|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | |  | |  |  |  |
| **ДИПЛОМНАЯ РАБОТА**  **НА ТЕМУ:** | | | | | | |
| «МЕТОД И СИСТЕМА СТАТИЧЕСКОГО АНАЛИЗА ДЛЯ ПОИСКА УЯЗВИМОСТЕЙ В ПРОГРАММНОМ ОБЕСПЕЧЕНИИ» | | | | | | |
| Студент: | | | Егоров Александр Андреевич | | | |
|  | | | Фамилия, Имя, Отчество | | | |
|  |
| Студент: | | | Русакова Яна Анатольевна | | | |
|  | | | Фамилия, Имя, Отчество | | | |
|  |
| Студент: | | | Лазаренко Анастасия Юрьевна | | | |
|  | | | Фамилия, Имя, Отчество | | | |

2024 г.

**АННОТАЦИЯ**

Дипломная работа посвящена разработке метода и системы статического анализа для поиска уязвимостей в программном обеспечении. С увеличением числа кибератак и угроз безопасности информационных систем становится критически важным обеспечение безопасности программного обеспечения. В работе проведен обзор существующих методов и инструментов статического анализа, выявлены их преимущества и недостатки. На основе этого обзора разработан собственный метод статического анализа, учитывающий специфику поиска уязвимостей. Для реализации метода был написан код на языке Python, который прошел экспериментальное тестирование. Полученные результаты могут быть использованы для повышения уровня безопасности программного обеспечения и защиты информационных систем от киберугроз.

**ABSTRACT**

The work is devoted to the development of a static analysis method and system for finding vulnerabilities in software. With the increasing number of cyber-attacks and threats to the security of information systems, it becomes critical to ensure software security. The paper reviews existing static analysis methods and tools, identifies their advantages and disadvantages. Based on this review, we developed our own static analysis method that takes into account the specifics of vulnerability search. To implement the method, a Python code was written and experimentally tested. The results obtained can be used to improve software security and protect information systems from cyber threats.

ОГЛАВЛЕНИЕ

[**ОГЛАВЛЕНИЕ** 3](#_Toc168659139)

[**СПИСОК СОКРАЩЕНИЙ 4**](#_Toc168659140)

[**ВВЕДЕНИЕ** 5](#_Toc168659141)

[**ГЛАВА I. Статический анализ**. 7](#_Toc168659142)

[1.1. Что такое статический анализ? 8](#_Toc168659143)

[1.2. Происхождение инструментов SAST 9](#_Toc168659144)

[1.3. Процесс использования SAST 10](#_Toc168659145)

[1.4. Преимущества использования SAST 13](#_Toc168659146)

[**ГЛАВА II. Уязвимости и их классификация** 14](#_Toc168659147)

[2.1. Основные виды уязвимостей 14](#_Toc168659149)

[2.2. Ошибки обработки ввода данных 16](#_Toc168659150)

[Пример атаки SQL-инъекции: Heartland Payment Systems и Hannaford Brothers (2014) 18](#_Toc168659151)

[Пример атаки XSS (межсайтовый скриптинг): Twitter (2010) 19](#_Toc168659152)

[2.3 Переполнение буфера 21](#_Toc168659153)

[Пример атаки: Моррисов червь (1988) 23](#_Toc168659154)

[Пример атаки: Уязвимость в Adobe Flash Player (2015) 24](#_Toc168659155)

[2.3. Ошибки форматирования строк 26](#_Toc168659156)

[2.4. Примеры небезопасных функций и их безопасные альтернативы 27](#_Toc168659157)

[2.5. Выводы 28](#_Toc168659158)

[**ГЛАВА III. Сравнение существующих статических анализаторов** 30](#_Toc168659159)

[3.1. Разнообразие статических анализаторов 30](#_Toc168659161)

[**ГЛАВА IV Практическая часть. Написание кода**. 35](#_Toc168659162)

[4.1. Обзор разработки статического анализатора кода 35](#_Toc168659164)

[4.2. Архитектура и функциональность 35](#_Toc168659165)

[4.3. Хранение и обновление базы уязвимостей 37](#_Toc168659166)

[4.4. Реализация 38](#_Toc168659167)

[4.5. Использование скрипта 44](#_Toc168659168)

[**ЗАКЛЮЧЕНИЕ** 46](#_Toc168659169)

[**СПИСОК ЛИТЕРАТУРЫ** 48](#_Toc168659170)

СПИСОК СОКРАЩЕНИЙ

ИБ – информационная безопасность

SAST – Static Application Security Testing (Статическое тестирование безопасности приложений)

DAST – Dynamic Application Security Testing (Динамическое тестирование безопасности приложений)

IAST – Interactive Application Security Testing (Интерактивное тестирование безопасности приложений)

RASP – Runtime Application Self-Protection (Защита и анализ в реальном времени)

CVE – Common Vulnerabilities and Exposures (Общие уязвимости и воздействия)

API – Application Programming Interface (Интерфейс прикладного программирования)

MIT – Massachusetts Institute of Technology (Массачусетский технологический институт)

CERT – Computer Emergency Response Team (Команда реагирования на компьютерные инциденты)

PEP – Python Enhancement Proposal (Предложение по улучшению Python)

JSON – JavaScript Object Notation (Формат обмена данными)

ODT – Open Document Text (Текстовый документ открытого формата)

CI/CD – Continuous Integration/Continuous Deployment (Непрерывная интеграция/непрерывное развертывание)

ORM – Object-Relational Mapping (Объектно-реляционное отображение)

GitHub – Платформа для хостинга проектов и совместной разработки

ВВЕДЕНИЕ

В современном информационном обществе, где программное обеспечение играет ключевую роль во многих аспектах нашей повседневной жизни и бизнес-процессов, обеспечение его безопасности становится неотъемлемым условием для защиты конфиденциальности, целостности и доступности данных. Вместе с тем, быстрый темп развития технологий и внедрение новых функций в ПО порождают новые уязвимости, которые могут быть использованы злоумышленниками для атак и нарушения безопасности.

В этом контексте методы и системы статического анализа программного обеспечения для поиска уязвимостей становятся крайне актуальными. Данный подход к обнаружению уязвимостей позволяет проводить анализ исходного кода программы без ее фактического выполнения, выявляя потенциальные уязвимости и ошибки, которые могут быть использованы злоумышленниками для проведения атак.

**Целью** данной дипломной работы является разработка метода и системы статического анализа для эффективного обнаружения уязвимостей в программном обеспечении. Для достижения этой цели необходимо решить следующие **задачи**:

* Изучение понятия «Статический анализ», существующих методов и инструментов статического анализа программного обеспечения.
* Провести анализ уязвимостей и их классификацию.
* Провести анализ существующих методов статического анализа и инструментов для выявления уязвимостей.
* Реализация автоматизированного скрипта, который использует инструмент статического анализа для поиска уязвимостей.

**Актуальность** данной работы обусловлена необходимостью обеспечения безопасности программного обеспечения в условиях быстрого развития информационных технологий и увеличения числа кибератак на различные информационные системы. Разработка эффективных методов и инструментов статического анализа является важным шагом в направлении повышения уровня защиты программного обеспечения и предотвращения возможных киберугроз.

ГЛАВА I. Статический анализ.

В соответствии с анализом статистических данных, преобладающее большинство случаев нарушения безопасности программного обеспечения связано с использованием злоумышленниками известных программных ошибок. Это обстоятельство подчеркивает важность раннего выявления и устранения уязвимостей на этапе разработки программных продуктов для снижения рисков информационной безопасности.

Одним из ключевых методов повышения безопасности разрабатываемого ПО является применение специализированных технологий, предназначенных для выявления и анализа потенциальных угроз безопасности на ранних этапах разработки. Разработчикам доступен целый ряд инструментов, облегчающих этот процесс и позволяющих избежать внедрения уязвимостей в конечный продукт. Среди них можно выделить:

Статический анализ кода (далее – SAST, Static Application Security Testing) – метод анализа программного кода на предмет наличия потенциальных уязвимостей без его выполнения. SAST позволяет выявлять ошибки и уязвимости на уровне исходного кода, что обеспечивает возможность их исправления на ранних этапах разработки.

Динамический анализ безопасности (далее – DAST, Dynamic Application Security Testing) – метод, основанный на тестировании программного обеспечения в режиме выполнения с целью выявления уязвимостей, которые могут быть эксплуатированы злоумышленниками. DAST позволяет оценить безопасность приложения в реальных условиях его использования и выявить уязвимости, недоступные при статическом анализе.

Интегрированный анализ безопасности приложений (далее – IAST, Interactive Application Security Testing) – метод, который сочетает в себе возможности как статического, так и динамического анализа для выявления уязвимостей в реальном времени в процессе выполнения приложения. IAST позволяет обнаруживать уязвимости, которые могут проявиться только в конкретных сценариях использования приложения.

Защита и анализ в реальном времени (далее – RASP, Runtime Application Self-Protection) – технология, предназначенная для обеспечения безопасности приложения в режиме реального времени путем мониторинга его работы и реагирования на потенциальные угрозы безопасности непосредственно во время выполнения. RASP позволяет обнаруживать и предотвращать атаки на приложение на этапе их выполнения, что повышает эффективность защиты и снижает риски нарушения безопасности.

В предстоящей главе будет уделено особое внимание статическому анализу кода в контексте обеспечения информационной безопасности программного обеспечения. Основываясь на принципах SAST, исследование будет направлено на изучение эффективности и применимости данного метода для выявления и устранения уязвимостей на ранних стадиях разработки ПО.

* 1. Что такое статический анализ?

SAST представляет собой метод тестирования «белого ящика», который используется уже на протяжении многих лет в области информационной безопасности программного обеспечения. Этот подход позволяет разработчикам обнаруживать и анализировать уязвимости безопасности, заложенные в исходном коде приложения, на ранних этапах жизненного цикла разработки ПО, еще до его фактического выполнения. SAST основывается на статическом анализе программного кода без его запуска, что позволяет выявлять потенциальные уязвимости, такие как недостаточная проверка ввода данных, возможности переполнения буфера, использование уязвимых библиотек и другие, еще на этапе написания кода. Благодаря этому SAST способствует повышению безопасности разрабатываемого программного продукта и обеспечивает соответствие руководствам и стандартам кодирования, даже без фактического выполнения базового кода, что делает его важным инструментом в процессе обеспечения безопасности информационных систем. [1]

* 1. Происхождение инструментов SAST

Хронология инструментов SAST представляет собой важный аспект эволюции методов обеспечения безопасности программного обеспечения.

В самом начале исследований в области статического анализа исходного кода, начиная с появления компьютеров, прослеживается постоянный интерес к обеспечению безопасности программного обеспечения. Однако, именно в 1990-х годах внимание общественности было привлечено к техникам тестирования безопасности, применяемым в компьютерных системах. Особенно это проявилось в связи с развитием веб-технологий, таких как JavaScript и Flash, а также с появлением новых угроз, включая атаки на язык структурированных запросов SQL (SQL injection), первое публичное обсуждение которых состоялось в 1998 году.

Поворотным моментом в развитии инструментов безопасности стал 2000 год, когда начали активно развиваться динамические инструменты тестирования безопасности приложений. В этот период вендоры начали активно предлагать свои решения на рынке, что привело к значительному росту этой технологии. DAST до сих пор остается важной составляющей программ безопасности приложений.

Следующим этапом стал 2002 год, когда был сделан значительный шаг в развитии открытого анализа программного обеспечения (open-source analysis), который сегодня известен как анализ состава программного обеспечения (software composition analysis). В тот же год на рынок вышли первые коммерческие продукты для сканирования открытых пакетов, что стало важным шагом в обеспечении безопасности программного обеспечения.

Однако, именно в 2002 году появление инструментов статического анализа исходного кода (SAST Tools) стало ключевым событием в развитии области безопасности программного обеспечения. Эти инструменты в первую очередь ориентировались на анализ безопасности кода приложений.

В 2006 году аналитики отрасли сосредоточили свое внимание на пространстве безопасности приложений и технологиях сканирования безопасности приложений. К этому времени фокус сместился на различные аспекты безопасности, и появились новые технологии, такие как интерактивное тестирование безопасности приложений (IAST) и инструменты защиты приложений в реальном времени (RASP), предназначенные для выполнения тестирования безопасности приложений различными способами. Тем не менее, инструменты SAST остаются отправной точкой для эволюции этих инструментов и технологий.

Таким образом, история развития инструментов SAST свидетельствует о постоянном стремлении к улучшению безопасности программного обеспечения и приспособлению к изменяющимся условиям и угрозам в информационной среде. [2]

* 1. Процесс использования SAST

Процесс использования инструментов статического тестирования безопасности приложений включает в себя ряд ключевых этапов:

1. **Конфигурационный анализ**, представляющий собой процесс оценки того, насколько продукт соответствует потребностям пользователей и предполагаемому использованию. Этот этап включает в себя анализ функциональности продукта, его интерфейса и выявление потенциальных угроз безопасности. Инструменты статического тестирования безопасности приложений выполняют конфигурационный анализ для обеспечения безопасности продукта и предотвращения возможных последствий его неправильного использования.
2. **Семантический анализ** – это процесс извлечения смысла из текста, который может включать в себя выявление тем, идей и основных концепций. Несмотря на простоту в теории, реализация семантического анализа в практике является сложной задачей из-за сложности естественного языка и его многозначности. Люди могут использовать слова по-разному, и их значения могут меняться в зависимости от контекста предложения или абзаца. Инструменты статического тестирования безопасности приложений используют методы обработки естественного языка для анализа текста на семантическом уровне и обеспечения соответствия его заданным требованиям безопасности.
3. **Анализ потока данных,** направленный на изучение данных программы для определения используемых и неиспользуемых значений. Он помогает разработчикам обнаруживать ошибки, такие как неинициализированные переменные, которые могут привести к непредвиденным последствиям при выполнении программы. Инструменты статического тестирования безопасности приложений помогают разработчикам соблюдать стандарты кодирования, а также выявлять неинициализированные и неиспользуемые переменные с целью повышения безопасности и надежности программного обеспечения. [3]

Схема процесса использования SAST изображена на рисунке 1.

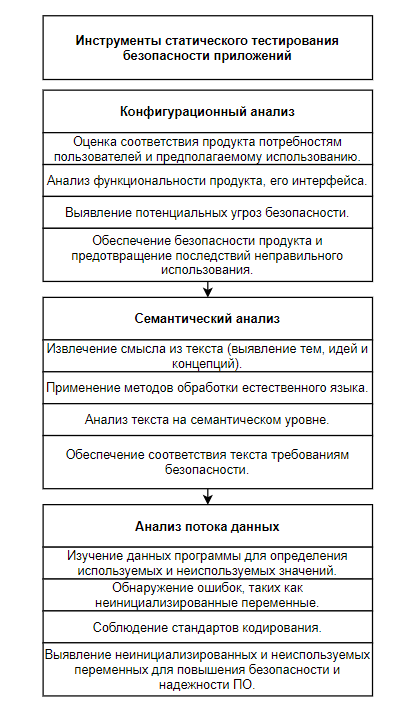


Рисунок 1 - схема работы SAST

* 1. Преимущества использования SAST

Инструменты SAST помогают тестировщикам программного обеспечения, предоставляя им больше информации о целевой программе, чем может дать динамическое тестирование. Одними из основных преимуществ SAST являются:

1. **Обнаружение дефектов.**

SAST-инструменты предоставляют возможность анализа исходного кода приложения с целью выявления потенциальных дефектов и уязвимостей, представляющих потенциальную угрозу для функциональной целостности и безопасности приложения. Путем направленного сканирования исходного кода, данные инструменты обеспечивают выявление проблем, недоступных для выявления другими методами динамического тестирования.

1. **Выявление рисков.**

SAST-инструменты выявляют скрытые или неожиданные риски в коде путем анализа его структуры и содержания. Преимущество заключается в том, что эти инструменты оперируют непосредственно с исходным кодом приложения, обходя ограничения, налагаемые компилятором, и изолируя потенциальные уязвимости, скрытые в рамках черного ящика API или других компонентов системы.

1. **Обеспечение точности.**

SAST-инструменты предоставляют возможность проведения комплексного сканирования исходного кода с высокой степенью точности и эффективности. Эти инструменты значительно превосходят человеческую способность в скорости анализа и обрабатывают миллионы строк кода за считанные минуты, обеспечивая тем самым высокий уровень достоверности исследования.

ГЛАВА II. Уязвимости и их классификация

1. 1. Основные виды уязвимостей

Когда требования к корректной работе программы нарушаются, возникает риск возникновения уязвимостей. Эти ошибки могут быть классифицированы как программные ошибки, так и их причины.

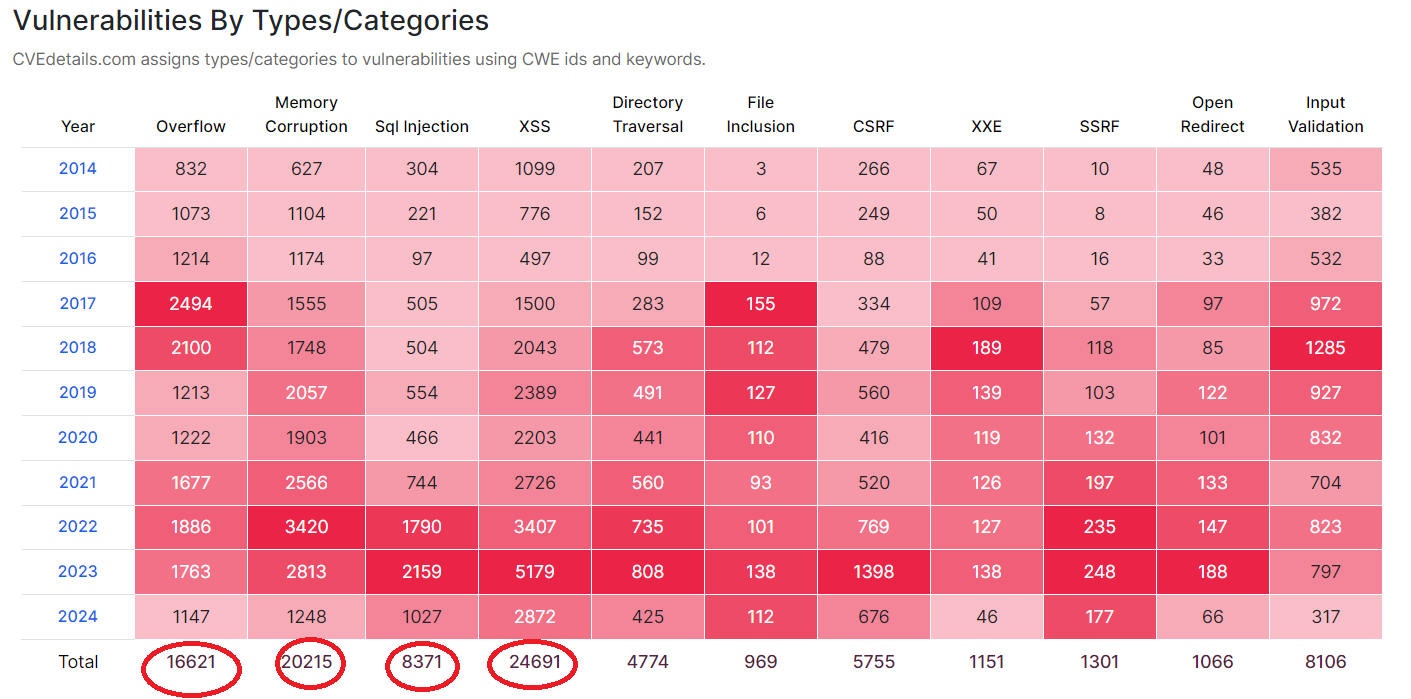


Рисунок 2 - классификация популярных уязвимостей.

Согласно CVE (Common Vulnerabilities and Exposures) - базе данных общеизвестных уязвимостей информационной безопасности [4] - по рисунку 2 можно сделать вывод, что уязвимости переполнения буфера, а также некорректной обработки ввода данных, повреждением памяти и форматирования строк стабильно являются наиболее популярными среди всех обнаруженных уязвимостей, поэтому далее рассмотрим их подробнее.

**1. Переполнение буфера.** Эта уязвимость возникает из-за отсутствия контроля за выходом за пределы массива в памяти во время выполнения программы. Когда слишком большой пакет данных переполняет буфер ограниченного размера, содержимое посторонних ячеек памяти перезаписывается, происходит сбой и аварийный выход из программы.

**Пример:** Переполнение буфера часто встречается в языках C и C++, когда программа записывает данные за пределы выделенной памяти. Классический пример - переполнение стека, где злоумышленник отправляет вредоносные данные, что приводит к перезаписи адреса возврата функции и выполнению вредоносного кода.

**Методы предотвращения:** Использование языков программирования, обеспечивающих безопасность памяти, таких как Rust или Java, или применение функций, которые контролируют размеры буферов, например, **strncpy** вместо **strcpy** в C.

**2. Уязвимость некорректной обработки ввода данных.** Уязвимости могут возникнуть из-за того, что данные, введенные пользователем без должного контроля, передаются на внешний язык интерпретатора. Таким образом, пользователь может задать входную информацию таким образом, что интерпретаторы будут выполнять совсем не те команды, которые были предусмотрены разработчиками программного обеспечения.

**Пример:** SQL-инъекции являются одним из наиболее известных видов этой уязвимости, когда злоумышленник может вставить вредоносный SQL-код через поля ввода данных. Примером может служить веб-форма, принимающая необработанный ввод и включающая его в SQL-запрос.

**Методы предотвращения:** Использование параметризованных запросов, валидация и фильтрация ввода, применение ORM (Object-Relational Mapping) для автоматической обработки SQL-запросов.

**3. Уязвимость некорректного форматирования строк.** Эта уязвимость возникает при недостаточном контроле параметров при работе с функциями форматного ввода-вывода (например, некоторых функций стандартной библиотеки языка С). Эти функции принимают в качестве одного из параметров символьную строку, определяющую формат ввода или вывода последующих аргументов функции, но при этом размер и характер переданных данных не анализируется.

**Пример:** Уязвимость форматирования строк в C возникает при использовании функции **printf** без указания правильного формата, что может привести к утечкам памяти или выполнению вредоносного кода.

**Методы предотвращения:** Избегать использования небезопасных функций и применять безопасные альтернативы, например, использовать **snprintf** вместо **sprintf**.

**4. Уязвимость повреждения памяти.** Уязвимости, связанные с повреждением памяти (Memory Corruption), возникают, когда программа случайно изменяет содержимое памяти, что может привести к неожиданному поведению или выполнению злоумышленником произвольного кода.

**Пример**: Уязвимость использования после освобождения (Use After Free) происходит, когда программа обращается к памяти после ее освобождения. Это может привести к выполнению произвольного кода.

**Методы предотвращения:** Инициализация переменных после освобождения (установление указателя в NULL), использование безопасных функций, таких как **strncpy()** вместо **strcpy()**, регулярный статический анализ кода для выявления ошибок памяти, применение инструментов, таких как Valgrind и AddressSanitizer, для выявления и устранения ошибок памяти, а также использование языков с автоматическим управлением памятью (например, Java или Python).

* 1. Ошибки обработки ввода данных

Одной из наиболее распространенных уязвимостей является некорректный ввод исходных данных. С одной стороны, может показаться, что в такой ошибке нет ничего важного, но на практике программистам следует обратить на это внимание, если программа не гарантирует, что введенные данные всегда правильно отформатированы, соответствуют стандартам, имеют смысл и соответствуют правилам кодирования. Источнику входных данных априори нельзя доверять, так как программа может получать входные данные из ненадежных источников или сам надежный источник может быть скомпрометирован.

**Что необходимо проверять:**

1. Проверьте все места ввода данных в программе.
2. Проверьте входные данные из всех источников, включая параметры командной строки, файлы конфигурации, запросы к базе данных, переменные среды, сетевые службы, системные свойства, временные файлы и любые другие внешние источники.
3. Храните надежные и ненадежные данные отдельно, чтобы гарантировать, что проверка входных данных всегда выполняется.

**Как необходимо проверять:**

1. Используйте самую сильную форму проверки входных данных, применимую в данном контексту.
2. Различайте проверку, которую приложение выполняет для удобства использования, с проверкой ввода для безопасности.
3. Отклоняйте данные, которые не проходят проверки. Не стоит корректировать данные и делать их более приемлемыми для дальнейшего использования.
4. Используйте слой абстракции вокруг важных или опасных операций, чтобы гарантировать, что проверки безопасности всегда выполняются и что опасные условия не могут возникнуть.
5. Проверьте входные данные по минимальной ожидаемой длине и максимальной ожидаемой длине.
6. Проверьте числовой ввод относительно максимального и минимального значения. Обратите внимание на операции и методы, которые работают с введенным числом, нарушится ли их поведение при выходящих за границы значениях.

Простой способ проверить входные данные с помощью статического анализа - указать инструменту все места, где программа принимает входные данные. Совокупность таких мест можно условно назвать поверхностью атаки приложения. В инструменте статического анализа целью атаки являются все точки входа в программу и вызов исходной функции, то есть набор функций, которые вызываются извне или выполняют пользовательский ввод в программу. В принципе, чем больше площадь атаки, тем больше внимания следует уделять проверке точности вводимых данных.

**Реальные примеры атак демонстрируют**, насколько важна правильная обработка входных данных.

Пример атаки SQL-инъекции: Heartland Payment Systems и Hannaford Brothers (2014)

Heartland Payment Systems и Hannaford Brothers стали жертвами одной из крупнейших атак SQL-инъекции, когда хакеры смогли проникнуть в их системы и получить доступ к миллионам записей о кредитных картах. Эта атака произошла в 2008 году и была связана с группой хакеров под руководством Альберто Гонсалеса. Хакеры использовали SQL-инъекцию для установки вредоносного кода на серверах компаний, что позволило им собирать данные о транзакциях и передавать их на внешние серверы. [5]

Хакеры воспользовались уязвимостями в веб-приложениях, которые не проверяли должным образом входные данные. Они ввели вредоносные SQL-запросы в поля ввода, которые были интерпретированы базой данных как легитимные команды. Это позволило злоумышленникам извлечь конфиденциальные данные, такие как номера кредитных карт, их владельцы и другие детали транзакций.

**Последствия:**

* Heartland Payment Systems была оштрафована на миллионы долларов и обязана провести масштабное обновление своей системы безопасности.
* Hannaford Brothers также понесла значительные финансовые потери и репутационные издержки.
* Сотни тысяч клиентов пострадали от утечки данных, что привело к многочисленным случаям мошенничества с кредитными картами.

**Методы предотвращения:**

* **Использование параметризованных запросов.**

Это позволяет отделить данные от команд SQL, что предотвращает внедрение вредоносного кода.

* **Валидация и фильтрация ввода.**

Все входные данные должны быть проверены на соответствие ожиданиям и отфильтрованы.

* **Регулярные аудиты безопасности.**

Проводить регулярные проверки кода и тестирование на уязвимости.

Пример атаки XSS (межсайтовый скриптинг): Twitter (2010)

В сентябре 2010 года Twitter стал жертвой массовой атаки XSS. Злоумышленники обнаружили уязвимость в обработке твитов, позволявшую вставлять JavaScript-код в сообщения. Этот код выполнялся в браузерах пользователей, когда они просматривали зараженные твиты. Атака началась рано утром и быстро распространилась, затронув тысячи пользователей.

Хакеры использовали уязвимость в обработке сообщений на Twitter, вставляя вредоносные скрипты в текст твитов. Когда пользователи просматривали эти твиты, скрипты автоматически выполнялись, что позволяло хакерам получать доступ к учетным записям пользователей, красть куки и выполнять действия от их имени. [7]

**Последствия данной атаки были следующие:**

* Атака вызвала значительные сбои в работе Twitter.
* Множество пользователей потеряли доступ к своим учетным записям или стали жертвами фишинга.
* Twitter пришлось временно отключить функциональность твитов и проводить масштабное исправление уязвимости.

Методы предотвращения подобных атак включают в себя:

* **Экранирование входных данных.**

Все вводимые пользователями данные должны быть экранированы, чтобы предотвратить выполнение кода.

* **Использование Content Security Policy (CSP).**

CSP помогает предотвратить выполнение непроверенного кода на веб-страницах.

* **Регулярные обновления и патчи.**

Обеспечение своевременного обновления систем безопасности и устранения выявленных уязвимостей.

Эти примеры подчеркивают важность правильной обработки входных данных и применения современных методов безопасности для предотвращения атак. SQL-инъекции и XSS могут привести к серьезным последствиям, включая финансовые потери и утрату доверия пользователей. Использование статических анализаторов, таких как SonarQube и ESLint, позволяет обнаруживать и устранять такие уязвимости на ранних этапах разработки, обеспечивая более высокий уровень безопасности программного обеспечения.

Пример кода для предотвращения SQL-инъекции на Python с использованием параметризованных запросов изображен на рисунке 3.

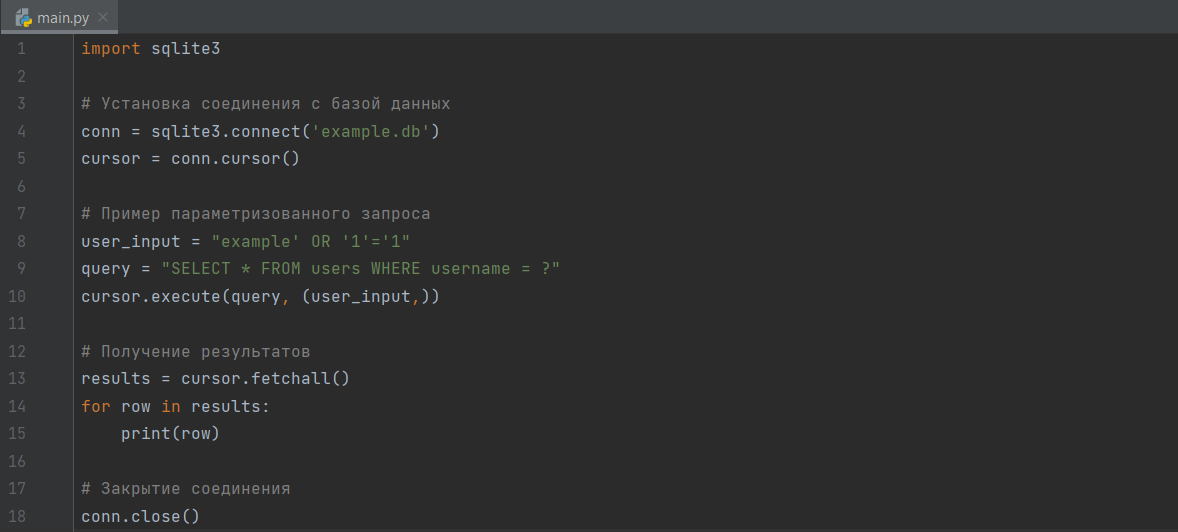


Рисунок 3 - пример кода для предотвращения SQL-инъекции на Python с использованием параметризованных запросов.

## 2.3 Переполнение буфера

Переполнение буфера происходит, когда программа записывает данные из выделенной памяти. Ошибка переполнения буфера позволяет злоумышленнику контролировать уязвимый код, то есть возможность перезаписывать значения в памяти в интересах злоумышленника.

Лучший способ предотвратить уязвимости, связанные с переполнением буфера, - это использовать язык программирования, гарантирующий безопасность памяти и типов. При написании программного кода на небезопасных языках, где наиболее широко используются C и C++, программист несет ответственность за предотвращение ненужных изменений в операциях с памятью. Операции, которые манипулируют памятью, могут привести к переполнению буфера, но на практике ошибки, которые чаще всего приводят к переполнению буфера, группируются вокруг ограниченного набора операций. В случае классической атаки с разрушением стека злоумышленник отправляет данные, содержащие сегмент вредоносного кода, уязвимой программе с переполнением буфера стека. В дополнение к вредоносному коду, злоумышленник содержит адрес памяти в начале кода. Когда происходит переполнение буфера, программа записывает данные злоумышленника в буфер и в конечном итоге преобразует обратный адрес функции в адрес запуска вредоносного кода.

Когда функция возвращается, она переходит к значению, сохраненному в ее адресе возврата. Поскольку адрес возврата был перезаписан, элемент управления переходит к буферу и начинает выполнять вредоносный код злоумышленника.

**Решение:** выбранный программистом подход к распределению памяти будет влиять на то, как вы используете статический анализ для выявления потенциальных переполнений буфера и других нарушений памяти в коде.

Механизмы статического распределения буфера обычно легче проследить и проверить. Поскольку буферы назначаются с фиксированным размером во время компиляции, анализатор может анализировать размеры буферов, используемых в различных операциях и идентифицировать ошибки. Кроме выхода за границы буфера, так же опасными ситуациями являются использование ресурса после его освобождения и двойное освобождение ресурса, т.к. они могут вызвать ошибки сегментации и потенциально могут привести к переполнению буфера.

Ошибки использования ресурса после освобождения возникают, когда программа продолжает использовать указатель после освобождения. Если освобожденная память уже перераспределена, злоумышленник может использовать указатель на память для запуска атаки переполнения буфера.

В качестве решения можно установить следующее правило для анализатора (рисунок 4), по которому после освобождения указателя, ему присваивалось значение NULL. Другое правило может искать вызовы функций освобождения памяти (например, функция delete в C++), а затем сообщает об ошибке, если следующая операция не устанавливает свободную переменную в NULL.

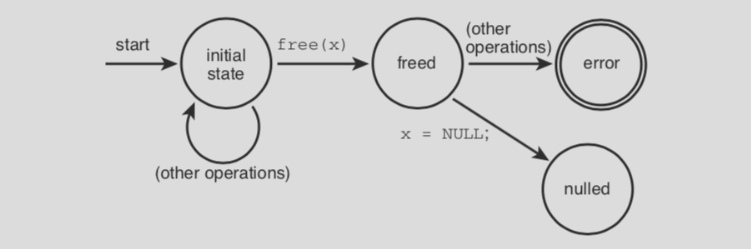


Рисунок 4 - Схематичное изображение правила работы с освобожденным ресурсом

Ошибки обработки ввода данных являются одной из самых распространенных уязвимостей, приводящих к серьезным последствиям, включая возможность выполнения произвольного кода и нарушения безопасности системы. Реальные примеры атак демонстрируют, насколько важна правильная обработка входных данных.

Пример атаки: Моррисов червь (1988)

Моррисов червь, созданный Робертом Таппаном Моррисом в 1988 году, стал одним из первых известных червей Интернета. Выпущенный из Массачусетского технологического института (MIT), червь быстро распространился и заразил около 6,000 из 60,000 компьютеров, подключенных к Интернету на тот момент. [7]

Червь использовал уязвимости в программах UNIX, таких как «finger» и демоны электронной почты, чтобы проникать в системы. Он был запрограммирован так, чтобы скрываться и распространяться автоматически.

**Последствия:**

* Системы в университетах и научно-исследовательских центрах, таких как Harvard, Princeton, Stanford и NASA, были серьезно нарушены.
* Стоимость ущерба оценивается в сотни тысяч до миллионов долларов.
* В результате инцидента был создан первый в стране центр реагирования на компьютерные инциденты (CERT).

**Методы предотвращения:**

* **Использование языков программирования с безопасностью памяти.**

Например, Rust и Go, которые автоматически управляют памятью и предотвращают переполнение буфера.

* **Безопасные функции для работы с буферами.**

Использование функций, которые контролируют размеры буферов, таких как strncpy вместо strcpy в C.

* **Инструменты статического анализа.**

Применение инструментов, таких как Coverity и PVS-Studio, для автоматического обнаружения уязвимостей.

### Пример атаки: Уязвимость в Adobe Flash Player (2015)

В 2015 году была обнаружена критическая уязвимость в Adobe Flash Player, связанная с переполнением буфера (CVE-2015-5122). Эта уязвимость позволяла злоумышленникам выполнять произвольный код на компьютере жертвы при посещении специально созданного веб-сайта.

Злоумышленники создавали веб-страницы с вредоносным Flash-контентом, который использовал уязвимость переполнения буфера в Flash Player. При загрузке такой страницы в браузере с включенным Flash Player, вредоносный код выполнялся, предоставляя злоумышленникам полный контроль над системой жертвы. [8]

**Последствия:** Уязвимость затронула миллионы пользователей по всему миру, включая корпоративные и государственные системы. Эксплойты для этой уязвимости были найдены в архивах Hacking Team, что ускорило их распространение и использование. Adobe выпустила экстренные обновления для устранения этой уязвимости, но многие системы оставались уязвимыми из-за задержек в обновлениях или использования устаревших версий программного обеспечения.

**Реакция и устранение:** Adobe выпустила уведомление безопасности и настоятельно рекомендовала пользователям отключить Flash Player до выхода исправлений. Эксплойт для уязвимости был быстро адаптирован для использования в популярных наборах эксплойтов, что сделало атаку еще более опасной.

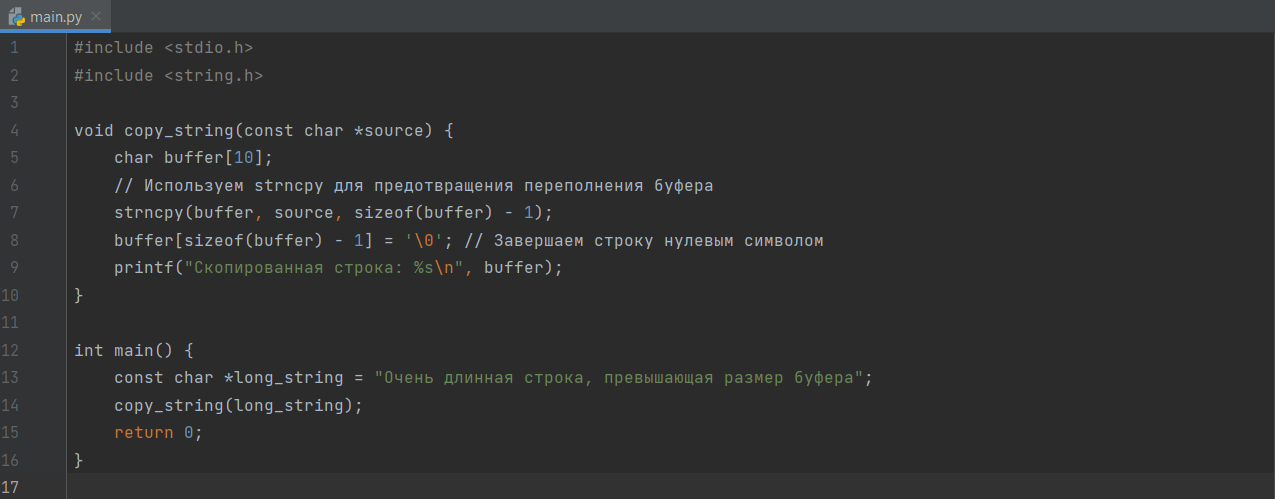
Пример кода с использованием безопасной функции изображен на рисунке 5.

Рисунок 5 - Пример кода с использованием безопасной функции.

* 1. Ошибки форматирования строк

Базовая структура данных строки в языке C (массив символов с нулевым символом в конце) подвержен ошибкам, а встроенные функции стандартной библиотеки для работы со строками только усугубляют ситуацию.

При использовании многих функций манипуляций со строками C легко допустить ошибку. Вместо того чтобы стараться быть с ними особенно осторожными, лучше всего вообще избегать их. В частности, стоит избегать использования таких функций манипуляции со строками как get(), scanf(), strcpy() или sprintf ().

Поведение метода get () довольно простое: функция читает из потока, на который указывает stdin, и копирует данные в буфер, пока не достигнет символа новой строки (\n). Соответственно, функция переполняет свой целевой буфер всякий раз, когда количество символов, считываемых из входного источника, превышает буфер, переданный в get (). Хотя функция scanf() немного сложнее, чем get(), она тоже уязвима, поскольку предназначена для чтения произвольного количества форматированных данных в один или несколько буферов фиксированного размера. Когда scanf() встречает спецификатор «%s» в строке формата, он считывает символы в соответствующий буфер до тех пор, пока не встретится значение, отличное от ASCII, что может привести к переполнению буфера, если в функцию передается количество данных больше, чем размер буфера. Если указан спецификатор ширины, например, «%255s», scanf() будет считывать до указанного количества символов в буфер. Из-за возможности ограничить количество читаемых входных данных функция может быть безопасно использована, если спецификатор формата правильно ограничивает объем считываемых данных. Такой метод использования scanf() более безопасный, однако данный метод подвержен другим ошибкам и корректность его работы сильно зависит от входных данных.

**Решение:** хорошей практикой является составление списка небезопасных функций, использование которых анализатор должен запрещать. Решение о том, какие именно функции должны быть запрещены, зависит от программы и зависит от таких факторов, как история базы кода, специфические для контекста риски, связанные с окружением, в котором используется программа и т.д.

Ошибки форматирования строк являются одной из распространенных причин уязвимостей в программном обеспечении. Неправильное использование функций, манипулирующих строками, может привести к таким проблемам, как переполнение буфера, что, в свою очередь, открывает путь для атак. В этой связи важно знать и избегать небезопасных функций, заменяя их безопасными аналогами. Ниже приведены примеры таких небезопасных функций и рекомендации по их безопасному использованию.

* 1. Примеры небезопасных функций и их безопасные альтернативы

1. **gets()**

**Проблема**: Не проверяет длину вводимой строки, что делает её уязвимой для переполнения буфера.

**Альтернатива**: **fgets()** – позволяет указать максимальное количество символов для чтения.



Рисунок 6 - пример корректного кода

2. **scanf()**

**Проблема**: Может переполнить буфер при большом вводе.

**Альтернатива**: Использование спецификатора ширины или комбинации fgets() и sscanf().

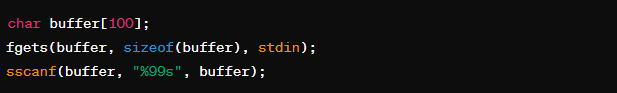


Рисунок 7 - пример корректного кода.

3. **strcpy()**

**Проблема**: Копирует строку без проверки размеров.

**Альтернатива**: **strncpy()** – ограничивает количество копируемых символов.

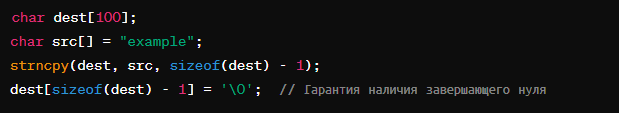


Рисунок 8 - пример корректного кода.

4. **sprintf()**

**Проблема**: Форматирует строку без ограничения размера.

**Альтернатива**: **snprintf()** – позволяет указать максимальный размер буфера.

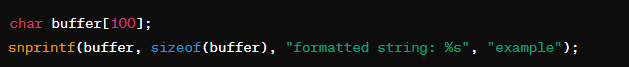


Рисунок 9 - пример корректного кода.

* 1. Выводы

В главе были рассмотрены основные классификации уязвимостей защиты, которые может предотвратить статический анализ, а именно: переполнение буфера, ошибки обработки ввода данных, ошибки форматирования строк. Хотя ошибкам обработки ввода данных и ошибкам форматирования строк зачастую не уделяется должного внимания, и данные ошибки считаются не критичными, они могут существенно повлиять на работоспособность вашей программы. В главе описано, какие части вашей программы могут содержать различные виды уязвимостей и как от них можно себя предостеречь.

Переполнение буфера до сих пор является актуальной и очень распространенной уязвимостью языков, без которых сложно представить разработку программного обеспечения в том числе и для встроенных систем. Поэтому пока сами разработчики данных языков не повысят их безопасность, наиболее эффективным решением является использование методов статического анализа.

ГЛАВА III. Сравнение существующих статических анализаторов

В современной разработке программного обеспечения важность обеспечения качества, надежности и безопасности кода невозможно переоценить. Программы становятся все более сложными и масштабными, и малейшая ошибка в коде может привести к серьезным последствиям, включая потерю данных, сбои в работе систем и даже угрозу безопасности пользователей. В таких условиях особое значение приобретает использование инструментов для анализа и проверки кода.

1. 1. Разнообразие статических анализаторов

На сегодняшний день существует множество различных статических анализаторов, каждый из которых имеет свои особенности и предназначен для разных языков программирования и типов проектов. Разбираться в разнообразии этих инструментов и понимать их отличия чрезвычайно важно для выбора наиболее подходящего решения для конкретного проекта. Например, SonarQube является универсальным инструментом, поддерживающим множество языков программирования и предоставляющим глубокий анализ качества кода. В то же время, ESLint специализирован для JavaScript и идеально подходит для фронтенд-разработки, предлагая гибкую настройку правил и интеграцию с редакторами кода.

Понимание особенностей и возможностей различных статических анализаторов позволяет разработчикам и командам эффективно использовать эти инструменты, улучшая качество и безопасность программного обеспечения. Выбор правильного анализатора для конкретного проекта может существенно ускорить процесс разработки, снизить количество ошибок и уязвимостей, а также повысить общую производительность и стабильность продукта.

Также, важно отметить, что статические анализаторы играют ключевую роль в современном процессе разработки программного обеспечения. Они позволяют разработчикам обнаруживать и устранять ошибки и уязвимости на ранних стадиях, обеспечивать соответствие стандартам кодирования, улучшать производительность и надежность кода, а также способствуют обучению и развитию профессиональных навыков. Разнообразие существующих статических анализаторов требует внимательного подхода к их выбору и использованию, что делает понимание их работы и отличий между собой важным аспектом для успешной разработки качественного и безопасного программного обеспечения.

В таблице 1 показаны самые популярные статические анализаторы по версии OWASP [9].

***Таблица 1 - сравнительный анализ популярных статических анализаторов***

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Название** | **Лицензия** | **Поддерживаемые платформы** | **Примечание** | **Плюсы** | **Минусы** |
| .NET Security Guard | Open Source | .NET, C#, VB.NET | Анализатор безопасности кода для .NET | Простота использования,специализирован для .NET | Ограниченная поддержка языков, менее гибкий чем некоторые конкуренты |
| SonarQube | Open Source | Многоязычная поддержка | Гибкость и интеграция с CI/CD | Поддержка множества языков, мощные метрики и отчеты | Требует настроек и ресурсов, возможны ложные срабатывания |
| Pylint | Open Source | Python | Проверка на соответствие стандартам PEP 8 | Глубокий анализ Python кода, настраиваемость | Может быть медленным на больших проектах |
| FindBugs/SpotBugs | Open Source | Java | Анализ байт-кода Java для поиска ошибок | Глубокий анализ байт-кода, множество встроенных правил | Ограничен только Java |
| ESLint | Open Source | JavaScript, Node.js | Настраиваемый инструмент для проверки стандартов кодирования | Высокая настраиваемость, поддержка модулей и плагинов | Может быть сложным в настройке для больших проектов |
| Brakeman | Open Source | Ruby on Rails | Анализатор уязвимостей для приложений на Ruby on Rails | Специализирован для Ruby on Rails, быстрый анализ | Ограничен только Ruby on Rails |
| Bandit | Open Source | Python | Комплексный анализатор уязвимостей для Python | Легкость интеграции, сосредоточен на безопасности | Специализируется только на безопасности, может не покрывать все типы ошибок |
| HCL AppScan | Commercial | Многоязычная поддержка | Интеграция с CI/CD, поддержка множества языков программирования | Широкая поддержка языков, мощные функции безопасности | Дорогой, сложность интеграции |
| Fortify | Commercial | Windows, Linux, MacOSX | Поддержка множества языков, бесплатная пробная версия | Глубокий анализ, хорошая интеграция с DevOps | Высокая стоимость, сложность в настройке |
| Checkmarx CxSAST | Commercial | SaaS, on-premises, Windows, Linux | Интеграция с CI/CD, поддержка множества языков | Высокая точность, гибкость, поддержка различных языков | Дорогой, требует обучения для эффективного использования |
| Coverity | Commercial | Windows, Linux, MacOSX | Анализатор статического кода для выявления критических дефектов | Высокая точность, интеграция с CI/CD | Дорогой, требует значительных ресурсов для запуска |
| Veracode | Commercial | SaaS, многоязычная поддержка | Комплексный анализатор безопасности кода | Облачное решение, широкая поддержка языков | Дорогой, возможны проблемы с конфиденциальностью данных |

Анализируя таблицу 1, можно сделать вывод, что среди статических анализаторов существует значительное разнообразие лицензий, поддерживаемых платформ и функциональных возможностей. Открытые решения, такие как SonarQube, Pylint и ESLint, предлагают гибкость и широкие возможности настройки, что делает их популярными среди небольших команд и open-source сообществ. Коммерческие инструменты, такие как Fortify, Checkmarx и Veracode, предоставляют более сложные функции анализа, но их стоимость может быть значительным фактором.

Разнообразие поддерживаемых платформ и языков программирования также важно при выборе статического анализатора. Инструменты, такие как SonarQube и Checkmarx, поддерживают множество языков и платформ, что делает их универсальными решениями для крупных проектов. В то же время специализированные анализаторы, такие как Pylint для Python и ESLint для JavaScript, предлагают более глубокий анализ для своих языков.

Статические анализаторы различаются по функциональности и интеграции с другими инструментами разработки. Например, SonarQube и HCL AppScan легко интегрируются с системами непрерывной интеграции и доставки (CI/CD), что позволяет автоматизировать процесс анализа кода. Инструменты, такие как CodeQL, предлагают уникальные возможности для создания запросов и анализа кода на уязвимости.

Каждый инструмент имеет свои плюсы и минусы. Открытые решения проще в настройке и использовании, но могут иметь ограниченные возможности. Коммерческие продукты предлагают продвинутые функции анализа и интеграции, но могут быть сложными и дорогостоящими. Специализированные анализаторы предоставляют более глубокий анализ для своих языков, но могут быть менее гибкими в многоязычных проектах.

Исходя из этого, можно сделать вывод, что выбор статического анализатора зависит от типа проекта, используемых языков, требований к безопасности и качества кода, а также бюджета команды. Разнообразие доступных инструментов позволяет найти подходящее решение для любой ситуации, будь то небольшой open-source проект или крупное корпоративное приложение. Для упрощения выбора, OWASP предоставляет на своем сайте таблицу с существующими анализаторами, их характеристиками и отличиями, что помогает ориентироваться в разнообразии доступных инструментов.

ГЛАВА IV Практическая часть. Написание кода.

1. 1. Обзор разработки статического анализатора кода

В условиях растущей важности кибербезопасности и защиты данных, разработка инструментов для автоматического выявления уязвимостей в программном обеспечении становится критически важной задачей. В данной работе мы представляем обзор процесса создания статического анализатора уязвимостей, написанного на языке Python. Целью данного проекта было создание утилиты, способной автоматически анализировать исходный код программ на наличие потенциальных уязвимостей и предоставлять рекомендации по их устранению.

**Выбор инструментов и технологий.**

Для реализации проекта были выбраны следующие инструменты и технологии:

* **Python** - язык программирования, выбранный за его удобство и широкие возможности для быстрой разработки.
* **Модули requests, docx, odfpy** - использованы для загрузки файлов с **GitHub** и формирования отчетов в различных форматах.
* **GitHub** - платформа для хранения файлов с описаниями уязвимостей.
* **Регулярные выражения** - для выявления уязвимостей в коде.
  1. Архитектура и функциональность

Проект состоит из нескольких ключевых компонентов:

1. **Загрузка зависимостей**.

Чтобы обеспечить автоматическую установку необходимых библиотек, был добавлен этап проверки и установки зависимостей. Это позволяет пользователю без дополнительной подготовки запускать скрипт на любой машине.

1. **Загрузка файлов с GitHub**.

Все файлы с описаниями уязвимостей хранятся на GitHub. Скрипт автоматически загружает соответствующие файлы при запуске, используя библиотеку **requests.**

1. **Определение языка программирования**.

Скрипт автоматически определяет язык программирования исходного файла на основе его расширения. Это позволяет применять соответствующие правила анализа для каждого языка.

1. **Анализ кода**.

Для анализа кода используются регулярные выражения, которые сопоставляются с шаблонами уязвимостей, описанными в JSON файлах. При обнаружении уязвимости формируется отчет с рекомендациями.

1. **Формирование отчетов**.

Результаты анализа сохраняются в нескольких форматах: Word (**.docx**), ODT и текстовый файл. Все файлы отчетов упаковываются в архив для удобства пользователя.

* 1. Хранение и обновление базы уязвимостей

На GitHub хранятся JSON файлы с примерами уязвимостей для следующих языков программирования. Количество примеров уязвимостей в файлах 25+:

* **Python**: файл **vulnpython.json**
* **PHP**: файл **vulnphp.json**
* **C**: файл **vulnc.json**
* **C#**: файл **vulnsharp.json**

Эти JSON файлы содержат шаблоны уязвимостей и соответствующие им рекомендации по исправлению. Содержимое данных файлов благодаря возможностям github дает базе постоянно обновляться, чтобы обеспечить актуальность и полноту проверки. Благодаря этому пользователи анализатора могут быть уверены, что их код проверяется на соответствие новейшим угрозам безопасности.

Пример JSON файла с примерами уязвимостей для C# изображен на рисунке 9.

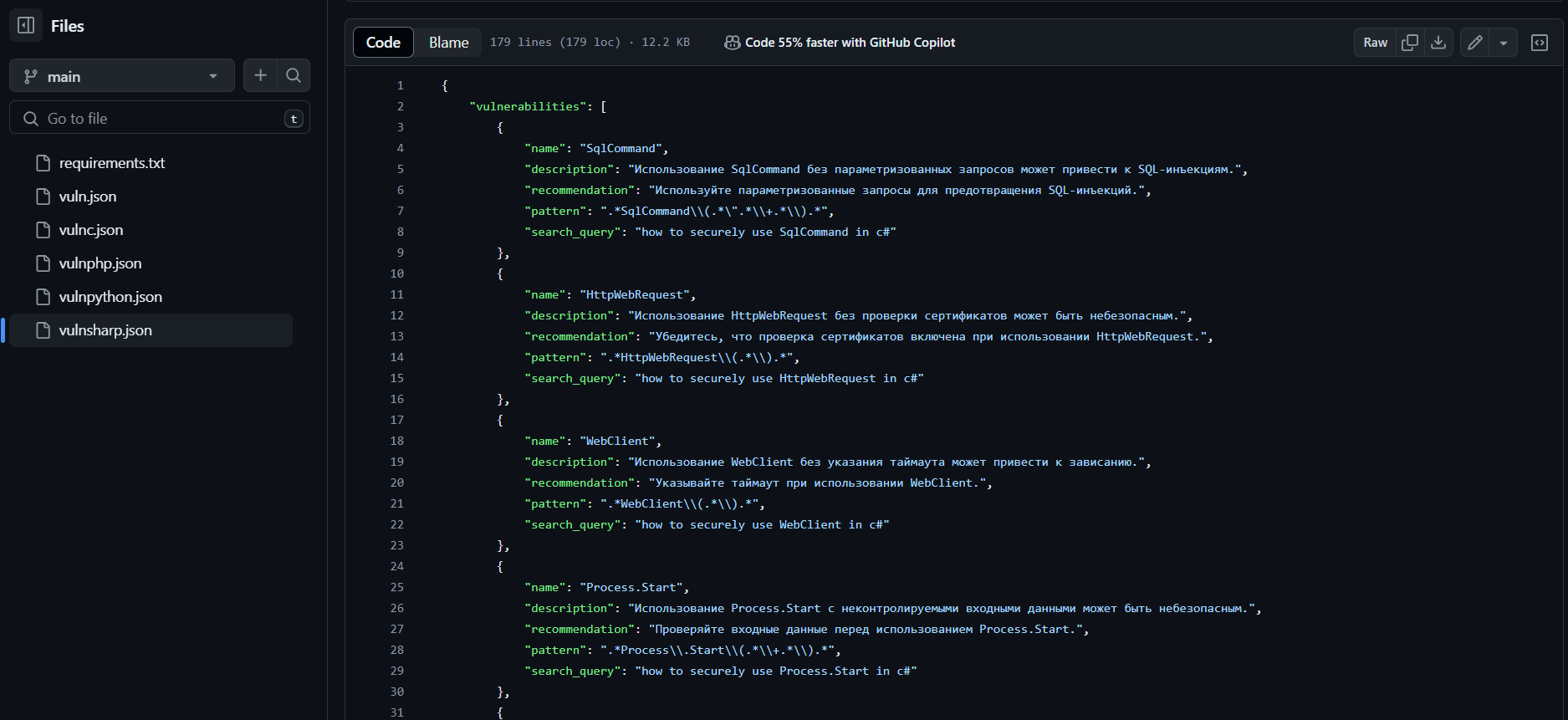


Рисунок 10 - содержимое файла vulnsharp.json

* 1. Реализация

Ключевым этапом в реализации было создание скрипта **analyzer.py**, который включает в себя все перечисленные выше компоненты. Ниже приведен код с комментариями, описывающими основные шаги и функциональные блоки:

Первым шагом скрипт проверяет наличие необходимых библиотек (python-docx, odfpy, requests) на системе. Если какая-либо из этих библиотек не установлена, скрипт автоматически загружает файл requirements.txt с GitHub и устанавливает все необходимые зависимости. Это позволяет пользователю запускать скрипт без предварительной настройки окружения. Для этого используется функция **install\_requirements().**



Рисунок 11 - часть кода

Скрипт анализирует расширение файла, переданного в качестве аргумента, чтобы определить, на каком языке программирования написан исходный код. Это необходимо для выбора соответствующего файла с описанием уязвимостей. Функция **detect\_language()** возвращает название языка на основе расширения файла.

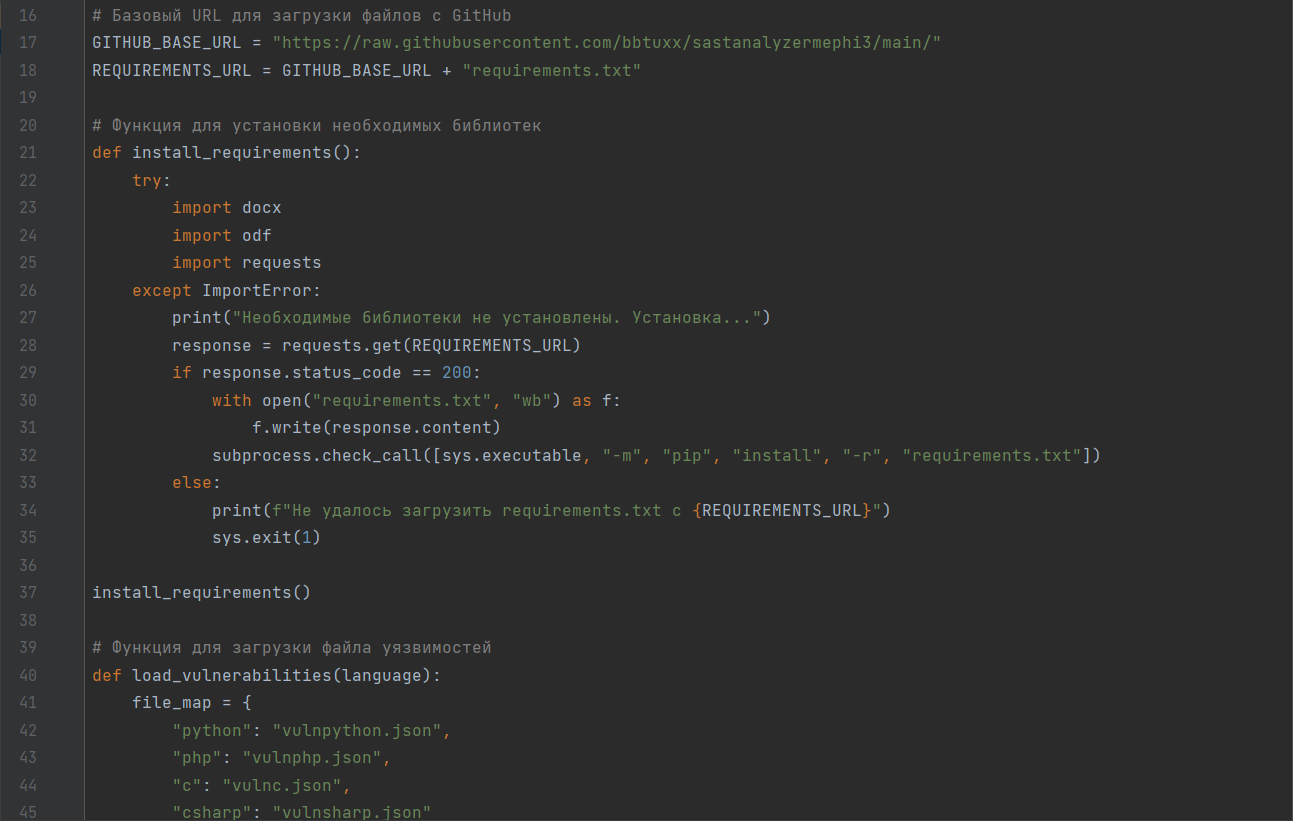


Рисунок 12 - часть кода

После определения языка программирования, скрипт загружает соответствующий JSON файл с описанием уязвимостей с GitHub. Для этого используется функция **load\_vulnerabilities()**, которая формирует URL для загрузки файла и обрабатывает ответ от сервера.

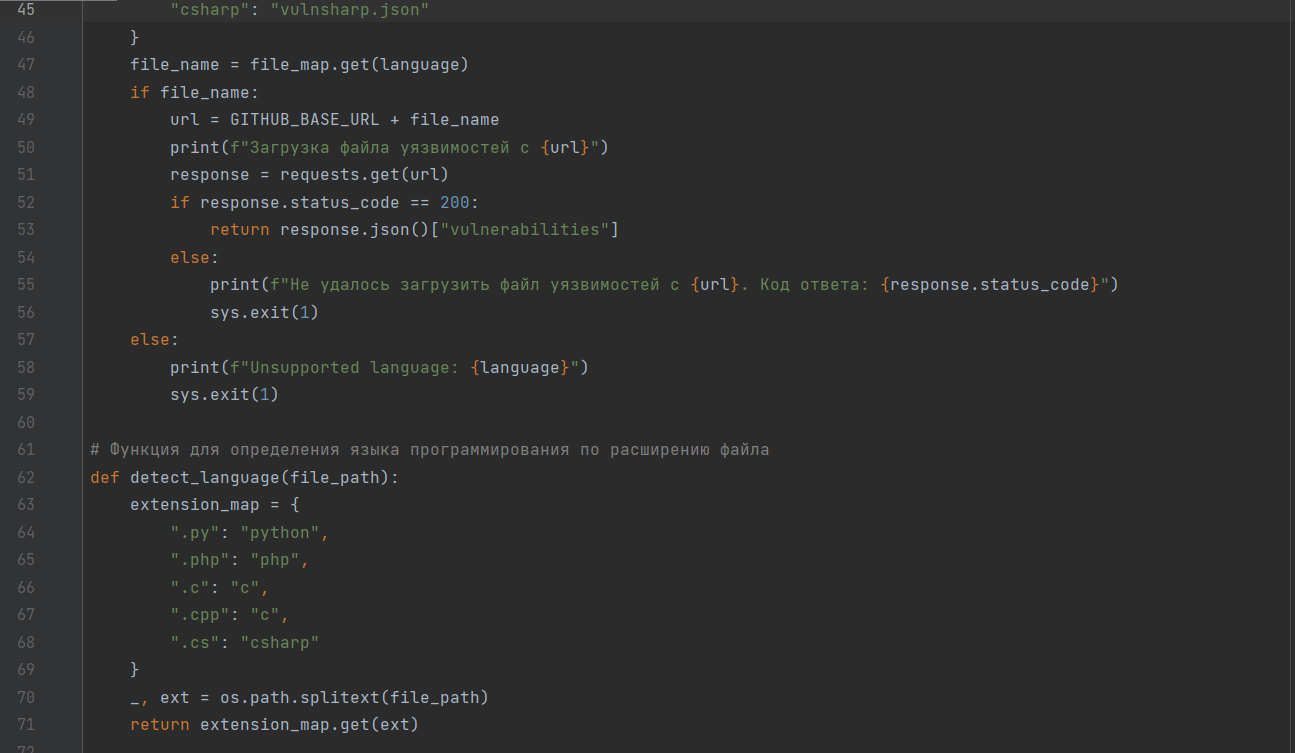


Рисунок 13 - часть кода

Основная функция анализа кода **analyze\_code()** читает исходный файл построчно и проверяет каждую строку на соответствие шаблонам уязвимостей, описанным в загруженном JSON файле. Если найдено совпадение, создается запись с информацией об уязвимости, включающая описание, рекомендацию и URL для поиска дополнительной информации.

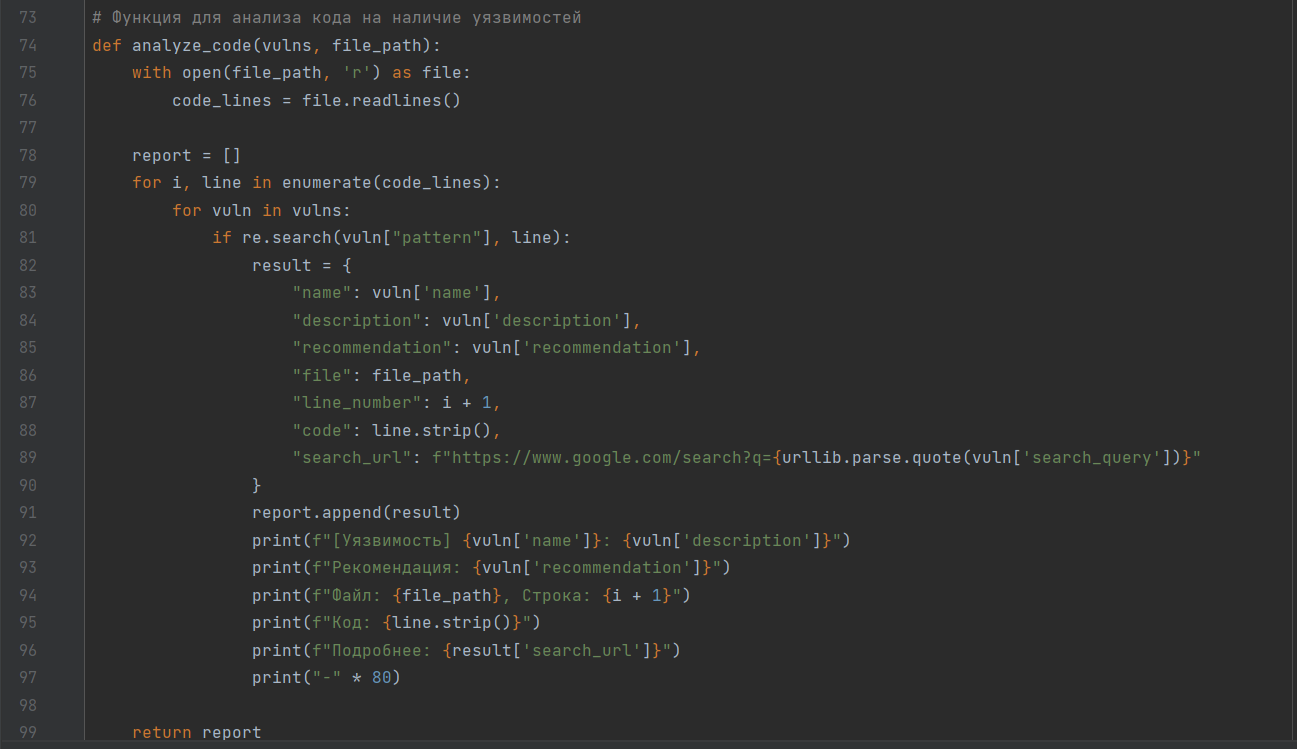


Рисунок 14 - часть кода

После завершения анализа кода, результаты сохраняются в трех форматах: **Word (.docx)**, **ODT** и текстовый (.**txt)** файл. Каждая функция сохранения создает файл отчета, содержащий найденные уязвимости и рекомендации по их устранению. Эти файлы затем упаковываются в ZIP архив для удобства пользователя.

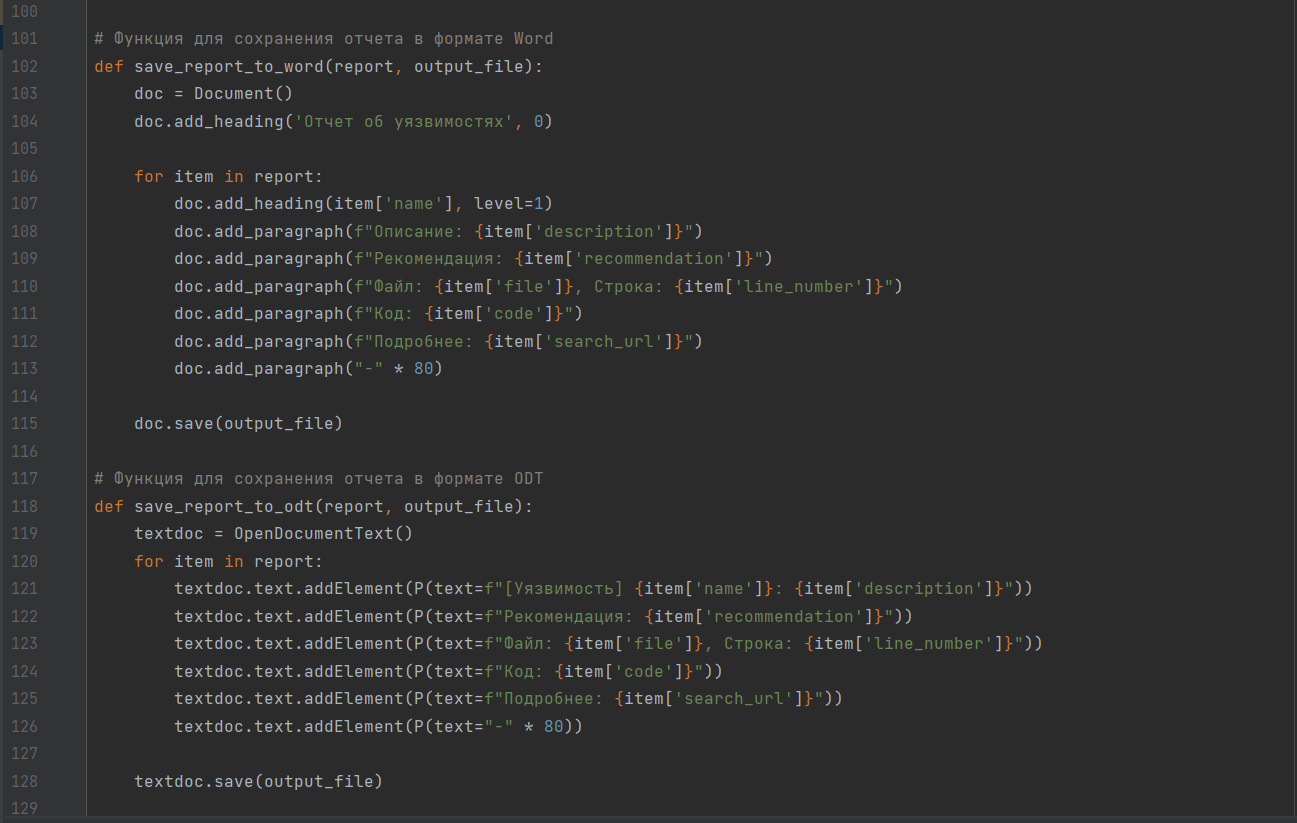


Рисунок 15 - часть кода



Рисунок 16 - часть кода

Основная логика выполнения скрипта заключается в последовательном вызове описанных функций: установка зависимостей, определение языка программирования, загрузка файла уязвимостей, анализ кода и формирование отчетов. Скрипт принимает один аргумент командной строки — путь к файлу, который необходимо проанализировать.

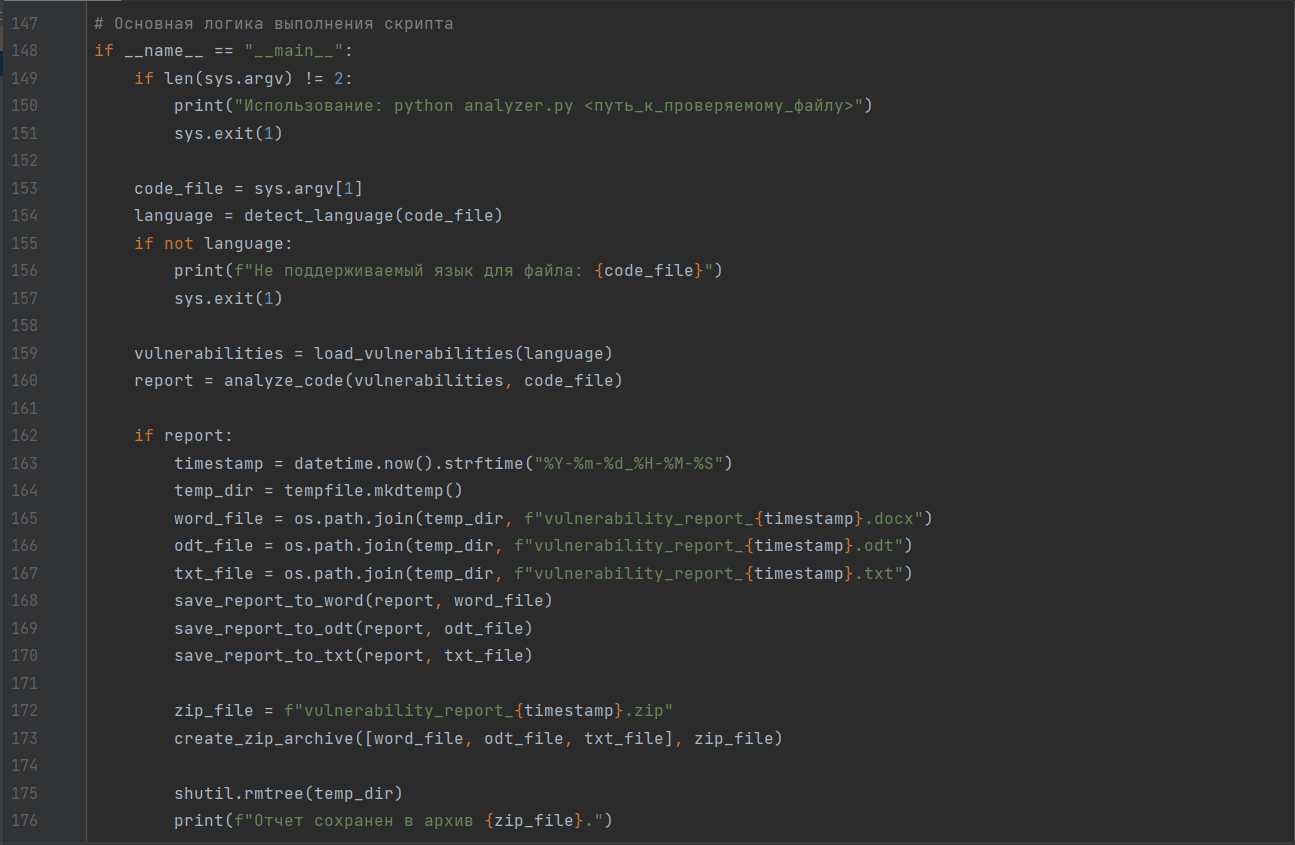
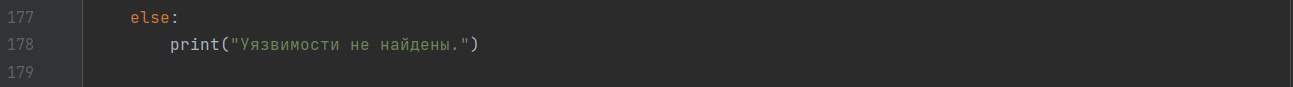


Рисунок 17 - часть кода



Результатом работы стал функциональный и универсальный инструмент для статического анализа исходного кода, способный выявлять широкий спектр уязвимостей в коде на различных языках программирования. Данный анализатор позволяет автоматизировать процесс обнаружения уязвимостей, что способствует повышению безопасности программного обеспечения и снижению рисков, связанных с эксплуатацией уязвимостей.

* 1. Использование скрипта

Для запуска скрипта необходимо воспользоваться командной строкой и выполнить следующую команду: **python3 analyzer.py <путь\_к\_проверяемому\_файлу>**

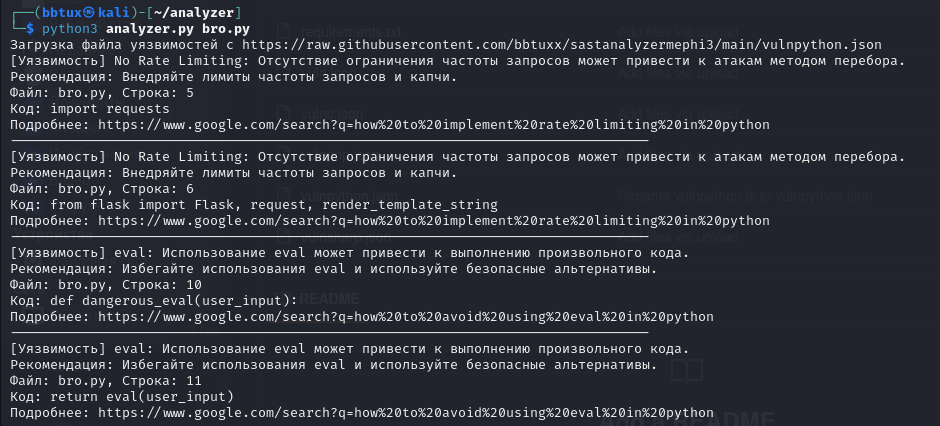


Рисунок 18 - активация скрипта

При обнаружении уязвимости скрипт не только предоставляет описание проблемы и рекомендации по её устранению, но и формирует ссылку на поиск в Google с заранее сформированным запросом. Это позволяет быстро получить дополнительную информацию и найти примеры решения проблемы.

По завершении анализа скрипт сформирует отчеты в форматах Word, ODT и текстовый файл, упакует их в ZIP архив и сохранит в текущей директории.

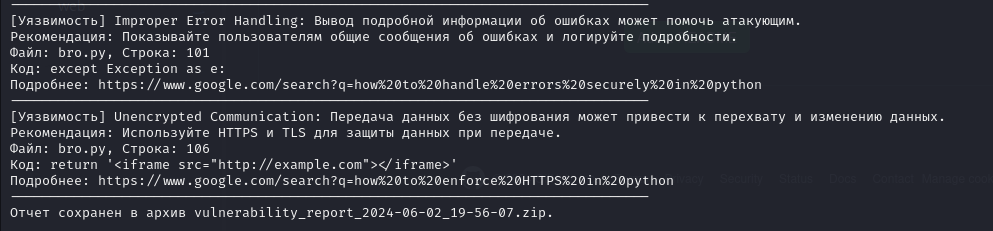


Рисунок 19 - пример сохранения отчета об уязвимостях

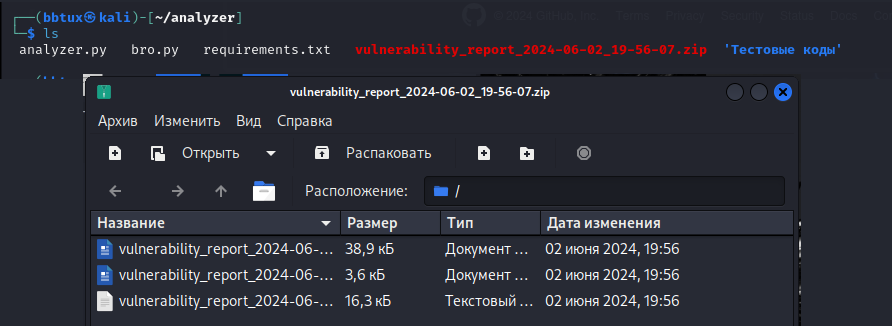


Рисунок 20 - пример сохранения отчета об уязвимостях

ЗАКЛЮЧЕНИЕ

В данной дипломной работе были рассмотрены и реализованы методы статического анализа для поиска уязвимостей в программном обеспечении. Проведенный анализ показал, что применение инструментов SAST является эффективным способом обеспечения безопасности программного обеспечения на ранних этапах его разработки.

Во-первых, была проведена детальная классификация уязвимостей, которая включала такие виды, как переполнение буфера, ошибки обработки ввода данных и ошибки форматирования строк. Анализ показал, что наиболее распространенными и критическими уязвимостями являются именно эти категории, что подтверждается статистическими данными из базы данных CVE.

Во-вторых, был проведен сравнительный анализ существующих инструментов статического анализа, таких как SonarQube, ESLint, Pylint, и других. Результаты этого анализа показали, что каждый из инструментов имеет свои преимущества и недостатки в зависимости от конкретного языка программирования и требований проекта​​.

В-третьих, была разработана и реализована система статического анализа, которая включает в себя автоматизированный скрипт на языке Python для анализа исходного кода и формирования отчетов. Этот скрипт использует регулярные выражения для поиска уязвимостей и может быть легко адаптирован для различных языков программирования за счет использования JSON-файлов с описаниями уязвимостей​​.

Созданный инструмент статического анализа имеет важное практическое значение. Он позволяет разработчикам автоматизировать процесс выявления уязвимостей, что существенно повышает эффективность и безопасность разработки программного обеспечения. Инструмент был протестирован на реальных примерах и показал точность в обнаружении различных типов уязвимостей, что подтверждает его применимость в условиях реальной разработки.

Дальнейшее развитие данного исследования может включать несколько направлений. Во-первых, улучшение алгоритмов анализа и внедрение машинного обучения для повышения точности и быстроты обнаружения уязвимостей. Во-вторых, расширение базы данных уязвимостей и добавление поддержки новых языков программирования, что сделает инструмент более универсальным и мощным. В-третьих, интеграция системы статического анализа с другими инструментами обеспечения безопасности, такими как динамическое тестирование и защита приложений в реальном времени, позволит создать комплексную систему безопасности программного обеспечения.

Таким образом, проведенное исследование подтвердило актуальность и необходимость разработки методов статического анализа для обеспечения безопасности программного обеспечения. Созданная система показала свою эффективность и применимость на примерах. Будущие исследования и разработки в этой области позволят повысить уровень защищенности информационных систем и противодействовать растущему числу кибератак.

СПИСОК ЛИТЕРАТУРЫ

* + - 1. Static Application Security Testing [Электронный ресурс] // Synopsis: [сайт]. [2020]. URL: https://www.synopsys.com/glossary/what-is-sast.html (дата обращения: 12.05.2024).
      2. The Past, Present And Future Of Static Application Security Testing Tools [Электронный ресурс] // techdogs.com: [сайт]. [2022]. URL: https://www.techdogs.com/td-articles/curtain-raisers/the-past-present-and-future-of-static-application-security-testing-tools (дата обращения: 10.05.2024).
      3. The architecture of SAST tools: An explainer for developers [Электронный ресурс] // githubblog: [сайт]. [2024]. URL: https://github.blog/2024-02-12-the-architecture-of-sast-tools-an-explainer-for-developers/ (дата обращения: 10.05.2024).
      4. Vulnerabilities By Types/Categories [Электронный ресурс] // https://www.cvedetails.com/: [сайт]. [2024]. URL: https://www.cvedetails.com/vulnerabilities-by-types.php (дата обращения: 18.05.2024).
      5. SQL injection attacks led to Heartland, Hannaford breaches [Электронный ресурс] // COMPUTERWORLD: [сайт]. [2009]. URL: SQL injection attacks led to Heartland, Hannaford breaches (дата обращения: 20.05.2024).
      6. The Twitter hack: how it started and how it worked [Электронный ресурс] // TheGuardian: [сайт]. [2010]. URL: https://www.theguardian.com/technology/blog/2010/sep/21/twitter-hack-explained-xss-javascript (дата обращения: 18.05.2024).
      7. Червь Морриса [Электронный ресурс] // Wikipedia: [сайт]. [1988]. URL: https://ru.wikipedia.org/wiki/
      8. В Adobe Flash Player обнаружена очередная критическая уязвимость [Электронный ресурс] // habr: [сайт]. [2015]. URL: https://habr.com/ru/companies/eset/articles/262319/ (дата обращения: 19.05.2024).
      9. Source Code Analysis Tools [Электронный ресурс] // OWASP: [сайт]. [2023]. URL: https://owasp.org/www-community/Source\_Code\_Analysis\_Tools (дата обращения: 20.05.2024).