).
   Security headers (CSP, HSTS, X-Frame-Options).
   Минимальные права для сервисов.

Моделирование угроз STRIDE:

Spoofing — подмена идентификатора.
Tampering — изменение данных.
Repudiation — отрицание действий.
Information disclosure — раскрытие информации.
Denial of service — отказ в обслуживании.
Elevation of privilege — повышение привилегий.

Задание

Вариант А (разработка с нуля):

1. Разработать простое веб-приложение с функциональностью: регистрация, вход, создание/редактирование/удаление личных заметок, просмотр публичных заметок всех пользователей.
2. Реализовать все изученные механизмы защиты.
3. Подготовить Dockerfile для безопасного развертывания.

Вариант Б (анализ существующего приложения):

1. Получить от преподавателя учебное приложение (или взять Damn Vulnerable Web Application).
2. Провести полный анализ безопасности:
   - SAST-сканирование;
   - анализ зависимостей;
   - ручное тестирование (XSS, SQLi, IDOR, CSRF);
   - проверка конфигурации сервера (заголовки, HTTPS);
   - анализ модели угроз.

3. Составить подробный отчет с найденными уязвимостями (по каждому типу — описание, как воспроизвести, CWE, критичность, рекомендации).
4. Исправить найденные уязвимости.

Содержание отчета по итоговой работе:

1. Описание приложения и его функциональности.
2. Модель угроз (диаграмма потоков данных, список угроз по STRIDE).
3. Описание реализованных (или найденных) мер защиты по каждому компоненту.
4. Результаты тестирования (SAST, DAST, ручное).
5. Описание исправленных уязвимостей (если вариант Б).
6. Инструкция по развертыванию (с акцентом на безопасность).

Контрольные вопросы:

1. Какие этапы включает жизненный цикл безопасной разработки (SDL)?
2. Что такое моделирование угроз и для чего оно нужно?
3. Какие security headers необходимо настроить на production-сервере?
4. Как обеспечить безопасность при использовании сторонних библиотек?
5. Почему безопасность должна учитываться на всех этапах разработки, а не только в конце?

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.