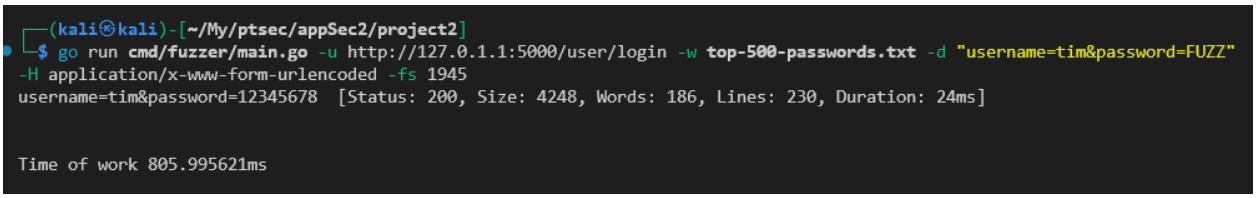
Исследуемое приложение https://github.com/fportantier/vulpy

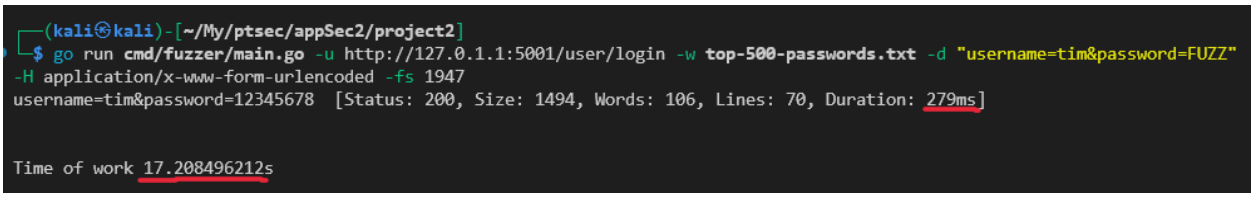
Динамический анализ

Запустим перебор паролей на уязвимом (bad версия) сервере (порт 5000)



Заметим, что перебор возможен

Теперь запустим тот же перебор, но уже на исправленном (good версия) сервере (порт 5001)



Заметим, что перебор так же возможен однако скорость полного перебора значительно выросла, из-за увеличения времени ответа сервера, на каждый запрос

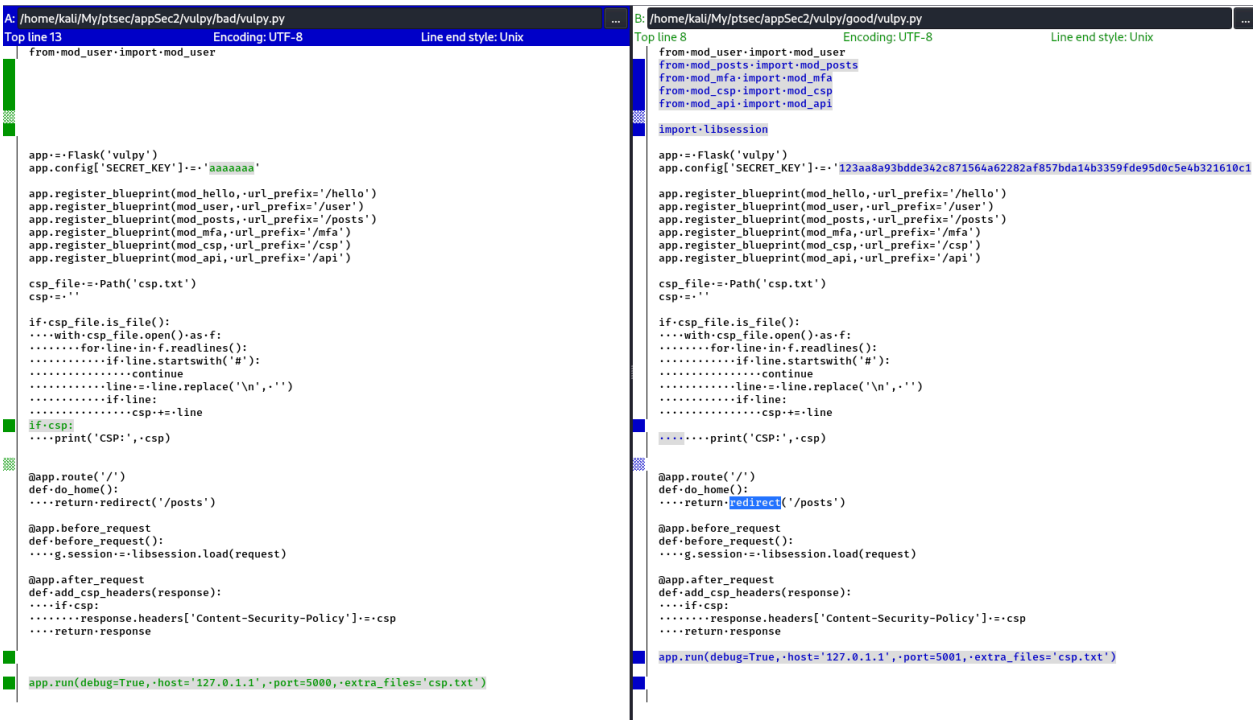
Время отклика может повлиять на возможность перебора паролей.

Анализ кода

Для сравнения кода между good и bad версий я использовал программу kdiff3

Найдем точку запуска python сервера. Он находится в файле vulnpy.py

Сравним между этот файл в разных версиях (слева bad-версия, справа – good)



Между собой файлы не сильно отличаются

Так как мы перебираем /user/login, найдем какие функции перехватывают запросы

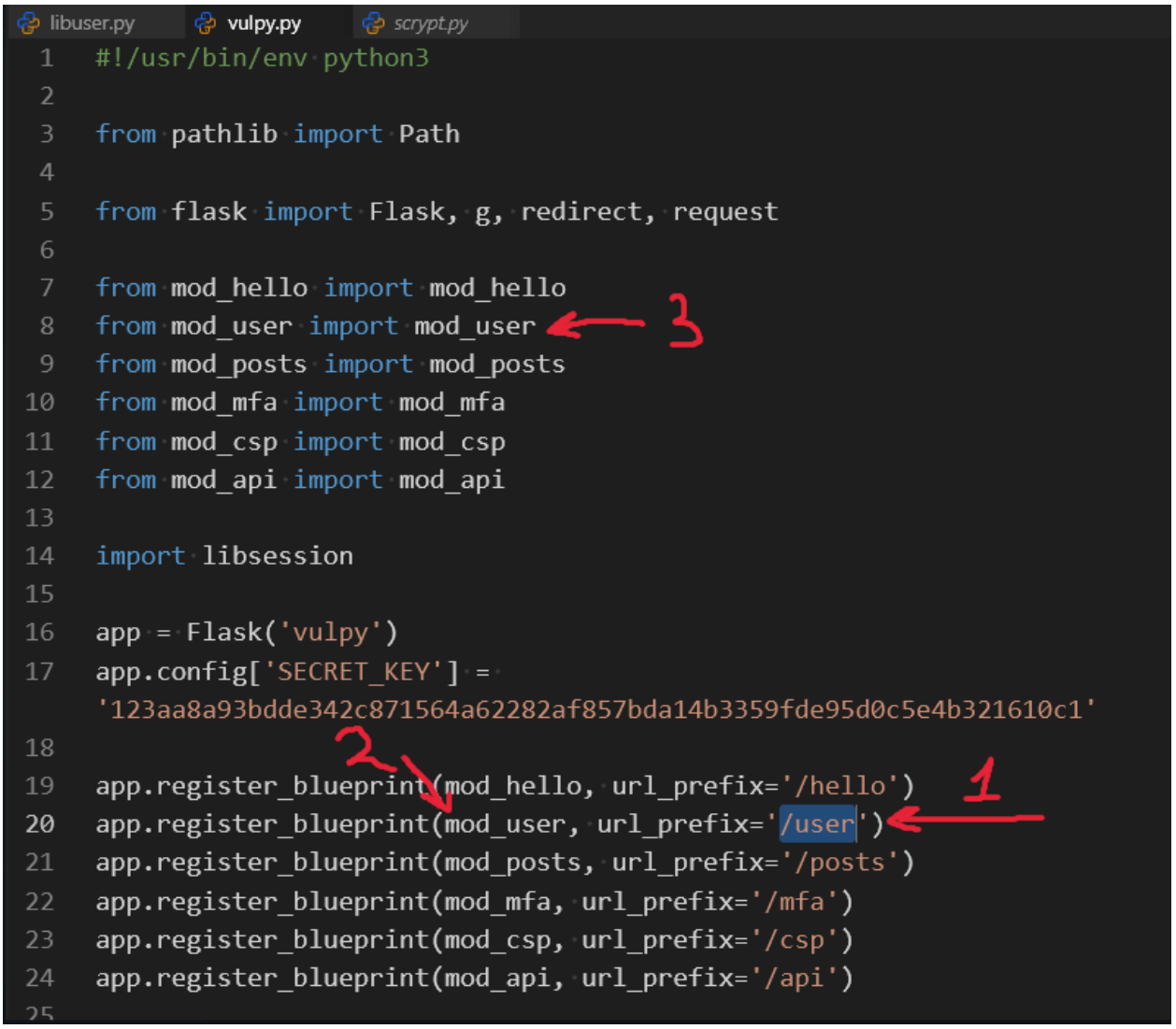


Рис.1 листинг файла vulpy.pu

Из рисунка 1 видно, что запросы на /user обрабатывает функция mod_user из файла mod_user

Найдем в этом файле функцию, перехватывающая запросы на /user/login и сравним файлы обеих версий

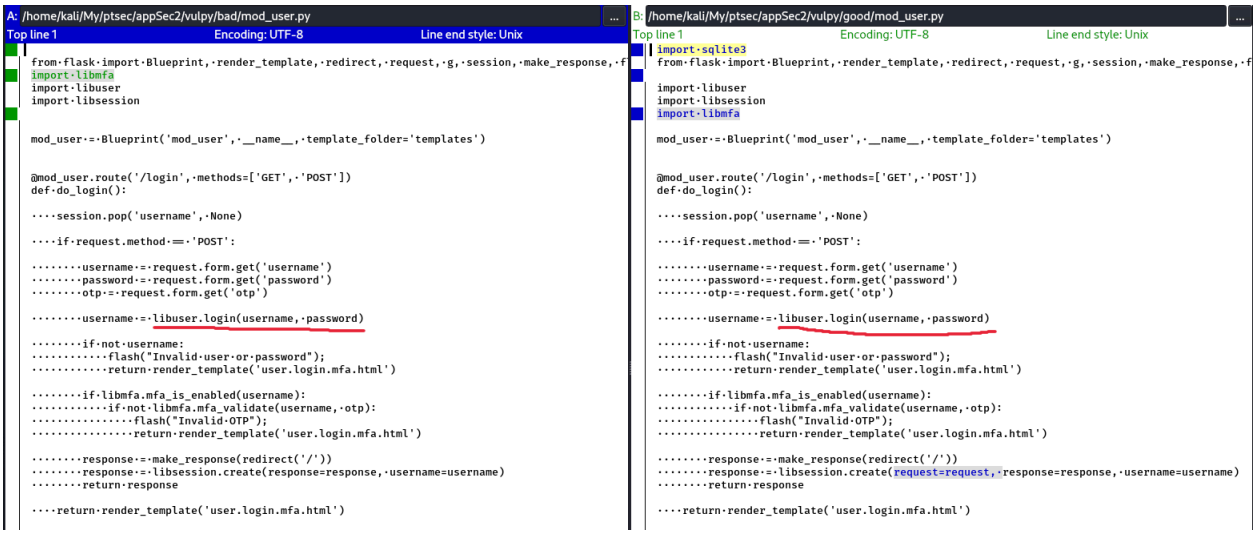


Рис.2 сравнение файла mod_user.py в разных версиях

В обоих версиях используется функция libuser.login, которая по всей видимости обращается к базе данных и возвращает данные пользователя, если он правильно ввел свой username и password

Перейдем к этой функции (libuser.login) и сравним обе версии

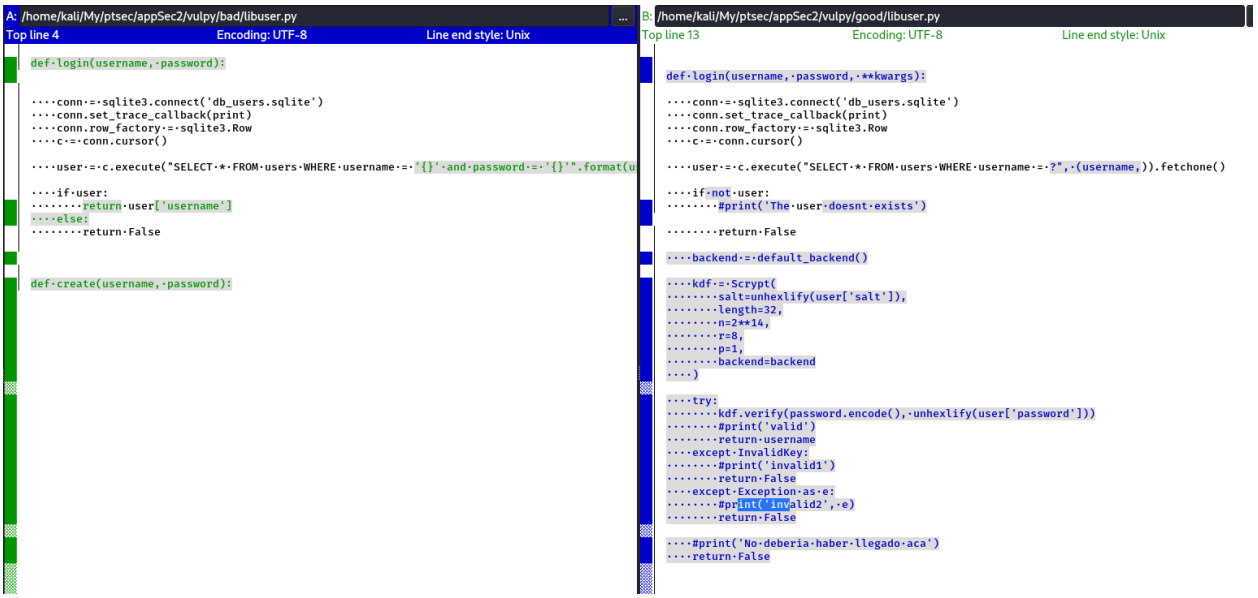


Рис.3 сравнение файла libuser.py в разных версиях

Заметим что теперь, эта функция значительно отличается в разных версиях.

В уязвимой версии функция возвращает пользователя, если он просто правильно ввел свои учётные данные.

(Также хочу отметить уязвимость SQLi в bad версии при выполнении запроса к базе данных при вызове функции c.execute)

В уязвимой версии пароль пользователя записывается в базу данных в открытом виде

PoC:

libuser.py good M | libuser.py bad M | libsession.py M | vulpy.py

```python
import sqlite3
import libuser


def login(username, password):

    conn = sqlite3.connect('db_users.sqlite')
    conn.set_trace_callback(print)
    conn.row_factory = sqlite3.Row
    c = conn.cursor()

    user = c.execute("SELECT * FROM users WHERE username = '{}' and password = '
    {}'".format(username, password)).fetchone()
    print(str(user['password']))

    if user:
        return user['username']
    else:
        return False
```

Рис.4 листинг программы libuser.py bad-версии (синим отмечена строка, необходимая для подтверждения суждения)

```
SELECT * FROM users WHERE username = 'admin' and password = 'SuperSecret'
SuperSecret
SELECT * FROM users WHERE username = 'admin' and mfa_enabled = 1
54802
<class 'bytes'>
127.0.0.1 - - [30/Jun/2024 15:32:49] "POST /user/login HTTP/1.1" 302 -
 * Detected change in '/home/kali/My/ptsec/appSec2/vulpy/bad/libsession.py', reloading
 * Restarting with stat
 * Debugger is active!
 * Debugger PIN: 899-221-760
SELECT * FROM users WHERE username = 'admin' and password = 'SuperSecret'
SuperSecret
SELECT * FROM users WHERE username = 'admin' and mfa_enabled = 1
127.0.0.1 - - [30/Jun/2024 15:33:38] "POST /user/login HTTP/1.1" 302 -
```

Рис.5 вывод программы vulpy.py (bad версии) при отправке запроса на /user/login (белым отмечен вывод, добавленный на рисунке 4)

Из рисунков 4 и 5 видно, что в бд пароли хранятся в открытом виде

Это небезопасное решение, так как есть риск дампа базы данных. В результате чего злоумышленники смогут узнать пароли пользователей

В базе данных следует хранить хэши паролей (с использованием солей)

Что и реализовано в good-версии

```
15  def login(username, password, **kwargs):
16
17      conn = sqlite3.connect('db_users.sqlite')
18      conn.set_trace_callback(print)
19      conn.row_factory = sqlite3.Row
20      c = conn.cursor()
21
22      user = c.execute("SELECT * FROM users WHERE username = ?", (username,)).
        fetchone()
23
24      if not user:
25          #print('The user doesnt exists')
26          return False
27
28      backend = default_backend()
29
30      kdf = Scrypt(
31          salt=unhexlify(user['salt']),
32          length=32,
33          n=2**14,
34          r=8,
35          p=1,
36          backend=backend
37      )
38
39      try:
40          kdf.verify(password.encode(), unhexlify(user['password']))
41          #print('valid')
42          return username
43      except InvalidKey:
44          #print('invalid1')
45          return False
46      except Exception as e:
47          #print('invalid2', e)
48          return False
49
```

Рис.6 листинг кода libuser.py (good версия)

На 30 строке мы можем заметить использование KDF с алгоритмом хэширования Scrypt, в параметрах N присваивается число 2^14 – кол-во итераций, которая проделает функция, чтобы получить хэш пароля

Именно от кол-ва итераций зависит скорость отклика сервера, поэтому выросло время ответа сервера

На строке 40 проводится проверка того, что хэш введенного пользователем пароля соответствует записи в базе данных.

Вывод: Из анализа кода мы поняли, что время ответа сервера увеличилось из-за использования функции хэширования паролей с большим количеством итерации.

Это решение усложняет bruteforce атаку. Но полностью не исключает возможность проведения атаки (видно из Динамического анализа), скорее предотвращает последствие взлома/дампа базы данных.

Против bruteforce атаки я бы порекомендовал использовать блокировку возможности входа (на время) после определенного количества неудачных попыток входа, также можно ввести способы дополнительного подтверждения личности (почта, телефон и т.п.) или же ввести captha, чтобы значительно увеличить время подбора.

(Хочу отметить, что ни одно решение полностью не избавляет от рисков: только снижает их)