**Онлайн-школа Skillfactory**

ФИО, группа 1

**Выпускная квалификационная работа**

**по направлению ДПО «Информационная безопасность»**

по теме

**«Метод и система статического анализа для поиска уязвимостей в программном обеспечении»**

Студент

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ В.В. Костюков

подпись И.О. Фамилия

|  |  |
| --- | --- |
| Рецензент  ученая степень, звание  \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  И.О. Фамилия | Руководитель  ученая степень, звание(при наличии)  С. СЕРОВ  И.О. Фамилия  Консультант  ученая степень, звание(при наличии)  \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  И.О. Фамилия |
| Москва 2024 г. | |

**СОДЕРАЖАНИЕ**

[ВВЕДЕНИЕ 3](#_Toc161110388)

[ПОСТАНОВКА ЗАДАЧИ 5](#_Toc161110389)

[МЕТОДЫ СТАТИЧЕСКОГО АНАЛИЗА 6](#_Toc161110390)

[ПРЕИМУЩЕСТВА И ПОТЕНЦИАЛЬНЫЕ ОГРАНИЧЕНИЯ СТАТИЧЕСКОГО АНАЛИЗА КОДА 10](#_Toc161110391)

[МЕТОДОЛОГИЯ РАБОТЫ SAST 12](#_Toc161110392)

[SAST И DAST: РАЗЛИЧИЯ 14](#_Toc161110393)

[ПРАКТИЧЕСКАЯ ЧАСТЬ. АНАЛИЗАТОР BANDIT 17](#_Toc161110394)

[КОД РАЗРАБОТАННОГО СКРИПТА ДЛЯ SAST-АНАЛИЗА 18](#_Toc161110398)

[ПОДГОТОВКА К ТЕСТИРОВАНИЮ 19](#_Toc161110400)

[ТЕСТИРОВАНИЕ, ОБЗОР РЕЗУЛЬТАТОВ И РЕКОМЕНДАЦИИ ПО УСТРАНЕНИЮ НАЙДЕННЫХ УЯЗВИМОСТЕЙ 20](#_Toc161110401)

[ЗАКЛЮЧЕНИЕ 28](#_Toc161110406)

[СПИСОК ИСПОЛЬЗУЕМОЙ ЛИТЕРАТУРЫ И ИНТЕРНЕТ РЕСУРСОВ 31](#_Toc161110407)

[Приложение 1 *Отзыва руководителя на ВКР* 32](#_Toc161110408)

ВВЕДЕНИЕ

В современном мире, где программное обеспечение становится все более сложным и широко используемым, важность обеспечения его безопасности не может быть недооценена. Уязвимости в коде могут привести к серьезным последствиям, начиная от утечки конфиденциальных данных до дестабилизации системы в целом. Для выявления потенциальных проблем и предотвращения возможных атак разработаны методы и системы статического анализа кода, которые позволяют обнаруживать уязвимости на ранних этапах разработки.

Статический анализ кода - это процесс проверки исходного кода на наличие различных типов проблем, таких как стиль, форматирование, качество, производительность и проблемы безопасности. Согласно данным OWASP, статический анализ исходного кода является частью процесса безопасной разработки ПО, направленной на предотвращение распространенных уязвимостей. Для обнаружения таких уязвимостей используют метод статического тестирования безопасности приложений (SAST). Инструменты SAST разработаны специально для поиска проблем безопасности с высокой точностью, стремятся к низкому уровню ложноположительных и ложноотрицательных срабатываний и предоставляют подробную информацию об основных причинах и способах устранения обнаруженных уязвимостей. Статический анализ играет ключевую роль в обеспечении качества кода и повышении безопасности приложений. Использование автоматизированных систем SAST помогает разработчикам быстро находить и устранять проблемы, связанные с безопасностью, что позволяет сократить время разработки и повысить надежность программного обеспечения.

Исследования OWASP также указывают на то, что использование инструментов SAST должно сочетаться с другими методами тестирования безопасности, такими как динамический анализ или ручное тестирование кода (code review), для достижения комплексного подхода к обеспечению безопасности приложений. Таким образом, статический анализ исходного кода является эффективным методом поиска уязвимостей в программном обеспечении, который помогает повысить безопасность приложений и защитить данные пользователей от потенциальных киберугроз.

ПОСТАНОВКА ЗАДАЧИ

В данной работе будут рассмотрены методы и системы статического анализа, используемые для поиска уязвимостей в программном обеспечении, их преимущества и особенности применения.

В практической части будут разобраны примеры кода для некоторых популярных уязвимостей, которые можно обнаружить с помощью sast-анализа. Также в практической работе будет реализован автоматизированный скрипт, который использует инструмент статического анализа для поиска уязвимостей. На выходе будет формироваться документ с краткой информацией о проведенной проверке.

МЕТОДЫ СТАТИЧЕСКОГО АНАЛИЗА

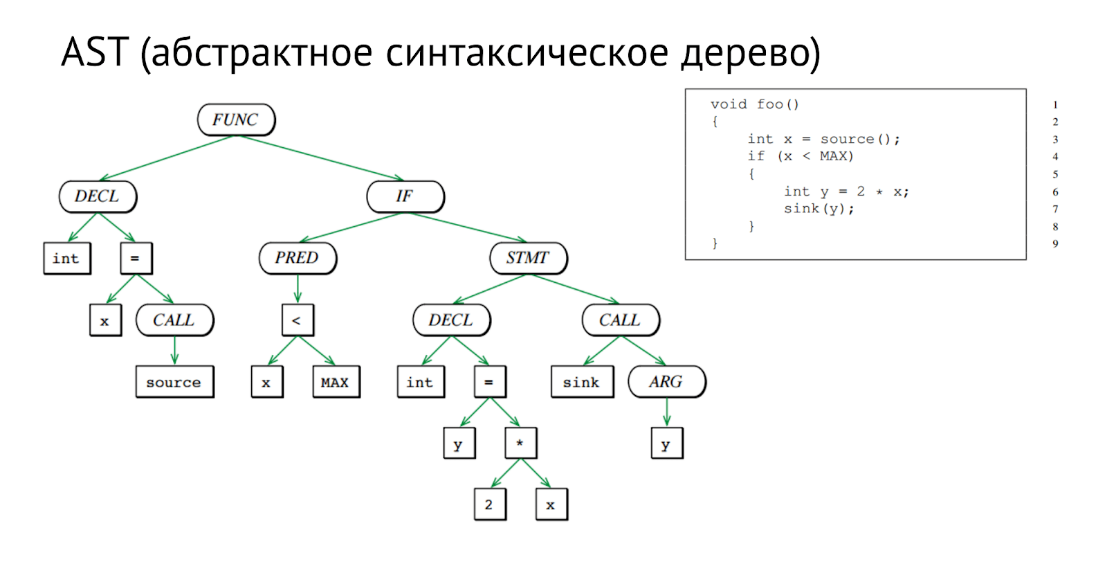
Прежде чем мы перейдём непосредственно к методам статического анализа, давайте разберемся, как работает статический анализ кода. Первым шагом в процессе статического анализа кода является ввод исходного кода. Разработчики должны предоставить доступ к своим файлам исходного кода или определённой кодовой базе, инструменту статического анализа, который они используют. Затем компилятор сканирует исходный код. В процессе сканирования, компилятор разбивает исходный код на более мелкие части, которые называют токенами. Если представить, что ваш исходный код представляет собой книгу, то токены – это слова, использованные при написании этой книги.

Следующий этап процесса – синтаксический анализ. Анализаторы берут токены и подтверждают, что они расположены в последовательности, которая имеет смысл в соответствии с используемым вами языком программирования, и организует их в структуру, называемую абстрактным синтаксическим деревом (AST), которое является абстрактным представлением структуры кода.

Теперь ваш инструмент статического анализа может начать проверку вашего кода. В зависимости от того, какой из них вы используете, инструмент выполняет методы анализа. Далее приведем основные методы анализа.

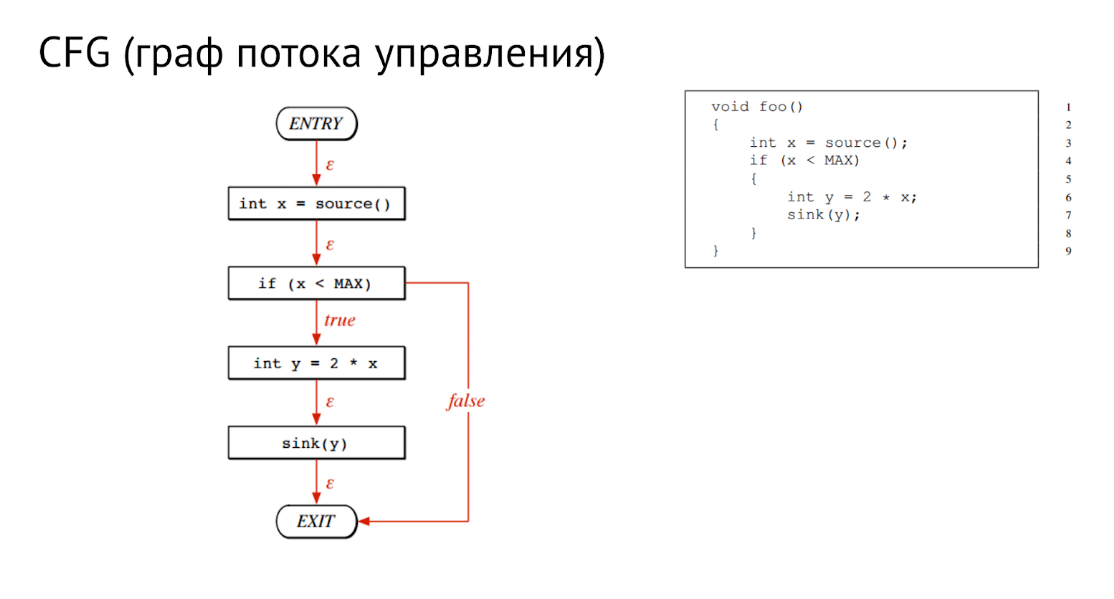
**Синтаксический анализ** - анализ, который проверяет наличие синтаксических ошибок и нарушений стиля кодирования, для обеспечения согласованности и удобочитаемости кода.

**Сигнатурный анализ, или поиск по шаблонам** - это то, что приходит в голову, если возникает задача найти какие-либо свойства исходного кода. Что может быть проще, чем написать регулярное выражение для поиска определенных моментов или дефектов? Такой подход основан на поиске заданных шаблонов, причем они могут быть заданы не только регулярным выражением, но и другими методами описания. Поиск также может производиться не только на строковом представлении исходного кода, но и, например, на потоке токенов или абстрактном синтаксическом дереве (слово «абстрактный» означает, что в таком дереве не представлены не значимые для анализа конструкции языка, например пробелы или комментарии).

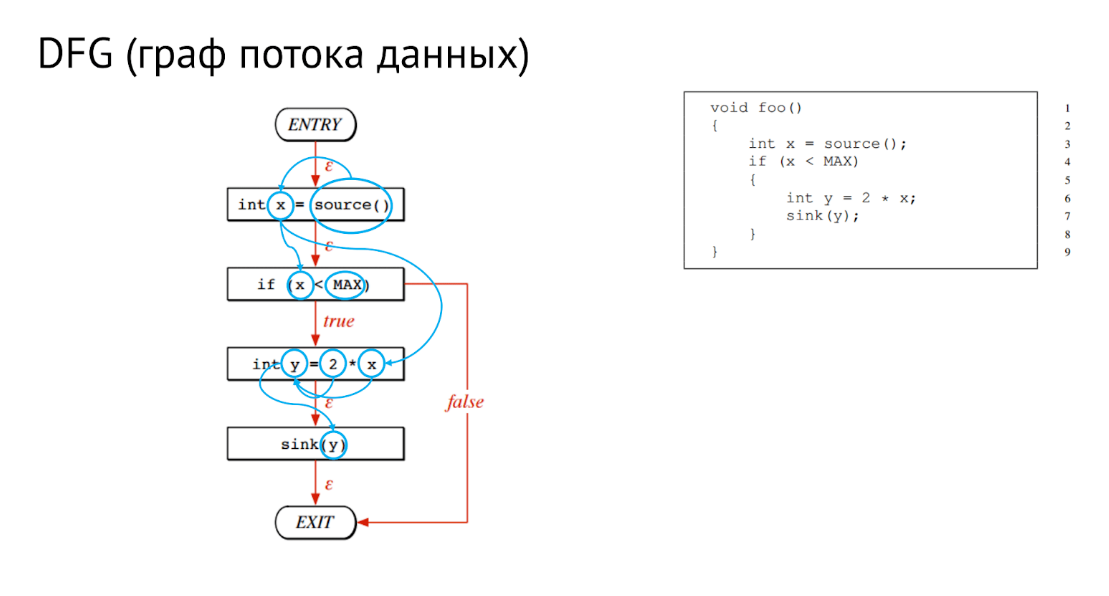


Недостатки такого подхода достаточно очевидны: он не учитывает семантику анализируемого приложения и уж тем более поток выполнения. Поэтому такой вид анализа достаточно неточен: в результате он имеет много ложных срабатываний. Однако стоит отметить, что поиск по шаблонам проходит относительно быстро, так как имеет линейную сложность.

**Анализ потока данных, или taint-анализ**, использует уже более сложные представления исходного кода. Например, граф потока данных, граф потока управления или различные их комбинации. Taint-анализ находит в коде зараженные данные (те, на которые может повлиять злоумышленник) и отслеживает все пути их распространения. Затем, если данные, которыекак-то были связаны с зараженными, попадают в место потенциальной уязвимости, то об этой уязвимости сообщается в результатах анализа.



Анализ потока управления исследует пути потока управления программой для выявления проблем, таких как мёртвый код, недоступный код и бесконечные циклы, для повышения надёжности и производительности кода.



Анализ потока данных отслеживает поток данных внутри кода для поиска проблем, связанных с использованием переменных, неинициализированных переменных и зависимостей данных, для обнаружения потенциальных ошибок во время выполнения.

Такой подход дает уже меньше ложных срабатываний, чем поиск по шаблонам, однако и он бывает неправ. Taint-анализ не учитывает условия достижимости путей выполнения и следует по всем ветвям, даже заведомо недостижимым. Кроме того, информация о том, может ли контролируемый извне поток данных достичь точки выполнения, которая потенциально приводит к уязвимости, не отвечает на вопросы о том, насколько сильно этот зараженный поток будет изменен, способен ли он привести к негативным последствиям и может ли такая ситуация вообще произойти. На все эти вопросы помогает ответить технология символьного исполнения, лежащая в основе большинства перспективных направлений статического анализа.

**Символьное исполнение (symbolic execution).** В этой технологии все входные данные становятся неизвестными переменными (символьными значениями), а затем происходит абстрактная интерпретация (аппроксимация семантики) без конкретного вычисления анализируемой программы. Это позволяет построить достаточно подробное представление заданной программы — граф потока вычисления. Из такого представления становится возможным получить в каждой точке выполнения программы условия, при которых эта точка будет достигнута, а также все значения переменных под теми условиями, под которыми они будут доступны в данном месте. Чаще всего, с помощью символьного исполнения, выявляются такие уязвимости, как SQL-иньекции, межсайтовый скриптинг XSS и переполнение буфера.

ПРЕИМУЩЕСТВА И ПОТЕНЦИАЛЬНЫЕ ОГРАНИЧЕНИЯ СТАТИЧЕСКОГО АНАЛИЗА КОДА

Статический анализ кода - один из столпов «движения за сдвиг тестирования влево», которое уделяет приоритетное внимание переносу тестирования программного обеспечения на как можно более ранние стадии разработки. При раннем и частом выполнении анализа исходного кода вы можете обнаружить и устранить проблемы до того, как они попадут в продукт, и их устранение станет более сложным и дорогостоящим.

Раннее тестирование исходного кода с помощью статического анализа помогает вашей команде достичь следующего:

* **согласованность** за счет применения стандартов кодирования, шаблонов и передовых практик;
* **повышенная безопасность** за счет раннего обнаружения недостатков в системе безопасности для снижения риска утечки данных;
* **повышение эффективности** за счет автоматизации тестирования вместо необходимости выполнять проверки кода вручную;
* **повышение ремонтопригодности** за счет быстрого устранения проблем и уменьшения подверженности ошибкам в вашей кодовой базе по мере ее роста.

Когда вы сможете создавать и отправлять код быстрее, сохраняя при этом согласованность, качество, безопасность и ремонтопригодность, вы сможете сократить затраты на разработку и даже увеличить доход, что является основным преимуществом эффективного процесса статического анализа кода. Однако статический анализ кода не является надежным решением, гарантирующим идеальный код.

Статический анализ не может постоянно выявлять все проблемы. Статические анализаторы обычно не обнаруживают проблем, связанных с поведением во время выполнения и внешними зависимостями. Они также могут не понимать логику, специфичную для домена. Статические анализаторы кода также склонны выдавать ложноположительные результаты.

Качество вашего анализа будет зависеть от тщательности правил, которые вы устанавливаете для каждого используемого вами инструмента.

МЕТОДОЛОГИЯ РАБОТЫ SAST

Методология работы SAST включает в себя несколько ключевых шагов, которые помогают эффективно выявить потенциальные проблемы без необходимости запуска приложения.

Первым этапом методологии является выбор подходящего инструмента статического анализа из множества доступных на рынке. OWASP представляет список рекомендуемых инструментов, таких как FindBugs, Checkmarx, Coverity, и другие, которые позволяют провести глубокий анализ исходного кода на предмет возможных уязвимостей.

Далее следует настройка выбранного инструмента в соответствии с требованиями проекта. Это включает определение набора правил анализа, настройку параметров сканирования и интеграцию SAST в процесс непрерывной интеграции и поставки (CI/CD). Критическое значение имеет также частота проведения сканирований - регулярные проверки кода с помощью SAST позволяют оперативно обнаруживать новые уязвимости.

Следующим шагом является выполнение самого сканирования и получение результатов. Инструменты SAST анализируют каждую строку кода на предмет потенциальных проблем, таких как SQL-инъекции, переполнения буфера или недостаточная обработка ошибок. Результаты сканирования содержат детальную информацию о найденных уязвимостях, их критичности и рекомендации по исправлению.

Наконец, последний этап методологии работы SAST - анализ результатов и исправление обнаруженных проблем. Разработчики должны проанализировать отчеты SAST, выявить и устранить уязвимости до того, как они окажутся в продакшене. После каждого цикла сканирования необходимо проводить повторные проверки для подтверждения исправлений.

Таким образом, методология работы SAST играет ключевую роль в обеспечении безопасности программного обеспечения путем раннего обнаружения и устранения потенциальных уязвимостей. Ее правильная реализация позволяет создавать более надежные приложения и защищать пользователей от возможных кибератак.

SAST И DAST: РАЗЛИЧИЯ

Статический и динамический анализ кода - это процессы, которые помогают вам обнаруживать дефекты в вашем коде. Разница заключается в том, на какой стадии цикла разработки выполняется анализ.

Статический анализ выявляет проблемы до запуска программы. Процесс происходит в среде, отличной от времени выполнения, между созданием кода и выполнением модульного тестирования.

Динамический анализ выявляет проблемы после запуска программы (во время модульного тестирования).  
В этом процессе вы тестируете код во время выполнения на реальном или виртуальном процессоре. Динамический анализ особенно эффективен для поиска малозаметных дефектов и уязвимостей, поскольку он рассматривает взаимодействие кода с другими базами данных, серверами и сервисами.

Однако динамический анализ сопровождается некоторыми важными предостережениями. Например, он обнаружит ошибки только в определенном фрагменте выполняемого кода, а не во всей кодовой базе.

Оба процесса являются важными. Статический анализ гарантирует меньшее количество дефектов во время модульного тестирования, а динамический анализ выявляет проблемы, которые могли быть пропущены вашими инструментами статического анализа. Для достижения максимально возможного уровня охвата тестированием следует объединить эти два метода.

**ВЫБОР СТАТИЧЕСКОГО АНАЛИЗАТОРА**

Статическое тестирование безопасности приложений значительно продвинулось вперед. Многие современные инструменты SCA интегрируются в DevOps и гибкие рабочие процессы и могут анализировать сложные, большие базы кода. Это означает лучший охват, меньше путаницы, меньше прерываний и более безопасные приложения. Но не все инструменты статического анализа кода предлагают надежные функции. Например, некоторые из них не зависят от среды или платформы; а некоторые поддерживают ограниченный набор фреймворков и языков. В этом разделе мы сосредоточимся на том, чтобы помочь вам выбрать инструменты статического анализа кода, которые помогут обезопасить ваше приложение, в первую очередь это инструменты SAST.

Выбор инструмента статического тестирования безопасности приложения зависит от ряда факторов, включая вашу среду разработки, бюджет безопасности, существующие инструменты, фреймворки, размер кодовой базы, языки и рабочий процесс разработки. Крайне важно выбрать правильный инструмент статического анализа кода, чтобы повысить производительность при минимизации разочарований разработчиков и дополнительных затрат. Вот несколько вопросов, которые следует учитывать при выборе инструмента с открытым исходным кодом или корпоративного инструмента:

* Является ли инструмент первым разработчиком?
* Поддерживает ли он отраслевые стандарты кодирования?
* Это платно или бесплатно?
* Насколько сложно его настроить и использовать?
* Насколько точны результаты?
* Насколько быстро выполняется сканирование?
* Предлагает ли он функции создания отчетов и оповещения?
* Поддерживает ли он ваши языки программирования?
* Насколько действенны и надежны результаты анализа?
* Предоставляет ли инструмент сводную информацию по коду и уязвимостям безопасности?
* Насколько хорошо он интегрируется с вашими существующими процессами разработки программного обеспечения?

Вот некоторые из лучших вариантов инструментов статического анализа кода с открытым исходным кодом. Инструменты в этом списке либо полностью с открытым исходным кодом, либо имеют бесплатный уровень.

1. **Snyk Code**: Доступный бесплатно, Snyk Code - это первый инструмент SAST для разработчиков, который охватывает множество языков, включая Python, Java, JavaScript и C ++. Snyk работает вместе с разработчиками в процессе написания кода и предоставляет им информацию, необходимую для предотвращения проникновения уязвимостей в кодовую базу.
2. **Bandit** - это инструмент с открытым исходным кодом, предназначенный для поиска распространенных проблем безопасности в коде Python.
3. **Brakeman - б**есплатный сканер уязвимостей , разработанный для приложений Ruby .
4. **Spotbugs - р**анее известный как "FindBugs" Spotbugs - это хорошо известный инструмент анализа кода с открытым исходным кодом для поиска ошибок и уязвимостей в коде Java.
5. **Graudit** - это скрипт и набор сигнатур, который позволяет находить недостатки безопасности в исходном коде с помощью утилиты GNU Grep.

Теперь давайте рассмотрим, как интегрировать инструменты SAST в конвейер DevSecOps.

С более полным списком SAST инструментов можно ознакомиться по этой ссылке : https://owasp.org/www-community/Source\_Code\_Analysis\_Tools

ПРАКТИЧЕСКАЯ ЧАСТЬ. АНАЛИЗАТОР BANDIT

В данной работе мы будем использовать Bandit - это инструмент, предназначенный для поиска распространенных проблем безопасности в коде Python. Для этого Bandit обрабатывает каждый файл, создает из него AST и запускает соответствующие плагины для узлов AST. Как только Bandit завершает сканирование всех файлов, он генерирует отчет.

Bandit поддерживает множество различных тестов для обнаружения различных проблем безопасности в коде python. Эти тесты создаются в виде плагинов, при необходимости можно создавать новые плагины для расширения функциональности, предлагаемой bandit сегодня. Так же присутствует возможность фильтрации плагинов, чтобы исключать поиск некоторых уязвимостей или наоборот включать поиск только конкретных.

Фильтрация в бандит доступна не только для плагинов, но и для уровня угрозы уязвимости. Отчёт можно сгенерировать в следующих форматах: csv, custom, html, json, sarif, screen, text, xml, yaml. В нашем скрипте мы будем использовать формат отчёта html и будем искать угрозы уровня medium или выше.

КОД РАЗРАБОТАННОГО СКРИПТА ДЛЯ SAST-АНАЛИЗА

Файл get\_from\_repo.py:

import subprocess

# Ввод адреса репозитория

repo\_url = input("Введите адрес репозитория GitHub: ")

# Клонирование репозитория

result = subprocess.run(["git", "clone", repo\_url])

if result.returncode == 0:

    print("Репозиторий успешно склонирован.")

    # Получение имени каталога

    repo\_name = repo\_url.split("/")[-1].replace(".git", "")

    # Запуск команды bandit

    bandit\_command = ["bandit", "-r", f"./{repo\_name}", "-ll", "-f", "html", "-o", f"report\_{repo\_name}.html"]

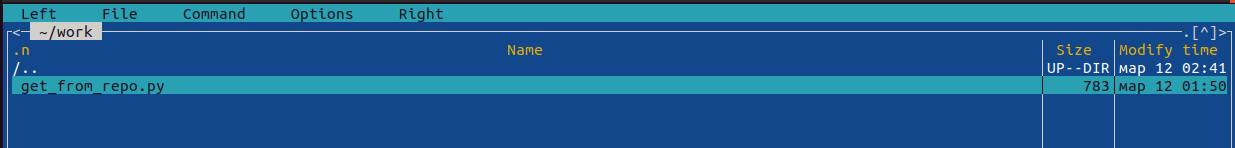
    subprocess.run(bandit\_command)

else:

    print("Ошибка при клонировании репозитория.")

ПОДГОТОВКА К ТЕСТИРОВАНИЮ

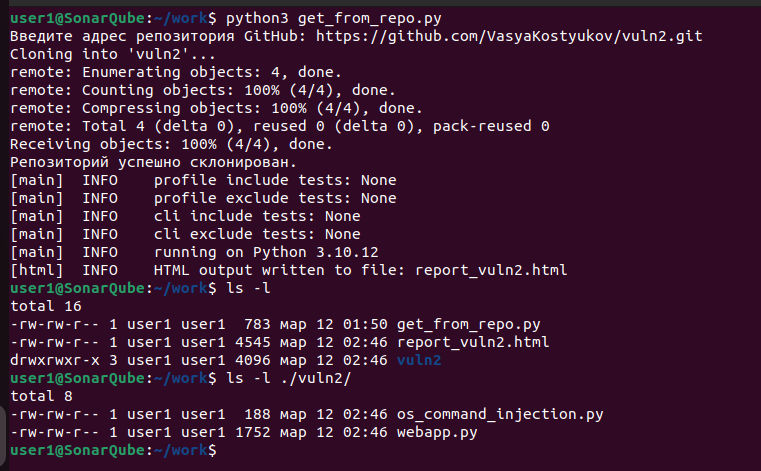
Подготовка к тестированию очень проста. Создаём каталог для работы, в который будем загружать из репозиториев код для анализа. Назовём это каталог work. Положим в него наш скрипт get\_from\_repo.py.



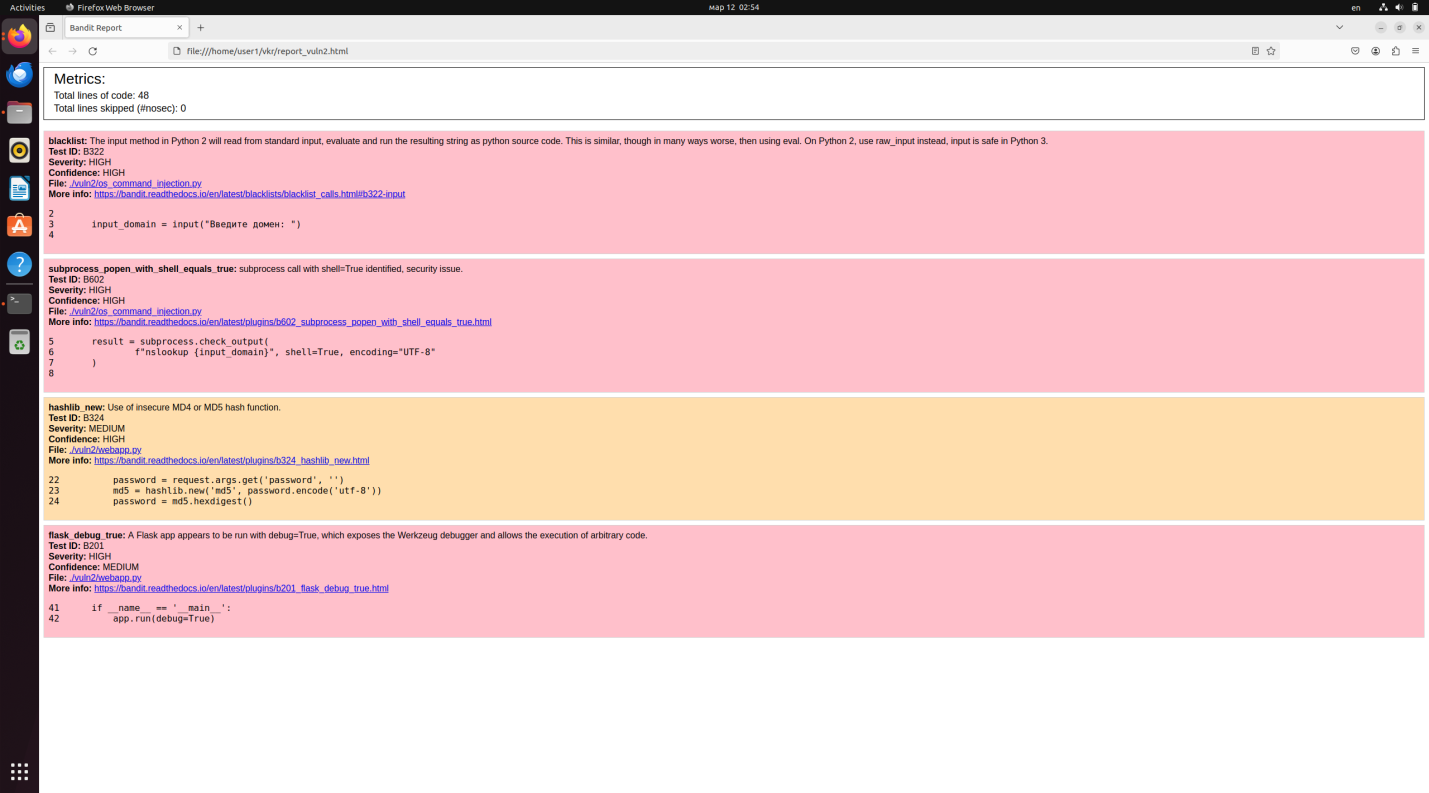
Также рекомендуется дать необходимые права доступа рабочему каталогу, с помощью команды chmod. После этого можно приступать к тестированию.

ТЕСТИРОВАНИЕ, ОБЗОР РЕЗУЛЬТАТОВ И РЕКОМЕНДАЦИИ ПО УСТРАНЕНИЮ НАЙДЕННЫХ УЯЗВИМОСТЕЙ

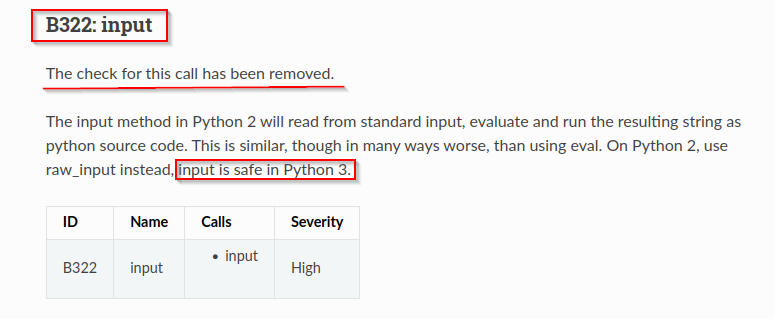
Запустим скрипт командой python3 get\_from\_repo.py. Скрипт запросит url адрес репозитория для тестирования. Введем адрес заранее подготовленного репозитория для проверки - <https://github.com/VasyaKostyukov/vuln2.git>. В результе, скрипт скачал наш репозиторий, запусти сканирование файлов и подготовил отчёт.



Теперь можно открыть отчёт и проанализировать найденные уязвимости



Итак, взглянем на первую ошибку с ID B322. Чтобы узнать про неё больше, переходим по ссылке из отчёта, откроется следующее описание:

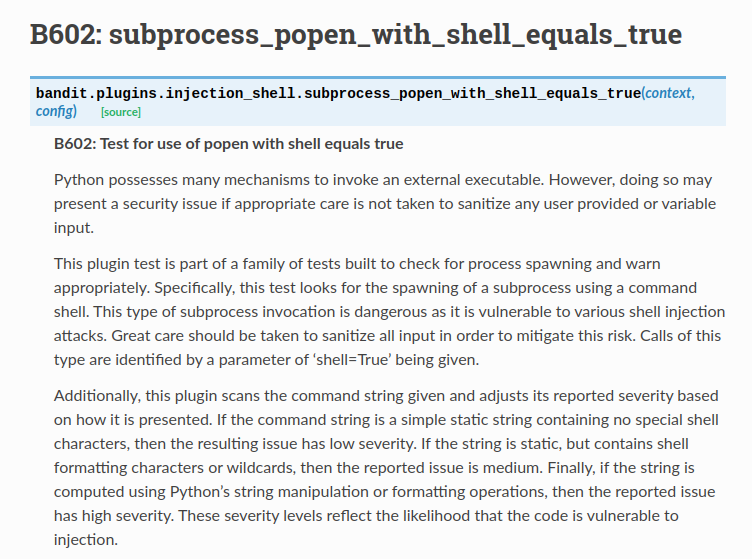


Видим, что проверка для этого вызова была удалена и указание, что для Python3 метод input безопасен. Следовательно, это ложноположительное срабатывание.

**Рекомендации по устранению уязвимости B602.**

Ничего исправлять не нужно.

Переходим к ошибке с ID B602. В описании указано следующее:

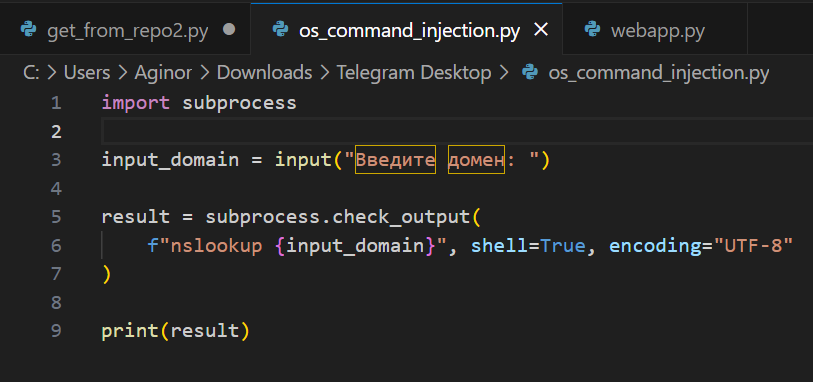


Мы обнаружили уязвимость, которая называется командная инъекция.

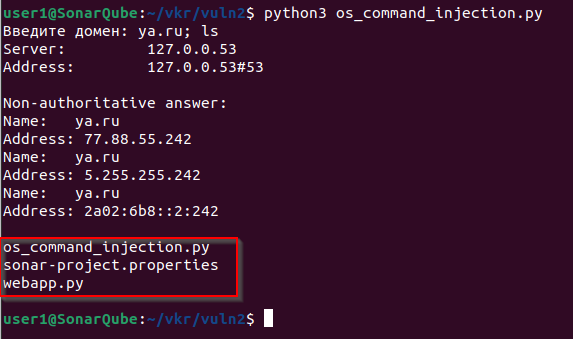
## По классификации MITRE - CWE-78: Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection'). Уровень угрозы высокий.

**Командная инъекция** — вид атаки, целью которой является выполнение произвольных команд в ОС сервера. Атака срабатывает, например, при запуске процесса с помощью функций модуля **subprocess**, когда в качестве аргументов используются значения, хранящиеся в переменных программы.

Посмотрим на код программы и попробуем разобраться, в чём тут дело.



В этом примере мы используем модуль subprocess, чтобы выполнить **nslookup** и получить информацию о домене. Что здесь может пойти не так? Конечный пользователь должен ввести домен, а скрипт должен вернуть результат выполнения команды nslookup. Но, если вместе с именем домена через точку с запятой ввести ещё и команду **ls**, будут запущены обе команды:



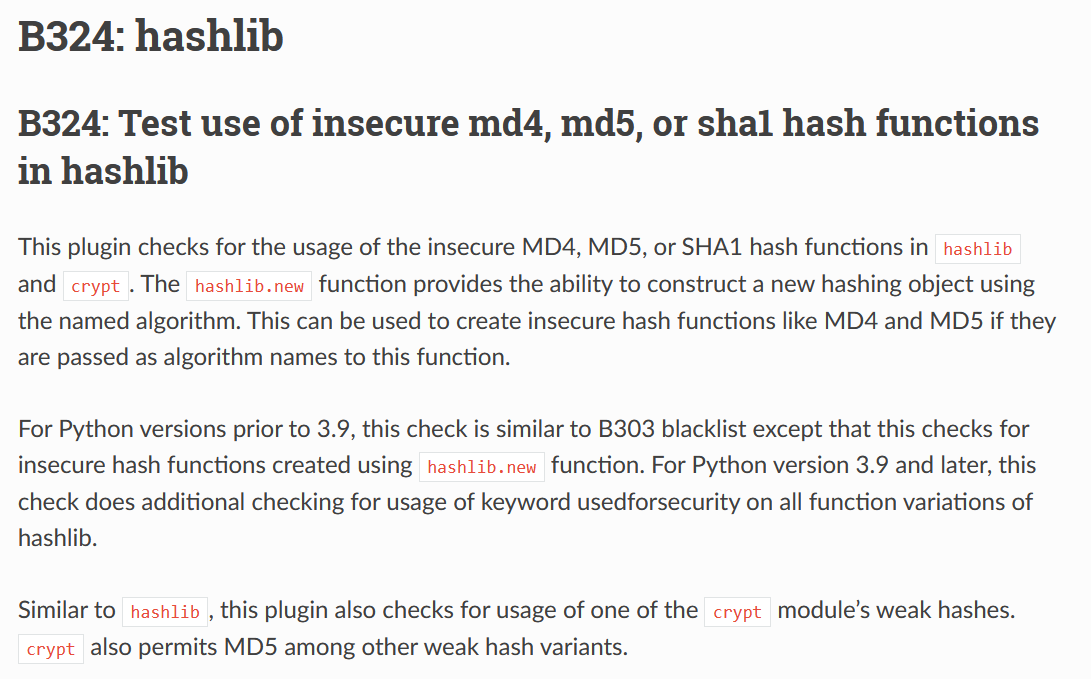
Используя эту уязвимость, можно выполнять команды на уровне ОС (у нас ведь **shell = true**). Представьте себе, что случится, если злоумышленник, например, отправит на выполнение команду **cat** для /etc/passwd, которая раскроет пароли существующих пользователей. Так что использование модуля subprocess может быть очень рискованным.

**Рекомендации по устранению уязвимости B602.**

Лучше всего будет отказаться от использования данного модуля с параметром shell = true. Если же это необходимо, то следует хорошо проанализировать введенную команду, попытаться запретить ввод спецсимволов вроде “;” чтобы избежать инъекции. А для bandit внести данный модуль в белый список в файле конфигурации и в список SKIPS внести идентификатор B602, чтобы пропускать этот тест при анализе:

**bandit-config-generator –s [B602] –o config.yml**

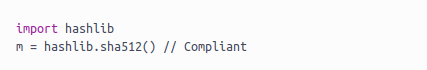
## Далее идёт уязвимость с ID B324 Test use of insecure md4, md5, or sha1 hash functions in hashlib. По классификации MITRE - CWE-327: Use of a Broken or Risky Cryptographic Algorithm. Уровень угрозы средний.



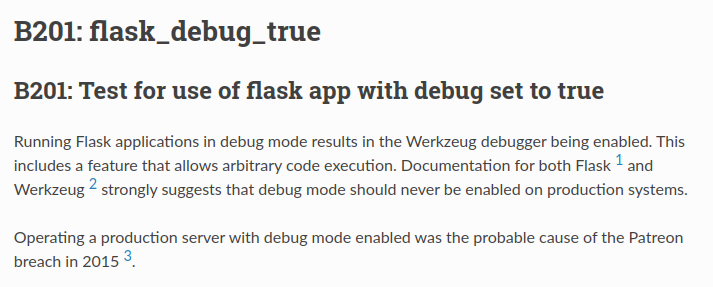
Эта уязвимость говорит о том, что в программе используется слабый и небезопасный метод шифрования хэш-функций. Данная уязвимость входила в [**OWASP Top 10 2021 Category A2**](https://owasp.org/Top10/A02_2021-Cryptographic_Failures/) **- Cryptographic Failures** и в настоящее время остаётся достаточно распространённой.

**Рекомендации по устранению уязвимости B324.**

Для устранения данной уязвимости рекомендуется использовать более безопасные алгоритмы хэширования функций, такие как SHA-256, SHA-384 и SHA-512. Также можно использовать криптографически безопасные библиотеки для работы с хэшами, такие как PyCryptodome или Cryptography для Python. Возможный пример решения в коде будет выглядеть так:



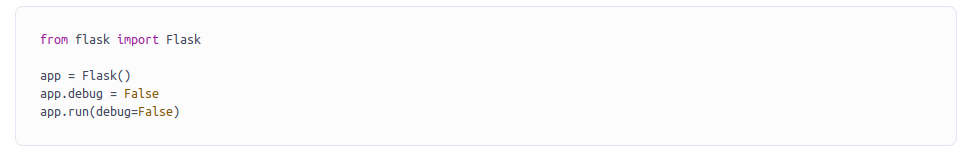
## Последняя в списке уязвимость B201: flask\_debug\_true. По классификации MITRE - CWE-94: Improper Control of Generation of Code ('Code Injection'). Уровень угрозы высокий.



Ошибка B201 указывает на использование устаревшего и небезопасного параметра flask\_debug\_true в конфигурации Flask. Средства разработки и фреймворки обычно имеют опции, облегчающие отладку для разработчиков. Хотя эти функции полезны во время разработки, их никогда не следует включать для приложений, развернутых в рабочей среде. Инструкции по отладке или сообщения об ошибках могут привести к утечке подробной информации о системе, такой как путь к приложению или имена файлов.

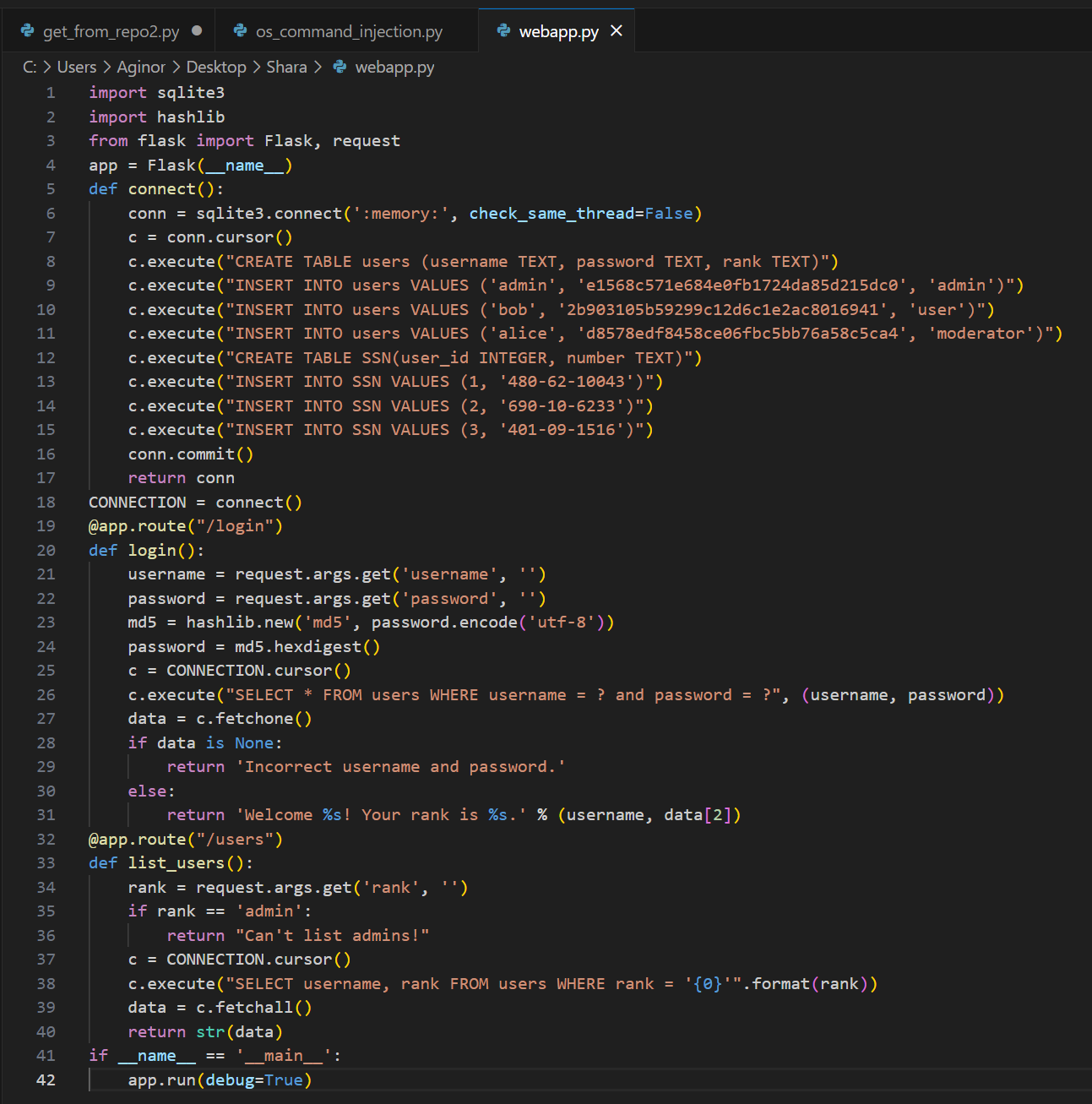
**Рекомендации по устранению уязвимости B201.**

В данном случае параметр flask\_debug\_true должен быть удалён из конфигурации или хотя бы выставлен с флагом false, чтобы избежать утечки конфиденциальной информации.



Так же рекомендуется использовать более безопасные методы отладки, такие как отладка через настройки среды или отладочные middleware, таких как Werkzeug Debug Middleware.

## На этом ошибки вроде бы закончились, но давайте посмотрим на код файла webapp.py и попробуем протестировать вручную эту программу:



Это веб-приложение имеет две конечные точки (URL): **login** и список пользователей (**users**). Для простоты обе конечные точки работают с параметрами **GET**. Вот несколько URL-адресов, которые мы будем попробовать при локальном запуске этого приложения:

http://localhost:5000/login?username=admin&password=l33t

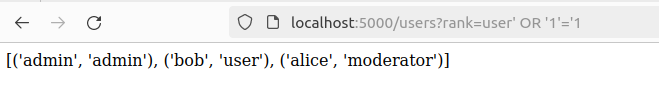
http://localhost:5000/login?username=admin&password=wrong\_pass

http://localhost:5000/users?rank=user

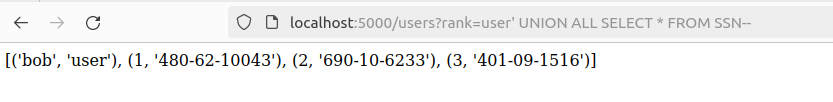
http://localhost:5000/users?rank=admin

Первая конечная точка регистрирует пользователя и защищена от атак внедрения SQL, поскольку она использует привязку параметров (подготовленные операторы). При подготовке запроса механизм SQLite кодирует и экранирует такие переменные, если это необходимо.

Вторая конечная точка выводит список всех пользователей и фильтрует их по переменной **rank**. Вместо привязки параметров используется форматирование строк и это имеет очень **серьезную уязвимость**. Поскольку переменная **rank** не экранирована или не обработана заранее, мы можем ввести произвольный код SQL в запрос. Добавив оператор **OR**, мы можем перечислить всех пользователей и обойти ограничение списка для категории администраторов:



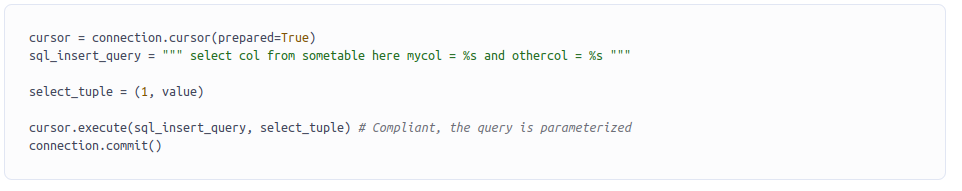
А это один из самых простых типов SQL инъекции, который позволяет вывести любую таблицу за один запрос:



Следовательно, в нашей программе присутствует уязвимость SQL-инъекция. По версии MITRE - **CWE-89: Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection'). Уровень угрозы Средний.** Хотя я лично с этим не согласен, так как по статистике это та самая уязвимость, которая наиболее часто используется злоумышленниками.

К сожалению, наш SAST инструмент, не смог найти данную уязвимость в файле webapp.py. Попытка просканировать приложение с помощью SonarQube тоже не дала результатов. Это говорит о том, что есть ложно отрицательное срабатывание, что весьма печально. В данном случае, мне только остаётся привести рекомендации по устранению SQL-инъекций и завершить свой анализ.

**Рекомендации по устранению :**



Для устранения SQL-инъекций в Python-коде можно использовать параметризованные запросы с помощью библиотеки SQLAlchemy или заменить использование конкатенации строк с параметрами на использование API для выполнения запросов с поддержкой параметров.

**ЗАКЛЮЧЕНИЕ**

В заключении приведём некоторые рекомендации по эффективному статическому анализу кода. Для достижения наилучших результатов статический анализ кода следует использовать в сочетании с другими методами тестирования, такими как:

* **динамическое тестирование** после интеграции кода для проверки поведения кода во время выполнения. Модульное тестирование, функциональное тестирование, тестирование интеграции, производительности и проникновения - все это динамические тесты, которые могут выявить потенциальные ошибки статического анализа кода;
* **ручной анализ кода,** при котором разработчики программного обеспечения читают свой исходный код и пытаются вручную обнаружить дефекты;
* **методы человеческого тестирования**, такие как удобство использования (usability) и тестирование приемлемости для пользователей, которые позволяют пользователям выявлять проблемы с производительностью и сообщать о них, которые ваша команда, возможно, не учла или с которыми столкнулась при создании определенных функций.

Чтобы получить максимальную отдачу от использования процессов и инструментов статического анализа, следует установить внутренние стандарты качества кода и задокументировать стандарты кодирования для вашего проекта.

Интегрируйте статический анализ в свой рабочий процесс разработки. Выберите инструменты, которые работают с предпочитаемой вами IDE и конвейером непрерывной интеграции и развертывания (CI / CD).

Не забывайте регулярно обновлять и поддерживать в рабочем состоянии инструменты статического анализа и наборы правил, чтобы повысить эффективность ваших инструментов и расширить спектр типов проблем, которые они могут выявлять.

СПИСОК ИСПОЛЬЗУЕМОЙ ЛИТЕРАТУРЫ И ИНТЕРНЕТ РЕСУРСОВ

Интернет ресурсы:

1. <https://owasp.org/www-community/controls/Static_Code_Analysis>
2. <https://blog.codacy.com/static-code-analysis>
3. <https://snyk.io/learn/open-source-static-code-analysis/>
4. <https://owasp.org/www-community/Source_Code_Analysis_Tools>
5. <https://bandit.readthedocs.io/en/latest/index.html>

Литература:

1. “Static Application Security Testing (SAST) - OWASP Foundation”
2. “SAST: Static Application Security Testing” - NIST
3. “Static Analysis Tools for Security” - CERT
4. “The Art of Software Security Assessment - Identifying and Preventing Software Vulnerabilities” - Patrick Ammann, Daniel Wohlgemuth
5. “A Survey of Static Analysis Methods for Security” - Stephen H. Edwards and Vishal Sharma
6. “Static Source Code Analysis and its Application to Security” - Michael D. Ernst, John H. Perkins, Philip J. Guo, Duvvathavel Vijayakumar, et al.
7. “Software Security Assurance: SAST” - ISECOM
8. “Static Code Analysis Tools: A Survey” - Xinwei Tian, Yulong Zhu, Zhiqiang Lin, Baowen Xu

### Приложение 1 *Отзыва руководителя на ВКР*

**ОТЗЫВ РУКОВОДИТЕЛЯ**

на выпускную квалификационную работу бакалавра

Студента \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Курса \_\_\_\_\_\_\_\_\_\_\_\_\_\_Группы\_\_\_\_\_

Руководитель\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

(Ф.И.О., ученая степень и (или) ученое звание)

Тема \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

*текст отзыва с указанием критериев оценивания*\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Выпускная квалификационная работа студента заслуживает оценки \_\_\_ баллов по 10‑балльной шкале.

Необходимо привести рекомендации о допуске выпускной работы к защите, отзыв о работе выпускника в период выполнения ВКР,оценку за работу по десятибалльной шкале и заключение о возможности присвоения студенту квалификации бакалавр по направлению подготовки «Информационная безопасность».

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Подпись \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

(Ф.И.О. руководителя)

Дата:\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_