Федеральное государственное автономное образовательное учреждение высшего образования
НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ «МОСКОВСКИЙ ФИЗИКО-ТЕХНИЧЕСКИЙ ИНСТИТУТ»

Работа по информационной безопасности

Выполнил: Дурнов Алексей Николаевич
студент М01-406

Долгопрудный, 2024

Содержание

1 Руководство по разработке безопасного программного обеспечения на языке программирования C++

1.1 Область действия руководства

Настоящее руководство предназначено для разработчиков программного обеспечения (далее, ПО), написанного на языке программирования C++, и охватывает ПО разного назначения: встраиваемые системы, высоконагруженные системы и т.д. Настоящее руководство распространяется на весь жизненный цикл разработки, начиная с проектирования системы и заканчивая поддержкой готового продукта.

1.2 Цели организации в области создания безопасного ПО

Основными целями организации в области создания безопасного ПО являются:

1. Минимизация рисков, связанных с различными уязвимостями ПО,
2. Соответствие стандартам и требованиям безопасности,
3. Повышение качества кода,
4. Оптимизация затрат на поддержку и эксплуатацию продукта,
5. Улучшение репутации компании.

1.3 Перечень и описание мер по разработке безопасного ПО

Безопасное программирование на языке C++ требует особого внимания к различным аспектам разработки, начиная от проектирования и заканчивая тестированием и поддержкой. В настоящем руководстве представлены меры, которые помогут создавать безопасное программное обеспечение на языке программирования C++.

1. Проектирование и архитектура
   (a) Разделяйте код на независимые модули, каждый из которых отвечает за свою конкретную задачу. Это сокращает время поиска и устранения уязвимостей.
   (b) Проектируйте приложение таким образом, чтобы оно работало с минимально возможными правами доступа. Это уменьшает возможные последствия успешных атак.
   (c) Определите и документируйте архитектуру безопасности вашего приложения, включая модели угроз и методы защиты.
2. Написание кода
   (a) Работайте с последними стандартами языка, которые предоставляют множество возможностей для безопасного программирования.
   (b) Используйте по возможности умные указатели для автоматического управление памятью, чтобы избегать утечек памяти и двойного освобождения.

(c) Применяйте идиому RAII (Resource Acquisition Is Initialization) для автоматического управления ресурсами, включая файлы, сокеты и другие объекты. Не работайте с примитивами синхронизации (мьютексы и другие) без RAII оберток (std::lock_guard, std::unique_lock и другие).

(d) Используйте std::string вместо сырых строк типа char*. Это предотвращает многие проблемы, связанные с переполнением буфера.

(e) Всегда экранируйте и валидируйте пользовательские данные, особенно если они используются в SQL-запросах, HTML-коде и т.д.

3. Тестирование и анализ кода

(a) Проводите юнит-тестирование и функциональное тестирования для проверки корректности работы отдельных функций и модулей.

(b) Используйте инструменты статического анализа (например, cppcheck, svace, pvs-studio) для выявления потенциальных уязвимостей и ошибок.

(c) Используйте инструменты динамического анализа (например, valgrind, clang code sanitizers) для обнаружения уязвимостей и ошибок.

(d) Проводите фаззинг-тестирования для поиска сбоев программы.

(e) Проводите тестирование на проникновение для выявления уязвимостей в реальных условиях эксплуатации.

4. Управление уязвимостями

(a) Быстро реагируйте на обнаруженные уязвимости, выпуская патчи и обновления.

(b) Следите за новыми уязвимостями в используемых библиотеках и компонентах. Регулярно обновляйте зависимости.

(c) Ведите журнал всех обнаруженных уязвимостей и их статуса (исправлено, отложено и т.д.).

5. Документирование и обучение

(a) Создавайте и поддерживайте документацию по безопасности, включая описания политик, процедур и рекомендуемых практик.

(b) Регулярно обучайте разработчиков основам безопасного программирования и новым технологиям.

(c) Периодически проводите внешние аудиты безопасности для независимой оценки качества безопасности вашего ПО.

6. Инфраструктура

(a) Внедряйте логирование и мониторинг для отслеживания аномального поведения и попыток взлома.

(b) Регулярно делайте резервные копии данных и тестируйте процедуры восстановления.

7. Сопровождение и поддержка

(a) Регулярно обновляйте операционную систему, библиотеки и другие компоненты для устранения известных уязвимостей.

(b) Обучайте пользователей правилам безопасного использования вашего ПО и помогайте им решать проблемы, связанные с безопасностью.

1.4 Распределение ролей и обязанностей, связанных с реализацией мер по разработке безопасного ПО, между работниками

В настоящем руководстве выделяются следующие роли:

1. Разработчики
2. Архитекторы
3. Инженеры по контролю качества
4. Инженеры по автоматизации
5. Менеджеры

Основными обязанностями разработчика, связанных с реализацией мер по разработке безопасного ПО, являются:

1. Проектирование и реализация безопасного кода.
2. Соблюдение стандартов безопасности: применение стандартов и рекомендаций, таких как SEI CERT Coding Standards, MISRA C/C++.
3. Тестирование безопасности: написание unit-тестов и интеграционных тестов, а также проверка кода на наличие уязвимостей.
4. Проведение регулярного ревью кода других разработчиков на предмет соответствия стандартам безопасности и выявление потенциальных уязвимостей.
5. Подготовка технической документации, связанной с вопросами безопасности, включая комментарии к коду, спецификации и отчёты.

Основными обязанностями архитектора, связанных с реализацией мер по разработке безопасного ПО, являются:

1. Выбор архитектурных решений, обеспечивающих высокий уровень безопасности на этапе проектирования.
2. Проведение регулярного ревью кода разработчиков на предмет соответствия стандартам безопасности и выявление потенциальных уязвимостей.
3. Консультирование разработчикам по вопросам безопасности и архитектуры.
4. Оценка рисков, связанных с выбранными архитектурными решениями, и предложение мер по их снижению.

Основными обязанностями инженера по контролю качества, связанных с реализацией мер по разработке безопасного ПО, являются:

1. Проведение функционального и нефункционального тестирования, включая тестирование на проникновение, для выявления уязвимостей и несоответствий стандартам безопасности.
2. Создание и поддержание набора тестов, направленных на проверку безопасности приложения.
3. Анализ результатов тестирования и подготовка отчётов о найденных уязвимостях и рекомендациях по их устранению.

Основными обязанностями инженера по автоматизации, связанных с реализацией мер по разработке безопасного ПО, являются:

1. Разработка и настройка инструментов для автоматизации тестирования безопасности, сборки и разворачивания системы.
2. Внедрение и поддержка процессов непрерывной интеграции и доставки (CI/CD), включая автоматическое выполнение тестов безопасности.
3. Настройка и обслуживание систем мониторинга безопасности, позволяющих оперативно реагировать на инциденты.
4. Оптимизация процессов разработки и тестирования для повышения эффективности и скорости выявления уязвимостей.

Основными обязанностями менеджера проекта, связанных с реализацией мер по разработке безопасного ПО, являются:

1. Планирование и контроль выполнения мероприятий по обеспечению безопасности, координация работы различных команд.
2. Распределение различных ресурсов для реализации мер по разработке безопасного ПО.
3. Поддержание эффективной коммуникации между всеми участниками процесса разработки, обеспечение информированности команды о стандартах и требованиях безопасности.
4. Подготовка и предоставление отчётности руководству и заинтересованным сторонам о ходе выполнения задач по обеспечению безопасности.

1.5 Перечень документации разработчика ПО, связанной с реализацией мер по разработке безопасного ПО

Перечень документации разработчика ПО, связанной с реализацией мер по разработке безопасного ПО состоит из:

1. Описания технических требований к проекту.

2. Кодекс безопасности: внутренний документ, содержащий лучшие практики и стандарты по безопасной разработке ПО.

3. Отчеты по тестированию: результаты статических и динамических анализов кода, отчеты по фаззинг-тестированию, отчеты по тестированию на проникновение.

4. Журналы аудита: записи о проведенных аудитах безопасности и результаты проверок.

5. Политика информационной безопасности: документ описывает общие принципы и правила работы с информацией внутри организации, включая меры по защите данных, доступу к ресурсам и управлению инцидентами.

6. Регламент управления уязвимостями: Определяет порядок выявления, оценки и устранения уязвимостей в разрабатываемом ПО. Включает процессы мониторинга, уведомления и отчетности.

7. Требования к безопасности системы: формулирует конкретные требования к функционалу и архитектуре программы, направленные на обеспечение безопасности. Например, требования к шифрованию данных, аутентификации пользователей, контролю доступа и т.п.

8. Соответствие стандартам безопасности: документы, описывающие соответствие разрабатываемого ПО международным стандартам, таким как ISO/IEC 27001 и другим.

9. Инструкция по применению методов безопасной разработки: рекомендации по использованию безопасных практик программирования, таких как отказоустойчивость, управление ресурсами, минимизация привилегий и другие.

1.6 Правила и требования, относящиеся к планированию и проведению внутренних проверок реализации мер по разработке безопасного ПО, сообщений о результатах

В настоящем руководстве будут изложены ключевые правила и требования, касающиеся планирования и проведения внутренних проверок реализации мер по разработке безопасного программного обеспечения на языке C++. Эти проверки являются важной частью процесса обеспечения безопасности и помогают выявить потенциальные уязвимости.

Целью внутренних проверок является оценка текущего состояния безопасности разрабатываемого ПО и выявление возможных недостатков в процессе его разработки.

Задачи включают:

1. Оценку соблюдения политик и процедур безопасности.

2. Проверку соответствия стандартам и лучшим практикам в области безопасной разработки.

3. Выявление уязвимостей и слабых мест в продукте.

4. Определение областей, требующих улучшения или доработки.

Перед началом проверки необходимо определить цель проверки, а также критерий успеха проверки.

План проверки должен содержать следующие элементы:

1. Описание целей и задач проверки.
2. Перечень модулей и компонентов ПО, подлежащих проверке.
3. Методы и инструменты, которые будут использоваться для проверки (статический анализ кода, динамическое тестирование, тестирование на проникновение и т.д.).
4. Календарный график проведения проверки.
5. Критерии оценки результатов проверки.

После завершения внутренней проверки необходимо подготовить отчет, содержащий следующую информацию:

1. Краткое описание проведенной проверки (цели, объем, методы).
2. Перечень выявленных уязвимостей с указанием уровня серьезности.
3. Рекомендации по устранению выявленных уязвимостей.

После устранения выявленных уязвимостей рекомендуется провести повторную проверку для подтверждения эффективности принятых мер. Следует регулярно проводить внутренние проверки для поддержания высокого уровня безопасности разрабатываемого ПО.

1.7 Описание действий, направленных на улучшение процессов, связанных с разработкой безопасного ПО

Улучшение процессов разработки безопасного программного обеспечения (ПО) на языке C++ требует комплексного подхода, который охватывает все стадии жизненного цикла разработки. Приведенное ниже руководство содержит описание действий, направленных на совершенствование этих процессов.

1. Интеграция принципов безопасной разработки в культуру компании.

   Проводите регулярные тренинги и семинары для разработчиков по вопросам информационной безопасности. Используйте реальные кейсы и примеры уязвимостей, чтобы показать важность безопасного программирования. Установите политику, согласно которой любая обнаруженная уязвимость должна быть немедленно устранена, независимо от её уровня критичности. Внедрите систему мотивации для разработчиков, которые активно участвуют в улучшении безопасности кода.

2. Автоматизация процессов.

   Интегрируйте инструменты статического анализа кода и тестирования безопасности в систему непрерывной интеграции. Это позволит автоматически проверять каждое изменение в репозитории на наличие уязвимостей и отклонять изменения, содержащие проблемы. Настройте автоматическую проверку и обновление используемых библиотек и компонентов на наличие последних версий, содержащих исправления безопасности. Внедрите системы мониторинга и оповещения, которые будут уведомлять разработчиков и администраторов безопасности о возникновении новых уязвимостей или инцидентов.

3. Применение лучших практик программирования.

   Переходите на использование современных стандартов C++. Современные версии языка содержат встроенные механизмы защиты от уязвимостей. Избегайте прямого обращения к памяти и заменяйте их на современные механизмы управления памятью. Всегда валидируйте пользовательский ввод. Это поможет предотвратить ряд уязвимостей.

4. Оптимизация процессов тестирования.

   Увеличьте покрытие кода юнит-тестами и интеграционными тестами. Особое внимание уделите функциям, связанным с безопасностью. Добавьте фаззинг в процесс тестирования для выявления скрытых уязвимостей, возникающих при обработке нестандартных входных данных. Организуйте регулярные сессии тестирования на проникновение с привлечением независимых экспертов.

5. Адаптация под новые угрозы.

   Постоянно следите за обновлениями отчётов о новых уязвимостях. Создайте механизм быстрого реагирования на новые угрозы, позволяющий оперативно вносить изменения в код и выпускать патчи. Регулярно обновляйте внутреннюю базу знаний по вопросам безопасности, включая лучшие практики, примеры уязвимостей и способы их устранения.

2 Анализ безопасности библиотеки «JSON for Modern C++»

Библиотека «nlohmann/json» (также известная как «JSON for Modern C++») представляет собой высокоэффективную библиотеку для работы с JSON-форматом данных в языке программирования C++. Она позволяет легко сериализовать и десериализовать объекты в формат JSON и обратно.

Основные особенности проекта:

1. Интуитивно понятный синтаксис: с помощью перегрузки операторов удалось добиться удобного и интуитивно понятного интерфейса на ровне с другими высокоуровневыми языками.

2. Простота интеграции: весь код состоит из одного заголовочного файла «json.hpp». У проекта нет ни внешних зависимостей, ни сложной системы сборки. Библиотека написана на C++11. Это не должно требовать изменений флагов компилятора или настроек проекта.

3. Высокая производительность: библиотека оптимизирована для высокой производительности при работе с большими объемами данных, несмотря на простоту интерфейса.

4. Кросс-платформенность: поддерживает сборка на Windows, Linux, macOS и другие.

5. Широкий функционал: Включает поддержку комментариев, сериализации пользовательских типов, UTF-8/UTF-16/UTF-32 кодировок и многое другое.

6. Открытый исходный код: проект имеет лицензию MIT, что делает его доступным для свободного использования в коммерческих и некоммерческих проектах.

2.1 Моделирование угроз безопасности информации

Для моделировании угроз безопасности информации будет использоваться формальная модель STRIDE, которая позволяет классифицировать угрозы по шести категориям: Spoofing (подмена личности), Tampering (несанкционированное изменение), Repudiation (отказ от ответственности), Information Disclosure (раскрытие информации), Denial of Service (отказ в обслуживании) и Elevation of Privilege (повышение привилегий).

Модель будет охватывать саму библиотеку и её взаимодействие с данными, которые она обрабатывает. Защищаемыми активами являются исходный код библиотеки, данные, обрабатываемые библиотекой, инфраструктура разработки и среда исполнения.

Классификация угроз по модели STRIDE:

1. Подмена личности

   (a) Утечка учетных данных разработчиков: если учетные данные разработчиков будут украдены, злоумышленники смогут выдавать себя за легитимных участников проекта и вносить изменения в исходный код.

(b) Подделка сертификатов: если сертификаты, используемые для защиты коммуникаций с внешними сервисами, будут подделаны, злоумышленники могут перехватывать и изменять данные.

2. Несанкционированное изменение

   (a) Изменение исходного кода: злоумышленники могут попытаться внести изменения в исходный код библиотеки, добавив вредоносный код или удалив важные элементы безопасности.

3. Отказ от ответственности

   (a) Отсутствие аудита: отсутствие должного аудита и журналирования действий может затруднить расследование инцидентов и установление виновных.

   (b) Невозможность доказать подлинность изменений: без надлежащей системы контроля версий и управления изменениями невозможно точно установить, кто и когда вносил изменения в код.

4. Раскрытие информации

   (a) Некорректная обработка данных: Некорректная обработка JSON-данных может привести к утечке конфиденциальной информации.

   (b) SQL-инъекции: Если библиотека используется для обработки запросов к базам данных, существует угроза SQL-инъекций, что может привести к раскрытию данных.

   (c) XSS (Cross-Site Scripting): Если данные, обрабатываемые библиотекой, отображаются в веб-интерфейсе, существует угроза XSS-атак, что может привести к раскрытию информации.

5. Отказ в обслуживании

   (a) Переполнение буфера: некорректная обработка больших объемов данных может привести к исчерпанию ресурсов и отказу в обслуживании.

   (b) Утечки памяти: утекаемая память может привести к исчерпанию ресурсов и отказу в обслуживании.

6. Повышение привилегий

   (a) Уязвимости в операционной системе: эксплуатация уязвимостей в операционной системе может позволить злоумышленникам повысить свои привилегии и получить доступ к чувствительным данным.

   (b) Неправильное управление правами доступа: недостаточное разграничение прав доступа может позволить злоумышленникам получить больше полномочий, чем им положено.

2.2 Сведения о проекте архитектуры программы

Архитектурно проект организован следующим образом:

1. Основной класс библиотеки.

   Центральным классом данной библиотеки является класс json: основной контейнер для хранения и обработки JSON-данных. Данный класс обеспечивает работу с различными типами данных: строками, числами, массивами, булевскими значениями и другими JSON-объектами. Интерфейсы для чтения и записи:

   – Для чтения JSON-данных используются методы parse() и from_json().

   – Для записи JSON-данных применяются методы dump() и to_json().

2. Сериализация и десериализация: библиотека предлагает удобные механизмы для автоматической сериализации и десериализации стандартных типов данных C++. Также поддерживается возможность расширения функциональности для пользовательских типов через перегрузку специальных функций.

3. Компиляция и интеграция: библиотека поставляется в виде одного заголовочного файла. Данный вид поставки библиотеки упрощает интеграцию в проекты за счёт отсутствия отдельной компиляции и линковки библиотеки.

4. Тестирование и документация: в репозитории содержится обширная база тестов. Документация также доступна в репозитории и включает в себя базовые примеры кода, руководство по использованию и справочную информацию по API библиотеки.

2.3 Используемые инструментальные средства

Ниже представлен список основных используемых инструментальные средств в проекте:

1. CMake – основной инструмент сборки проектов. Используется для создания кросс-платформенной среды разработки и сборки библиотеки. С помощью CMake можно генерировать разные системы сборки.

2. Mkdocs – инструмент для генерации документации из комментариев в исходном коде. Позволяет создавать HTML, LaTeX и PDF-документы с описанием классов, методов и функций.

3. GitHub Actions – система непрерывной интеграции, которая автоматически запускает тесты и проверки кода после изменения в проекте. Это помогает быстро выявлять ошибки и поддерживать высокое качество кода.

4. Doctest – один из основных фреймворков для тестирования кода на C++, используемый для написания юнит-тестов.

5. Clang-format – инструмент для автоматического форматирования кода.

Полный список используемых инструментальных средств можно найти в описании репозитория библиотеки.

2.4 Информация о прослеживаемости исходного кода программы к проекту архитектуры программы

В данной библиотеке простая структура проекта. Исходный код разбит на нескольких ключевых компонентов:

1. json.hpp – главный заголовочный файл, содержащий всю функциональность библиотеки. Этот файл включает определения основных классов и функций, таких как класс json, методы для сериализации и десериализации, а также вспомогательные классы и структуры.

2. detail – директория, содержащая реализацию внутренней логики библиотеки. Здесь находятся специализированные алгоритмы, утилиты и вспомогательные функции, которые не предназначены для прямого использования пользователями библиотеки.

3. tests – директория, содержащая тесты, написанными с использованием фреймворка doctest.

4. examples – директория, содержащая примеры использования библиотеки, демонстрирующие базовые сценарии работы с JSON-данными. Эти примеры помогают новым пользователям быстрее освоиться с функционалом библиотеки.

2.5 Порядок оформления исходного кода программы

Порядок оформления исходного кода программы определен и закреплен с помощью конфигурационного файла clang-format, находящегося в корне репозитория.

2.6 Сведения о результатах проведения экспертизы исходного кода программы

Сведений о результатах проведения внешних экспертных аудитов не содержится в репозитории, но можно отметить, что репозиторий имеет более 250 участников разработки (contributer). А также данная библиотека прошла проверку в Лаборатории Касперского и Apple.

Библиотека придерживается рекомендациям безопасной разработки от Core Infrastructure Initiative (CII).

2.7 Сведения о результатах проведения функционального тестирования программы

Библиотека имеет обширный набор тестов. Написаны они с использованием фреймворка doctest. Код библиотеки прошел тщательное модульное тестирование, имеет 100% покрытие кода, включая все исключительные ситуации.

2.8 Сведения о результатах проведения тестирования на проникновение

Сведения о результатах проведения тестирования на проникновения у библиотеки отсутствуют.

2.9 Сведения о результатах проведения динамического анализа кода программы

Для динамического анализа кода программы используются следующие инструменты: Valgrind и Clang code sanitizers. Данные инструменты помогают находить такого рода ошибки как утечки памяти, неопределенное поведение, использование неинициализированной памяти, состояние гонок, выходы за граница памяти и другого рода ошибок. В результате применений данных инструментов ошибок подобного рода не было выявлено.

Помимо динамического анализа кода библиотеки используются также и статический анализ: cppcheck и прочие утилиты. Оба вида анализа дополняют друг друга для обеспечения безопасности библиотеки.

2.10 Сведения о результатах проведения фаззинг-тестирования программы

Для фаззинг-тестирования в проекте используется Google OSS-Fuzz. Он дополнительно запускает фаззинг-тесты со всеми анализаторами в режиме реального времени. На данный момент корпус тестов насчитывает более миллиарда тестов.

2.11 Описание процедур отслеживания и исправления обнаруженных ошибок ПО и уязвимостей программы

Данная библиотека является с открытым исходным кодом, и для неё используются лучшие практики разработки программного обеспечения для отслеживания и исправления ошибок и уязвимостей.

Пользователи и разработчики могут сообщать об ошибках и уязвимостях несколькими способами:

1. С помощью issue tracker. Это основной способ уведомления разработчиков об ошибках и проблемах. Пользователь создает issue, описывая проблему, предоставляя минимальный воспроизводимый пример и, опционально, предложение по решению. Также пользователь может предложить свои изменения в проект, направление для решения этой ошибки или проблемы. Участники проекта тщательно будут рассматривать данные изменения и запрашивать соответствие требованиям проекта, по безопасности в том числе.

2. Если проблема связана с безопасностью, то рекомендуется использовать соответствующую функцию Github, чтобы сообщить о проблеме приватно. Это позволит команде разработчиков исправить уязвимость до публичного раскрытия информации.

После получения отчёта команда разработчиков анализирует серьезность ошибки или уязвимости, оценивает последствия возможных атак для пользователей, устанавливает приоритет задачи в зависимости от ее влияния на стабильность и безопасность библиотеки.

Создается патч, который устраняет проблему. Патчи проходят строгое код-ревью, чтобы гарантировать их корректность и отсутствие новых ошибок, а также проводится полное тестирование, включая регрессионное тестирование, чтобы убедиться, что

исправление не вызывает новых проблем. Все изменения документируются в журнале изменений.

Пользователям рекомендуется обновить свою версию библиотеки до последней доступной, чтобы получить исправления и улучшения.

2.12 Описание процедуры поиска разработчиком ПО уязвимостей программы

Процедура поиска уязвимостей в программном обеспечении требует систематического подхода и применения различных инструментов и методик. Рассмотрим основные шаги, которые разработчик может предпринять для выявления потенциальных уязвимостей:

1. Статический анализ кода — это процесс анализа исходного кода без выполнения программы. Он помогает находить потенциальные уязвимости и ошибки, такие как утечки памяти, переполнение буферов, небезопасные вызовы функций и другие проблемы, связанные с безопасностью.

   Инструменты: cppcheck, clang-tidy, svace, pvs-studio.

   Необходимо запустить анализ кода, проанализировать найденные предупреждения и определить, какие из них являются реальными проблемами, затем исправить обнаруженные уязвимости и повторить анализ для подтверждения их устранения.

2. Динамический анализ — выполняется во время выполнения программы и позволяет обнаружить уязвимости, которые сложно выявить статическим анализом, такие как ошибки времени исполнения, проблемы с обработкой ввода-вывода и др.

   Инструменты: valgrind, clang code sanitizers.

   Необходимо запустить исполнение программы с набором тестовых данных с включенными инструментами (для некоторых необходимо поменять опции компилятора и пересобрать проект), изучить отчёты, генерируемые инструментами, определить наличие потенциальных уязвимостей, исправить обнаруженные проблемы и повторно протестировать программу.

3. Фаззинг - это техника тестирования, при которой программа подвергается случайным или псевдослучайным входным данным для выявления неожиданных состояний и сбоев.

   Инструменты: AFL++, Google OSS-Fuzz.

   Необходимо определить точки входа для фаззинга, собрать корпус тестов, начальный набор данных, которые в последствие будет мутироваться, запустить фаззинг, обновлять корпус данных, на которых в программе обнаруживаются уязвимости или увеличивают покрытие корпуса, затем необходимо исправить обнаруженные уязвимости и провести дополнительное тестирования для подтверждения их устранения.

4. Тестирование на проникновение - это имитация атаки злоумышленника на систему с целью выявления слабых мест и уязвимостей.

Инструменты: Burp Suite, OWASP ZAP.

Необходимо определить цели тестирования, затем проанализировать результаты инструментов тестирования, определить, какие уязвимости были обнаружены, исправить найденные уязвимости и провести повторное тестирование для подтверждения их устранения.

2.13 Реализация и использование процедуры уникальной маркировки каждой версии ПО

Процедура уникальной маркировки каждой версии программного обеспечения реализована с использованием семантического версионирования и системы контроля версий Git.

Семантическое версионирование (SemVer) - это стандарт для управления номерами версий программного обеспечения. Версия ПО согласно этому стандарту имеет вид major.minor.patch, где:

1. major: увеличивается при добавлении несовместимых изменений;
2. minor: увеличивается при добавлении новых функций, не нарушающих совместимость;
3. patch: увеличивается при исправлении багов, не влияя на совместимость.

Каждая новая версия библиотеки помечается уникальным тегом в системе контроля версий Git. Теги создаются в соответствие описанному стандарту. Также каждой новой версии создается файл со списком изменений, внесенные в данную версию.

Выпуск новой версии состоит из следующих шагов: завершается разработка и тестирования всех изменений, определяется номер новой версии продукта, генерируется пакет для распространения по необходимости, оформляется список изменений, и затем эти изменения публикуются на платформе, в данном случае, Github, с тегом новой версии. В системах сборки можно указывать точную версию библиотеки, которую нужно использовать, это гарантирует, что проект будет собирать и использовать только ту версию, которая была протестирована и подтверждена.

Благодаря уникальному номеру версии и подробному журналу изменений, пользователи могут легко определить, какие изменения были внесены в каждую версию, и принять решение о необходимости обновления. Семантическое версионирование помогает пользователям понять, какие изменения вносятся в каждую версию, и оценить риск несовместимости при переходе на новую версию.

2.14 Использование системы управления конфигурацией ПО

Система управления конфигурацией охватывает систему контроля версий, построения и распространения.

Для данной библиотеки используется система контроля версий Git. Git позволяет разработчикам отслеживать изменения в кодовой базе, создавать ветки для параллельной разработки, объединять изменения и фиксировать состояние кода на определенных этапах. Для автоматизации процесса сборки и тестирования используется система непрерывной интеграции, основанная на GitHub Actions. Эта система позволяет автоматически запускать сборку и тестирование кода при каждом изменении в репозитории.

2.15 Меры, используемые для защиты инфраструктуры среды разработки ПО

В данном проекте используются следующие меры по защите инфраструктуры среды разработки ПО:

1. Управление доступом на Github происходит с помощью ограничений прав членов команды. Разработчики имеют разные уровни доступа в зависимости от своей роли (читатель, участник, администратор). Это помогает предотвращать случайные или злонамеренные действия со стороны отдельных участников.

2. Two-Factor Authentication (2FA): рекомендуется использовать двухфакторную аутентификацию для повышения безопасности учетных записей разработчиков.

3. Шифрование данных используется для SSH-ключей и для HTTPS. Это помогает обеспечить защиту передаваемых данных от прослушивания и несанкционированного доступа даже в случае компрометации инфраструктуры.

4. Аудит действий в GitHub: GitHub предоставляет подробные журналы аудита, которые регистрируют все действия, выполненные участниками проекта.