Уязвимости python программ.

Автор: Вихлянцев Константин

2 ноября 2024 г.

1 Введение.

Первая четверть 21 века ознаменовалась эпохой глобальной и масштабной цифровизации, которая охватывает практически все аспекты жизни современного человека. Мы наблюдаем, как всё больше сфер: от общения и образования до бизнеса и развлечений, переходит в цифровое пространство. Это явление значительно повышает удобство и качество жизни, позволяя людям получать доступ к информации и услугам в любое время и в любом месте. Тем не менее, с этой трансформацией возникает и новая проблема: растущие объемы личной информации становятся уязвимыми для различных угроз и атак.

В условиях, когда данные о пользователях могут быть использованы в корыстных целях, задача защиты этой информации выходит на первый план. Кибератаки становятся всё более изощрёнными, и количество инцидентов, связанных с утечками данных, продолжает расти. В таких условиях безопасность данных и программного обеспечения становятся критически важными для обеспечения доверия со стороны пользователей и устойчивости бизнеса.

Одним из ключевых методов обеспечения сохранности данных является безопасная разработка программного обеспечения. Ведущие компании в области технологий осознают важность интеграции процессов безопасности на всех этапах жизненного цикла разработки. Это включает в себя не только тестирование программ на наличие уязвимостей, но и внедрение мер безопасности на стадии проектирования и разработки. В частности, многие из них внедряют статические анализаторы кода — инструменты, предназначенные для выявления потенциальных уязвимостей ещё до того, как программа будет запущена. Эти анализаторы проверяют исходный код на предмет ошибок, недочётов и уязвимостей без его фактического выполнения, что позволяет обнаружить проблемы на ранних стадиях.

Тем не менее, современные информационные продукты часто создаются с использованием нескольких языков программирования одновременно, что значительно усложняет работу статических анализаторов. Языки программирования можно условно разделить на две большие группы: строго типизированные и динамические. Первые появились на рынке значительно раньше вторых и, соответственно, имеют более широкое покрытие среди статических анализаторов, что обеспечивает более точный и надёжный анализ. В то же время динамически типизируемые языки, такие как Python, стали популярными относительно недавно, особенно в контексте современных технологий, таких как машинное обучение и искусственный интеллект.

Python, в частности, завоевал признание благодаря своей простоте, читаемости и универсальности, что сделало его одним из самых востребованных языков программирования в мире. Он активно используется в различных областях, включая веб-разработку, анализ данных и автоматизацию. Однако его динамическая природа создаёт дополнительные сложности для статического анализа кода. Это обусловлено тем, что динамически

типизированные языки позволяют изменять типы переменных во время выполнения, что затрудняет анализ корректности и безопасности программ.

В данной работе я намерен рассмотреть три наиболее распространённые уязвимости программ на Python: SQL-инъекции, внедрение кода и чрезмерные права доступа. Мы также проанализируем, способны ли популярные статические анализаторы, такие как CodeQL, SonarQube и Svace, успешно их выявлять. Это исследование поможет лучше понять эффективность этих инструментов в контексте безопасности программного обеспечения и может послужить основой для дальнейших улучшений в области защиты информации. Мы также предложим практические рекомендации по безопасному программированию, которые помогут разработчикам избегать распространённых ошибок и повышать уровень защищенности своих приложений.

2 Основная часть.

2.1 Внедрение SQL-кода: Уязвимость и Защита [CWE-89].

Внедрение SQL-кода (другими словами, SQL-инъекции) представляет собой одну из самых распространённых и опасных угроз для веб-приложений и систем, работающих с базами данных. Эта уязвимость возникает, когда приложение не в состоянии корректно обработать пользовательский ввод, что позволяет злоумышленникам внедрять произвольные SQL-запросы в код приложения. Чаще всего такие атаки становятся возможными из-за недостаточной валидации и фильтрации данных, поступающих от пользователей.

Рассмотрим следующий пример кода на Python, который демонстрирует эту уязвимость:

```
@app.route("/login")
def login():
  username = request.values.get("username")
  password = request.values.get("password")

  db = pymysql.connect("localhost")
  cursor = db.cursor()

  cursor.execute("SELECT␣*␣FROM␣users␣WHERE␣username␣=␣'%s'␣AND␣password␣=␣
      '%s'" % (username, password))

  record = cursor.fetchone()
  if record:
    session["logged_user"] = username

  db.close()
```

В этом коде пользовательский ввод для полей username и password напрямую подставляется в SQL-запрос. При вводе корректных данных, например, username="bob" и password="1234", будет сформирован следующий запрос:

SELECT * **FROM** users **WHERE** username = 'bob' **AND** password = '1234'

В этом случае база данных корректно обработает запрос и вернет запись о пользователе только в том случае, если он существует и введен правильный пароль. Однако если злоумышленник изменит пользовательский ввод на username="' OR 'a'='a';-"и введет любой набор символов для поля password, то сформированный запрос будет выглядеть так:

SELECT * **FROM** users **WHERE** username = '' **OR** 'a'='a';—' *AND password* = ' '

В результате база данных вернет все записи из таблицы users, поскольку условие OR 'a'='a' всегда истинно. Таким образом, злоумышленник может получить доступ к конфиденциальной информации, не зная правильных учетных данных.

SQL-инъекции активно использовались в 2008 году для взлома различных сервисов, что привело к утечкам данных и финансовым потерям. Эта угроза остается актуальной и по сей день, так как многие приложения продолжают использовать небезопасные практики обработки пользовательского ввода.

Для борьбы с этой уязвимостью разработчики библиотек для работы с базами данных, таких как sqlite3 для Python, внедрили специальные методы формирования безопасных запросов. Эти методы используют параметризованные запросы или подготовленные выражения, которые экранируют подставляемые параметры и предотвращают возможность выполнения вредоносного кода.

Кроме того, современные статические анализаторы кода, такие как CodeQL, SonarQube и Svace, активно проверяют исходный код Python-программ на предмет возможных SQL-инъекций. Эти инструменты помогают разработчикам выявлять потенциальные уязвимости на ранних этапах разработки, что существенно снижает риск появления опасных уязвимостей в новых проектах и приложениях с открытым исходным кодом. Например, SonarQube имеет более 20 шаблонов, на основе которых он ищет уязвимости типа SQL-инъекция. У CodeQL и Svace их меньше, однако они предлагают простой способ создания пользовательских шаблонов на основе плагинов и спецификаций соответственно.

Таким образом, внедрение SQL-кода представляет собой серьёзную угрозу для безопасности веб-приложений. Однако благодаря современным методам защиты и инструментам анализа кода разработчики могут значительно уменьшить вероятность успешной атаки и защитить свои системы от злоумышленников. Применение надлежащих практик программирования и регулярное использование статических анализаторов могут обеспечить высокий уровень безопасности данных и доверие пользователей к веб-приложениям.

2.2 Внедрение кода[CWE-94]

Уязвимость, связанная с внедрением кода, представляет собой серьезную проблему для систем, которые допускают использование пользовательского кода в качестве параметров для исполняемых команд без предварительной обработки и валидации. Эта уязвимость может привести к выполнению произвольных и потенциально опасных команд, что ставит под угрозу безопасность всей системы.

Рассмотрим следующий пример кода на языке Python:

```
1 import os
2
3
4 user_input = input("Enter␣a␣file␣name:␣")
5 os.system("cat␣" + user_input)
```

В данной программе используется функция os.system(), которая позволяет выполнять команды операционной системы. Однако она принимает необработанный ввод от пользователя, что создает уязвимость. Если злоумышленник введет специальный код, то он сможет выполнить произвольные команды в операционной системе.

Например, если пользователь введет строку `"--help; rm -rf /"`, то программа выполнит не только команду 'cat', но и команду 'rm -rf /', которая удалит все файлы и директории на корневом уровне файловой системы. Это может привести к полной потере данных и нарушению работы системы.

Еще один пример опасного ввода — это строка `"--help; export GLOBAL="bad path"; export PATH=$GLOBAL/bin:$PATH"`. В данном случае злоумышленник использует меха-

низм переменных окружения для изменения пути выполнения программ, что может позволить ему запускать вредоносные программы из нестандартных мест.

Таким образом, уязвимость внедрения кода подчеркивает важность тщательной обработки и валидации пользовательского ввода. Разработчики должны использовать безопасные методы выполнения команд, такие как параметры командной строки или специализированные библиотеки, которые минимизируют риск выполнения нежелательного кода. Кроме того, необходимо внедрять механизмы контроля доступа и мониторинга для защиты систем от подобных атак.

Статические анализаторы CodeQl, SonarQube и Svace обрабатывают уязвимости типа Code-injection так же хорошо, как и SQL-injection, так как эти ошибки относятся к одному семейству и очень востребованы у пользователей статических анализаторов.

2.3 Чрезмерные права доступа для критического ресурса[CWE-732].

Чрезмерные права доступа для критического ресурса представляют собой серьезную уязвимость, которая возникает, когда ресурс получает больше прав доступа, чем необходимо программе для нормального взаимодействия с ним. Это может привести к утечке конфиденциальной информации, несанкционированному использованию ресурса или даже к его повреждению. Особенно опасно, когда такой ресурс связан с конфигурацией программы, её выполнением или хранением конфиденциальных данных пользователей, таких как пароли, личная информация или финансовые данные.

Когда права доступа к ресурсам не ограничены должным образом, это может создать множество проблем. Например, злоумышленник может получить доступ к конфиденциальной информации или изменить параметры конфигурации приложения, что может вызвать сбои в его работе. Таким образом, важно обеспечить минимально необходимые права доступа для всех ресурсов, чтобы предотвратить потенциальные угрозы.

2.3.1 Пример 1.

Рассмотрим следующий код, который устанавливает маску режима создания пользовательских файлов (umask) процесса, равной нулю, создаёт конфигурационный файл и записывает в него строку данные, которые моделируются строкой "configuration data":

```
1 import os
2
3
4 os.umask(0)
5
6 with open('config.cfg', 'w') as f:
7         f.write('configuration␣data');
```

После выполнения этого кода на UNIX-системе результат использования команды 'ls -l' может выглядеть следующим образом:

```
-rw-rw-rw- 1 <name> <name> 13 Sep 22 11:39 config.cfg
```

Строка "rw-rw-rw-"указывает на то, что владелец файла, его группа и все пользователи могут читать и редактировать этот файл. Так как данный файл является конфигурационным, то запись в него некорректных параметров запуска может привести к непредсказуемому поведению программы. Например, злоумышленник может изменить параметры конфигурации, что приведет к уязвимостям в системе или даже к её поломке.

2.3.2 Пример 2.

Теперь рассмотрим стандартный процесс создания файла без изменения прав доступа:

```
with open('config.cfg', 'w') as f:
    f.write('configuration data');
```

В этом случае файл config.cfg будет иметь параметры доступа rw-rw-r−−, это означает, что сторонние пользователи не могут редактировать его. Это более безопасный подход.

Теперь добавим несколько дополнительных строк кода для изменения прав доступа:

```
with open('config.cfg', 'w') as f:
    f.write('configuration data');

from os import chmod
chmod('config.cfg', 0o666)
```

После выполнения данного кода на UNIX-системе результат команды 'ls -l' будет следующим:

−rw−rw−rw− 1 <name> <name> 13 Sep 22 11:39 config.cfg

Таким образом, мы снова сталкиваемся с той же проблемой, что и в предыдущем примере: файл становится доступным для редактирования всеми пользователями системы. Это создает риск того, что злоумышленники смогут вносить изменения в конфигурацию или содержимое файла, что может привести к серьезным последствиям для безопасности приложения и системы в целом.

В обоих примерах мы видим, как неправильное управление правами доступа может привести к серьезным уязвимостям. Для минимизации рисков разработчики должны тщательно контролировать права доступа к критическим ресурсам и применять принцип наименьших привилегий. Это значит, что ресурсы должны иметь только те права доступа, которые необходимы для их нормальной работы. Внедрение таких практик поможет значительно повысить уровень безопасности приложения и защитить его от потенциальных угроз.

Например, код ниже устанавливает такие права на файл, при которых только владелец файла может его читать и изменять.

```
import os


with open('config.cfg', 'w') as f:
    f.write('configuration data')

os.chmod('config.cfg', 0o600)
```

Данный тип уязвимости является не настолько распространённым в статических анализаторах. Из всех трёх, что мы взяли за основу, шаблон для нахождения неккоректного использования функции chmod модуля os есть только в CodeQL.

3 Заключение.

Программы на Python, благодаря своей популярности, стали основой для множества современных веб-приложений и сервисов. Однако их гибкость и динамическая природа также делают их уязвимыми для различных видов атак. SQL-инъекции, внедрение кода и чрезмерные права доступа — это лишь часть проблем, с которыми сталкиваются разработчики. Использование современных статических анализаторов, таких как CodeQL,

SonarQube и Svace, позволяет своевременно выявлять уязвимости и повышать уровень безопасности приложений.

Тем не менее, безопасность программного обеспечения — это не только использование инструментов для анализа кода. Разработчики должны внедрять культуру безопасного программирования, использовать лучшие практики и регулярно проводить проверки кода. Тщательная проверка пользовательского ввода, правильное управление правами доступа и использование безопасных методов работы с данными — это ключевые аспекты защиты программ.

С ростом количества атак на цифровые системы важно понимать, что безопасность — это не единовременное мероприятие, а непрерывный процесс, требующий постоянного внимания и обновления инструментов защиты.

Список литературы

[1] MITRE. *Common Weakness Enumeration: SQL Injection (CWE-89).* https://cwe.mitre.org/data/definitions/89.html

[2] Wikipedia. *Внедрение SQL-кода.* https://ru.wikipedia.org/wiki/%D0%92%D0%BD%D0%B5%D0%B4%D1%80%D0%B5%D0%BD%D0%B8%D0%B5_SQL-%D0%BA%D0%BE%D0%B4%D0%B0

[3] MITRE. *Common Weakness Enumeration: Code Injection (CWE-94).* https://cwe.mitre.org/data/definitions/94.html

[4] MITRE. *Common Weakness Enumeration: Incorrect Permission Assignment for Critical Resource (CWE-732).* https://cwe.mitre.org/data/definitions/732.html

[5] GitHub. *CodeQL for Python - Security Queries.* https://github.com/github/codeql/tree/v1.27.0/python/ql/src/Security

[6] SonarSource. *SonarQube Rule: SQL Injection (RSPEC-5122).* https://rules.sonarsource.com/python/tag/cwe/RSPEC-5122/

[7] Python Software Foundation. *Python 3.12 Documentation.* https://docs.python.org/3.12/