**Учебный центр**

**OOO «Скилфэктори»**

совместно с

**Федеральное государственное автономное**

**образовательное учреждение высшего образования**

**Национальный исследовательский ядерный университет «МИФИ».**

Выпусканая квалифирационная работа

на тему:

**“Метод и система статического анализа для поиска уязвимостей в программном обеспечении”**

Программа дополнительного профессионального образования

“Информационная безопасность”

**Выполнили студенты MIFIIB**

**2 поток:**

Иванов Александр

Миронов Никита

Зимодра Юлия

**Научный руководитель:**

Серов Сергей

Москва, 2024

Оглавление

[ВВЕДЕНИЕ 3](#_Toc162978790)

[ГЛАВА 1 Введение в статический анализ 4](#_Toc162978791)

[1.1 Понятие и принципы работы статического анализа 4](#_Toc162978792)

[1.2 Преимущества и ограничения статического анализа 7](#_Toc162978793)

[1.3 Основные задачи статического анализа 8](#_Toc162978794)

[1.4 Выводы 8](#_Toc162978795)

[ГЛАВА 2. Методы статического анализа 9](#_Toc162978796)

[2.1 Анализ потока данных (Data-Flow Analysis) 9](#_Toc162978797)

[2.2 Анализ потока управления (Control-Flow Analysis) 9](#_Toc162978798)

[2.3 Символьное выполнение (Symbolic Execution) 10](#_Toc162978799)

[2.4 Аннотирование методов (Method Annotations) 10](#_Toc162978800)

[2.5 Сопоставление с шаблоном (Pattern-based analysis) 10](#_Toc162978801)

[2.6 Выводы 10](#_Toc162978802)

[ГЛАВА 3. Виды уязвимостей 11](#_Toc162978803)

[3.1 Уязвимости ввода данных (Input Validation Vulnerabilities) 11](#_Toc162978804)

[3.2 Уязвимости аутентификации и авторизации (Authentication and Authorization Vulnerabilities) 11](#_Toc162978805)

[3.3 Уязвимости управления сеансами (Session Management Vulnerabilities) 11](#_Toc162978806)

[3.4 Уязвимости безопасности веб-приложений (Web Application Security Vulnerabilities) 11](#_Toc162978807)

[3.5 Уязвимости внедрения кода (Code Injection Vulnerabilities) 12](#_Toc162978808)

[3.6 Уязвимости внешних зависимостей (External Dependencies Vulnerabilities) 12](#_Toc162978809)

[3.7 Вывод 12](#_Toc162978810)

[ГЛАВА 4 Системы идентификации и каталогизации уязвимостей в ПО 13](#_Toc162978811)

[4.1 Common Vulnerabilities and Exposures (CVE) 13](#_Toc162978812)

[4.2 Common Weakness Enumeration (CWE) 14](#_Toc162978813)

[4.3 Разница между CVE и CWE 15](#_Toc162978814)

[4.4 Вывод 16](#_Toc162978815)

[ЗАКЛЮЧЕНИЕ 18](#_Toc162978816)

[СПИСОК ИСПОЛЬЗОВАННОЙ ЛИТЕРАТУРЫ 19](#_Toc162978817)

[Приложение 20](#_Toc162978818)

# ВВЕДЕНИЕ

Программное обеспечение используется повсюду: для автоматизации процессов на предприятиях, оптимизации торговли и передачи информации между людьми. Однако без соответствующего обеспечения безопасности корректное функционирование данных систем невозможно. Стоит обратить внимание на то, что большая часть деятельности, которая происходит под видом компьютерной безопасности, вовсе не связана с решением проблем безопасности. Скорее с тем, как навести порядок там, где эти проблемы уже существуют. Антивирусные программы, системы исправления ошибок и системы обнаружения вторжений - все это средства, с помощью которых восполняются недостатки в безопасности программного обеспечения. Соответственно, больше усилий прилагается для компенсации плохой безопасности, вместо того, чтобы изначально создавать безопасное программное обеспечение. Хотя наиболее популярные и опасные для безопасности программы уязвимости давно известны, их количество с каждым годом не уменьшается, а программисты, как правило, повторяют одни и те же ошибки. Удобным средством для их предотвращения и устранения и является статический анализ как метод поиска общих ошибок безопасности в исходном коде. Термин статический анализ относится к любому процессу оценки кода без его выполнения, то есть он позволяет рассмотреть большое количество различных сценариев работы программы, в том числе в труднодоступных состояниях, которые могут быть сложными для осуществления и тестирования. Статический анализ необходимо использовать так же, когда речь идет о разработке встроенных систем для использования в аэрокосмической, автомобильной, медицинской, ядерной, железнодорожной и других отраслях. Цена ошибки в программном обеспечении в этих областях может быть очень высока, поэтому разработчики, которые пишут код для встроенных систем, должны придерживаться стандартов производительности и безопасности, которые превосходят стандарты большинства других отраслей. Ни один инструмент или техника никогда не обеспечит решение всех проблем безопасности, но с помощью грамотного использования средств статического анализа можно исправлять ошибки безопасности на раннем этапе, минимизировать количество ошибок в программе, повысить ее отказоустойчивость.

**Актуальность вопроса:** с каждым годом количество кибератак и инцидентов безопасности, связанных с эксплуатацией уязвимостей в программном обеспечении, продолжает расти. В связи с этим, разработчики программного обеспечения, а также предприятия и организации, использующие программные продукты, сталкиваются с необходимостью обеспечения высокого уровня безопасности приложений. Статический анализ выгодно отличается от других методов поиска уязвимостей тем, что может обнаруживать потенциальные проблемы без выполнения программы, что позволяет выявлять даже самые скрытые и трудноуловимые уязвимости.

В настоящее время индустрия информационной безопасности активно развивается, внося новые методики и инструменты статического анализа, а также улучшая существующие системы. Это подчеркивает актуальность темы и необходимость постоянного совершенствования методов и систем статического анализа для обеспечения надежной защиты программного обеспечения от потенциальных угроз.

# ГЛАВА 1

**ВВЕДЕНИЕ В СТАТИЧЕСКИЙ АНАЛИЗ**

## 1.1 Понятие и принципы работы статического анализа

Статический анализ кода — это процесс выявления ошибок в исходном коде программы до ее выполнения. Общую структуру обработки данных при статическом анализе можно увидеть на рисунке 1.

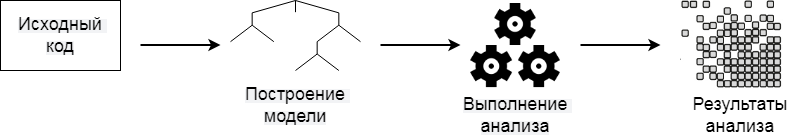


Рисунок 1 - Общая структура обработки данных при статическом анализе

Часто встречается рассуждение о том, что динамический анализ кода или тесты на проникновение могут заменить статический анализ, поскольку эти методы проверки выявят реальные проблемы и ложных срабатываний не будет. Однако стоит обратить внимание на то, что динамический анализ, в отличие от статического, не проверяет весь код, а проверяет только устойчивость программного обеспечения к набору атак, которые имитируют действия злоумышленника. Злоумышленник же может оказаться изобретательнее проверяющего вне зависимости от того, кто выполняет проверку: человек или машина. Динамический анализ будет полным только в том случае, если выполняется на полном тестовом покрытии, что в применении к реальным приложениям — трудновыполнимая задача. Доказательство полноты тестового покрытия — алгоритмически неразрешимая задача. Почти все статические анализаторы так или иначе построены по принципу компиляторов, то есть в их работе присутствуют этапы преобразования исходного кода — такие же, какие выполняет компилятор. Можно выделить следующие шаги в работе статического анализатора:

1. Разбор

1.1. Лексический анализ (текст программы считывается строка за строкой, разбивается на лексемы: зарезервированные слова, идентификаторы и константы)

• выделение лексем;

• определение, какой группе лексем принадлежит данная лексема;

• передача результатов дальше, чтобы понять, правильно ли составлены предложения;

1.2. Синтаксический анализ (определяется, правильно ли составлены предложения из слов)

• проверка на наличие синхронизирующих лексем: текст до синхронизирующей лексемы считается правильным;

• локальная коррекция (происходит автоматическое добавление синхронизирующих лексем);

• расширение грамматики ошибками (исходная грамматика дополняется неправильными выражениями и затем в нужный момент предлагается вариант исправления ошибки);

1.3. Контекстный анализ (проверяется, допустимо ли использование слов в данном контексте)

2. Применение некоторых правил

Как правило, лексический и синтаксический анализаторы строятся на основе формализованного описания лексики и синтаксиса языка. Лексика языка (правила выделения лексем) может быть описана в виде регулярных выражений или грамматик, синтаксис всегда описывается с помощью формальных грамматик (или их графического изображения в виде схем). В общем случае грамматики состоят из правил преобразования последовательностей символов в другие последовательности. Грамматики одинаково помогают решать задачи как программистов, использующих язык, так и создателей компиляторов/анализаторов для данного языка:

• грамматика предоставляет точную и достаточно легкую для понимания синтаксическую спецификацию языка программирования;

• для некоторых классов грамматик можно автоматически сконструировать эффективный анализатор, который определяет, является ли исходная программа синтаксически правильной;

• компиляторы, разработанные на базе грамматик, могут быть достаточно легко расширены (это особенно полезно для добавления новых конструкций, появившихся в результате развития языка);

Собственно, порядок применения правил грамматики для разбора входных данных будет называться деревом разбора (рисунок 2).

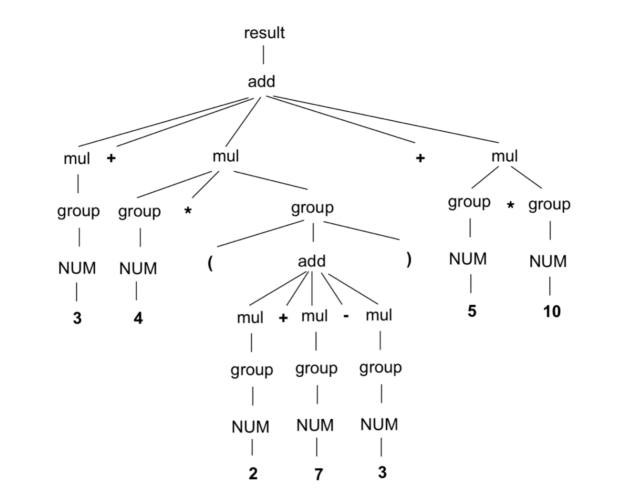


Рисунок 2 - Пример дерева разбора арифметического выражения «3+4\*(2+7-3)+5\*10»

Дерево разбора, как результат синтаксического анализа, удобно представлять в виде абстрактного синтаксического дерева – AST (рисунок 3).

Это позволяет оптимизировать некоторые участки на более позднем этапе генерации объектного кода, так же язык может иметь очень много ключевых слов, однако AST показывает только значимые компоненты, отражающие смысл программы, поэтому оно лишено семантических подробностей, мешающих отображению сути программы.

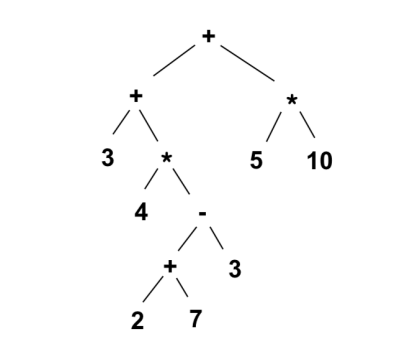


Рисунок 3 - Пример AST-дерева арифметического выражения «3+4\*(2+7-3)+5\*10»

В качестве узлов в AST-дереве выступают операторы, к которым присоединяются их аргументы, которые в свою очередь также могут быть составными узлами.

## 1.2 Преимущества и ограничения статического анализа

Преимущества статического анализа:

• Раннее обнаружение ошибок (во время разработки);

• Код при этом может не собираться и не запускаться;

• Полное покрытие кода;

• Не требуется задавать тестовые входные данные;

• Лучше проверяет участки кода, сложные для тестирования (обработка граничных ситуаций);

Однако статический анализ не является заменой тестирования, поскольку не проверяет логику программ.

Для статического анализа есть ряд ограничений:

• Отсутствие знаний о входных значениях (неопределённые значения переменных).

• Экспоненциальный рост числа путей в программе.

• Большая длина отдельных путей в программе (циклы, рекурсивные вызовы).

• Сложность анализа циклов и рекурсивных вызовов.

• Большой объём данных описывающих состояния программы (возможные значения).

## 1.3 Основные задачи статического анализа

**1. Задача доказательства корректности с пользовательскими аннотациями.**

Для каждой функции известны пред-, пост-условия и, возможно, инварианты циклов. Задача анализатора заключается в доказательстве соблюдения всех пред-,пост-условий и отсутствия ошибок времени выполнения. Допускается наличие ложных срабатываний, гарантируется отсутствие пропуска ошибок. Основная сложность разработки заключается в поиске оптимального соотношения между временем работы и минимизацией количества программ, которые являются корректными, но при этом не прошли проверки анализатора (т.е. ложных срабатываний).

**2. Задача доказательства корректности для подмножества языка.**

Анализу подвергаются специальные программы, которые не используют некоторые возможности языка. Задача анализатора заключается в доказательстве отсутствия ошибок времени выполнения. Возможно наличие ложных срабатываний, гарантируется отсутствие пропуска ошибок. Основная сложность разработки также заключается в поиске оптимального соотношения между временем работы и минимизацией количества программ, которые являются корректными, но при этом не прошли проверки анализатора.

**3. Задача доказательства корректности без ограничений.**

Никаких дополнительных ограничений на анализируемые программы не накладывается. Задачей статического анализа является доказательство отсутствия некоторых классов ошибок времени выполнения. Гарантируется отсутствие как ложных срабатываний, так и пропусков ошибок, однако анализ подвержен зацикливанию. Основная сложность разработки заключается в минимизации классов программ, на которых анализ зацикливается.

**4. Задача поиска дефектов.**

Никаких дополнительных ограничений на анализируемые программы не накладывается. Задача анализа — поиск потенциальных ошибок времени выполнения. Допускается присутствие ложных срабатываний и пропусков ошибок. Сложность разработки заключается в обнаружении практически значимого класса ошибок при сохранении низкого процента ложных срабатываний.

## 1.4 Выводы

В главе были рассмотрены принципы работы статического анализа: каким образом происходит лексический и синтаксический анализ, как во время синтаксического анализа исходный код преобразовывается в структуру данных «дерево разбора» и затем представляется в виде АSТ-дерева. Также были перечислены задачи, которые решает статический анализ, его преимущества и недостатки, которые накладывают на анализ исходного кода некоторые ограничения.

# ГЛАВА 2

МЕТОДЫ СТАТИЧЕСКОГО АНАЛИЗА

Существует множество различных методов статического анализа, в частности, анализ с обходом дерева кода, анализ потока данных, анализ потока данных с выбором пути и т. д. Конкретные реализации этих методов различны в различных анализаторах. Однако независимо от языков программирования анализаторы могут использовать один и тот же базовый код (инфраструктуру). Этот базовый код содержит набор основных алгоритмов, которые могут использоваться в разных анализаторах кода вне зависимости от предоставляемых задач и анализируемого языка. Набор поддерживаемых методов и конкретная реализация этих методов, опять же, будет зависеть от конкретной инфраструктуры. Рассмотрим основные методы анализа данных.

## 2.1 Анализ потока данных (Data-Flow Analysis)

Под анализом потока данных понимают совокупность задач, нацеленных на выяснение некоторых глобальных свойств программы, то есть извлечение информации о поведении тех или иных конструкций в некотором контексте. Соответственно, такой тип анализа кода проверяет наличие проблемных конструкций на основе наборов правил, причем инструменты анализа потока также моделируют пути принятия решений, что позволяет углубиться в приложение и сильно расширить возможности поиска дефектов, таких как нулевые указатели, переполнение буфера и другие дефекты безопасности. Существует несколько классических задач анализа потока данных.

**Задача 1: Достижимые определения**

Задачу можно сформулировать следующим образом: для каждого вхождения переменной требуется определить множество присваиваний, такое, что для каждого из них существует путь, в котором между ним и данным вхождением отсутствуют другие присваивания той же переменной. Другими словами задача достижимых определений заключается в выяснении, где именно устанавливаются значения того или иного вхождения данной переменной.

**Задача 2: Живые переменные**

В задаче требуется для каждой вершины графа потока управления построить множество переменных, обладающих следующим свойством: существует путь через данную вершину, начинающийся присваиванием данной переменной и кончающийся ее использованием, не содержащий иных присваиваний той же переменной. Решения данных задач применяются при оптимизации кода программы.

## 2.2 Анализ потока управления (Control-Flow Analysis)

С помощью данного метода удобно исследовать различные пути выполнения программы, которые могут иметь место, когда присутствуют вызовы функций, не возвращающих управление, исключения, а также используются ключевые слова break, continue (если говорить о языке С/C++) и т.д. Чтобы сделать эти алгоритмы эффективными, большинство инструментов строят график потока управления поверх АСТ. Узлы в графе потока управления являются основными блоками: последовательностями команд, которые всегда будут выполняться, начиная с первой инструкции и заканчивая последней командой, без возможности пропуска каких-либо инструкций. Ребра в графе потока управления имеют направление и представляют собой потенциальные пути управления потоком между базовыми блоками.

## 2.3 Символьное выполнение (Symbolic Execution)

Символьное выполнение позволяет вычислять значения переменных, которые могут приводить к ошибкам, производить проверку диапазонов значений. Зная предполагаемые значения переменных, можно выявлять такие ошибки как:

• утечки памяти;

• переполнения;

• выход за границу массива;

• разыменование нулевых указателей в C++ / доступ по нулевой ссылке в C#;

• бессмысленные условия;

• деление на 0;

## 2.4 Аннотирование методов (Method Annotations)

Вместо того, чтобы рекурсивно проверять корректность работы каждого метода, можно задать аннотации для методов языка, то есть какие-то характеристики и ограничения на входные параметры, которые помогут в поиске ошибок. Можно составлять аннотации, самостоятельно анализируя тело функции. Так приходится делать, когда нельзя получить доступ к телу функции или тело функции описано в модуле, который собирается динамически и т.д. Для популярных библиотек создатели статических анализаторов составляют ручные аннотации методов.

## 2.5 Сопоставление с шаблоном (Pattern-based analysis)

Этот метод достаточно прост в реализации. На основе дерева разбора (или АСТ-дерева) применяется определенный паттерн для определенных типов узлов. С помощью этого метода удобно искать ошибки Copy Paste.

## 2.6 Выводы

В главе были рассмотрены основные методы статического анализа исходного кода, а именно: анализ потока данных, анализ потока управления, символьное выполнение, аннотирование методов, сопоставление с шаблоном.

# ГЛАВА 3

ВИДЫ УЯЗВИМОСТЕЙ

Типология уязвимостей в программном обеспечении может быть разнообразной и включать различные категории в зависимости от характера и причин возникновения уязвимости. Вот несколько основных типов уязвимостей:

## 3.1 Уязвимости ввода данных (Input Validation Vulnerabilities)

Эти уязвимости возникают, когда программа не правильно обрабатывает внешние данные, вводимые пользователем или получаемые из других источников. Примерами могут быть SQL-инъекции, когда злоумышленник вводит SQL-код в форму, что может привести к выполнению нежелательных операций в базе данных.

## 3.2 Уязвимости аутентификации и авторизации (Authentication and Authorization Vulnerabilities)

Эти уязвимости возникают, когда система позволяет несанкционированным пользователям получить доступ к защищенным ресурсам или выполнить привилегированные операции без необходимых разрешений. Например, уязвимость слабой аутентификации может позволить злоумышленнику получить доступ к учетной записи другого пользователя.

## 3.3 Уязвимости управления сеансами (Session Management Vulnerabilities)

Эти уязвимости касаются неправильного управления сеансами пользователей, что может привести к возможности перехвата или подделки сеансов, а также к другим атакам, связанным с сеансами пользователей.

## 3.4 Уязвимости безопасности веб-приложений (Web Application Security Vulnerabilities)

Эти уязвимости специфичны для веб-приложений и включают в себя такие проблемы, как межсайтовый скриптинг (XSS), кросс-сайтовая подделка запроса (CSRF), утечка информации и другие.

## 3.5 Уязвимости внедрения кода (Code Injection Vulnerabilities)

Эти уязвимости возникают, когда злоумышленник может внедрить и выполнить свой собственный код в системе. Это может быть, например, внедрение кода в исполняемый файл или внедрение зловредного кода через веб-формы или URL.

## 3.6 Уязвимости внешних зависимостей (External Dependencies Vulnerabilities)

Эти уязвимости возникают из-за неправильного использования или управления внешними библиотеками, компонентами или сервисами, что может привести к возможности атаки через них.

## 3.7 Вывод

Это лишь небольшой обзор типов уязвимостей, которые могут существовать в программном обеспечении. Различные уязвимости могут сочетаться или проявляться в различных контекстах, поэтому важно обеспечить комплексный анализ безопасности при разработке и тестировании программного обеспечения.

# ГЛАВА 4

СИСТЕМЫ ИДЕНТИФИКАЦИИ И КАТАЛОГИЗАЦИИ УЯЗВИМОСТЕЙ В ПО

## 4.1 Common Vulnerabilities and Exposures (CVE)

Представляет собой стандартную систему идентификации и каталогизации уязвимостей в программном обеспечении. Он разработан с целью облегчения обмена информацией о безопасности между организациями, исследователями и поставщиками программного обеспечения.

Вот подробное описание ключевых аспектов системы CVE:

**1. Уникальные идентификаторы (CVE-ID):** Каждая уязвимость в системе CVE получает уникальный идентификатор в формате "CVE-ГОД-НОМЕР", где "ГОД" - год выявления уязвимости, а "НОМЕР" - порядковый номер уязвимости в этом году. Например, CVE-2022-12345. Этот идентификатор используется для однозначной идентификации уязвимости и ссылки на нее в различных базах данных, инструментах анализа безопасности и других ресурсах.

**2. Описание уязвимости:** Каждая запись CVE содержит описание уязвимости, которое включает в себя информацию о том, какая часть программного обеспечения или системы подвержена уязвимости, какую атаку она допускает, какие последствия могут возникнуть в случае успешной эксплуатации уязвимости, и другие подробности.

**3. Назначение аутентифицированных экспертов (CNA):** Для назначения и управления CVE-ID ответственные организации, называемые CNA (CVE Numbering Authorities), которые могут быть частными компаниями, организациями общественной безопасности или правительственными органами. Они отслеживают уязвимости, выдают им уникальные идентификаторы CVE и передают информацию об уязвимости в официальную базу данных CVE.

**4. Определение уровня угрозы:** Уязвимости, зарегистрированные в системе CVE, могут быть классифицированы по уровню угрозы, например, по стандарту CVSS (Common Vulnerability Scoring System), который оценивает серьезность уязвимости и определяет ее важность для безопасности.

**5. Международное использование:** Система CVE широко принята на международном уровне и используется как в коммерческих, так и в государственных организациях для идентификации и отслеживания уязвимостей в программном обеспечении.

## 4.2 Common Weakness Enumeration (CWE)

Это стандартная система классификации и каталогизации общих слабостей в программном обеспечении. CWE была разработана с целью предоставления единого и стандартизированного языка для описания и классификации различных типов слабостей, которые могут привести к уязвимостям в программном обеспечении.

Вот более подробное описание ключевых аспектов системы CWE:

**1. Уникальные идентификаторы (CWE-ID):** Каждая слабость в системе CWE получает уникальный идентификатор в формате "CWE-НОМЕР", где "НОМЕР" - это порядковый номер слабости в системе. Например, CWE-79. Этот идентификатор используется для однозначной идентификации слабости и ссылки на нее в различных базах данных, инструментах анализа безопасности и других ресурсах.

**2. Структурированный список слабостей:** CWE представляет собой структурированный список различных типов слабостей, охватывающих широкий спектр проблем, начиная от ошибок программирования и дизайна и заканчивая проблемами в процессах разработки программного обеспечения. Каждая слабость имеет свое описание и классифицируется на основе ее характеристик и контекста использования.

**3. Описание слабости:** Для каждой слабости в системе CWE предоставляется подробное описание, которое включает в себя информацию о том, что представляет собой слабость, какие типы ошибок или уязвимостей она может привести, примеры использования и рекомендации по устранению.

**4. Отношения между слабостями:** CWE также описывает связи между различными слабостями, позволяя анализировать их в контексте друг друга. Например, одна слабость может быть предпосылкой для другой или использоваться вместе для эксплуатации уязвимости.

**5. Интеграция с другими системами:** Система CWE широко принята и используется во многих других системах безопасности, таких как системы классификации уязвимостей (например, CVE), системы оценки рисков и стандарты безопасности разработки программного обеспечения.

## 4.3 Разница между CVE и CWE

CVE (Common Vulnerabilities and Exposures) и CWE (Common Weakness Enumeration) - это две различные системы классификации и идентификации уязвимостей в программном обеспечении. Вот их основные различия:

**1.CVE (Common Vulnerabilities and Exposures):**

* + CVE представляет собой словарь общеизвестных уязвимостей и экспозиций.
  + Каждая запись в CVE-словаре представляет собой уникальный идентификатор (CVE-ID), описание уязвимости и ссылки на дополнительные ресурсы.
  + CVE-ID используется для обозначения конкретной уязвимости и используется широко для идентификации и ссылки на уязвимости в различных базах данных, инструментах анализа безопасности и других ресурсах.
  + CVE сосредотачивается на конкретных уязвимостях и предоставляет краткое описание каждой из них.

**2.CWE (Common Weakness Enumeration):**

* + CWE, напротив, сосредотачивается на общих слабостях в программном обеспечении, которые могут стать причиной уязвимостей.
  + Он представляет собой структурированный список типов слабостей и ошибок, которые могут существовать в программном обеспечении на всех этапах его жизненного цикла.
  + Каждая слабость в CWE имеет свой уникальный идентификатор (CWE-ID), описание и ссылки на дополнительные ресурсы.
  + CWE-ID используется для обозначения конкретной слабости, а также для ссылок на различные примеры уязвимостей, которые могут быть связаны с этой слабостью.

Таким образом, основное различие между CVE и CWE заключается в том, что CVE сосредотачивается на конкретных уязвимостях, тогда как CWE сосредотачивается на общих слабостях, которые могут привести к уязвимостям. Обе системы широко используются сообществом информационной безопасности для идентификации, классификации и анализа уязвимостей в программном обеспечении.

## 4.4 Вывод

Использование системы CWE помогает стандартизировать описание и классификацию слабостей в программном обеспечении, что облегчает обмен информацией о безопасности между сторонами, а также помогает разработчикам исключать или уменьшать риски, связанные с этими слабостями в процессе разработки и тестирования программного обеспечения.

Использование системы CVE облегчает обмен информацией о безопасности между сторонами, позволяет эффективно отслеживать и анализировать уязвимости в программном обеспечении, а также принимать меры по их устранению или смягчению рисков.

# ЗАКЛЮЧЕНИЕ

С каждым годом количество кибератак и инцидентов безопасности, связанных с эксплуатацией уязвимостей в программном обеспечении, продолжает расти. В связи с этим, разработчики программного обеспечения, а также предприятия и организации, использующие программные продукты, сталкиваются с необходимостью обеспечения высокого уровня безопасности приложений. Статический анализ выгодно отличается от других методов поиска уязвимостей тем, что может обнаруживать потенциальные проблемы без выполнения программы, что позволяет выявлять даже самые скрытые и трудноуловимые уязвимости.

В настоящее время индустрия информационной безопасности активно развивается, внося новые методики и инструменты статического анализа, а также улучшая существующие системы. Это подчеркивает актуальность темы и необходимость постоянного совершенствования методов и систем статического анализа для обеспечения надежной защиты программного обеспечения от потенциальных угроз. Нами была проделана работа на основе анализатора Semgrep. Он полностью оупен-сорсный, легковесный и может анализировать код на различных ЯП. Главное условие для работы с инструментом- залогиниться. Итак, мы имеем Semgrep, установленный на нашей машине (репозиторий, склонированный из гитхаб локально на машину) и загружаем в него скрипт- пример. Сканирование запускается и результат высвечивается в командной строке и сохраняется в файле с расширением .txt. В нашем случае найдено 3 уязвимости:

1. Mass assignment (массовое присвоение), которое может быть проэксплуатировано атакующим.
2. Possible formatted SQL query (возможный форматированный SQL запрос)
3. Avoiding SQL string concatenation (избегание конкатенации строк) (Конкатенация строк в Python — это операция соединения двух или более строк в одну)

К каждому пункту прилагаются советы по исправлению и ссылки для более глубокого изучения вопроса. Также указаны номера строк с уязвимостями.

Исходя из вышеперечисленных критериев, связанных с отбором ПО для САСТ можно сделать вывод, что наше решение отвечает большинству требований: Семгреп имеет широкое покрытие, он гибкий, имеет возможность автоматизации, корректно обрабатывает изменения и дополнения, довольно точный, хорошо интегрируется со средой IDE и, главное, абсолютно доступный по цене.

# СПИСОК ИСПОЛЬЗОВАННОЙ ЛИТЕРАТУРЫ

1. С.С. Гайсарян, А.В. Чернов, А.А. Белеванцев, О.Р. Маликов, Д.М. Мельник,
2. А.В. Меньшикова. О некоторых задачах анализа и трансформации программ. Труды ИСП РАН, №5, с.7-41, 2004.
3. О.Р. Маликов, А.А. Белеванцев. Автоматическое обнаружение уязвимостей в программах. Материалы конференции «Технологии Майкрософт в теории и практике программирования», Москва, 2004.
4. О.Р. Маликов. Автоматическое обнаружение уязвимостей в исходном коде программ. Известия ТРТУ, №4, с. 48-53, 2005.
5. В.С. Несов, О.Р. Маликов. Использование информации о линейных зависимостях для обнаружения уязвимостей в исходном коде программ. Труды ИСП РАН, №9, с. 51-57, 2006.
6. О.Р. Маликов, В.С. Несов. Автоматический поиск уязвимостей в больших программах. Известия ТРТУ, Тематический выпуск «Информационная безопасность», №7 (62), с. 114-120, 2006.
7. В.С. Несов. Использование побочных эффектов функций для ускорения автоматического поиска уязвимостей в программах. Известия ЮФУ. Технические науки. Тематический выпуск «Информационная безопасность». Таганрог: Изд-во ТТИ ЮФУ, 2007. № 1(76), с. 134-139.
8. В.С. Несов, С.С. Гайсарян. Автоматическое обнаружение дефектов в исходном коде программ. Методы и технические средства обеспечения безопасности информации: Материалы XVII Общероссийской научно-технической конференции. СПб.: Изд-во Политехн. ун-та, 2008, с.107.
9. В.С. Несов. Автоматическое обнаружение дефектов при помощи

межпроцедурного статического анализа исходного кода. Материалы XI

Международной конференции «РусКрипто’2009».

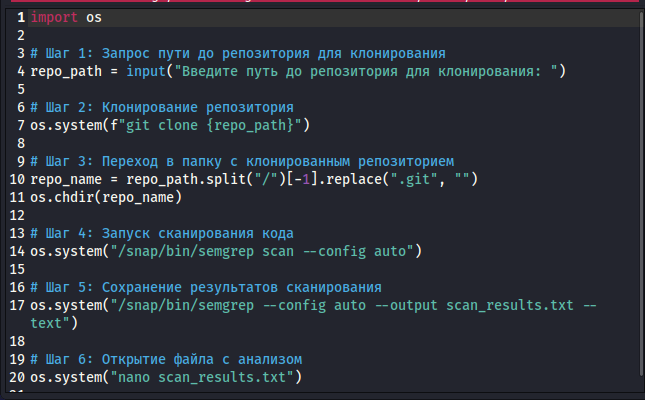
1. Vladimir Nesov. Automatically Finding Bugs in Open Source Programs. Electronic Communications of the EASST 20. ISSN 1863-2122, 2009.
2. Компилятор GCC. [ <http://gcc.gnu.org> ]
3. Компиляторная инфраструктура LLVM. [ <http://llvm.org> ]
4. Инструмент Coverity Prevent. [ <http://www.coverity.com/library/pdf/coverity_prevent.pdf> ]
5. Инструмент статического анализа компании Klocwork.

[ <http://www.klocwork.com/products/insight/klocwork-truepath> ]

1. Среда Eclipse [ <http://www.eclipse.org/> ]
2. P. Godefroid. The Soundness of Bugs is What Matters (position statement). In BUGS'2005 (PLDI'2005 Workshop on the Evaluation of Software Defect Detection Tools), 2005.
3. P. Emanuelsson & U. Nilsson. A Comparative Study of Industrial Static Analysis Tools (extended version). Tech. rep., Linköping University, 2008.
4. D. Liang & M. J. Harrold. Efficient Computation of Parameterized Pointer Information for Interprocedural Analyses. In SAS '01: Proceedings of the 8th International Symposium on Static Analysis, pp. 279-298, London, UK, Springer-Verlag, 2001.

# Приложение

1. Исходный код скрипта



1. Указываем репозиторий

Изображение выглядит как текст, снимок экрана, программное обеспечение, Веб-сайт

Автоматически созданное описаниеИзображение выглядит как текст, снимок экрана, программное обеспечение, Шрифт

Автоматически созданное описание

1. Вывод результата

Изображение выглядит как текст, электроника, снимок экрана, программное обеспечение

Автоматически созданное описаниеИзображение выглядит как текст, снимок экрана, программное обеспечение, мультимедиа

Автоматически созданное описание

Изображение выглядит как текст, снимок экрана, программное обеспечение, Значок на компьютере

Автоматически созданное описание

1. Исходный код сканируемых файлов

Изображение выглядит как текст, снимок экрана, программное обеспечение, Мультимедийное программное обеспечение

Автоматически созданное описание

Изображение выглядит как текст, снимок экрана, программное обеспечение, Мультимедийное программное обеспечение

Автоматически созданное описание