Динамический анализ

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

```
(kali%kali)-[~/My/ptsec/appSec2/project2]

$ go run cmd/fuzzer/main.go -u http://127.0.1.1:5000/user/login -w top-500-passwords.txt -d "username=tim&password=FUZZ"
-H application/x-www-form-urlencoded -fs 1945
username=tim&password=12345678 [Status: 200, Size: 4248, Words: 186, Lines: 230, Duration: 24ms]

Time of work 805.995621ms
```

Заметим, что перебор возможен

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

```
(kali@kali)-[~/My/ptsec/appSec2/project2]

$ go run cmd/fuzzer/main.go -u http://127.0.1.1:5001/user/login -w top-500-passwords.txt -d "username=tim&password=FUZZ"
-H application/x-www-form-urlencoded -fs 1947
username=tim&password=12345678 [Status: 200, Size: 1494, Words: 106, Lines: 70, Duration: 279ms]

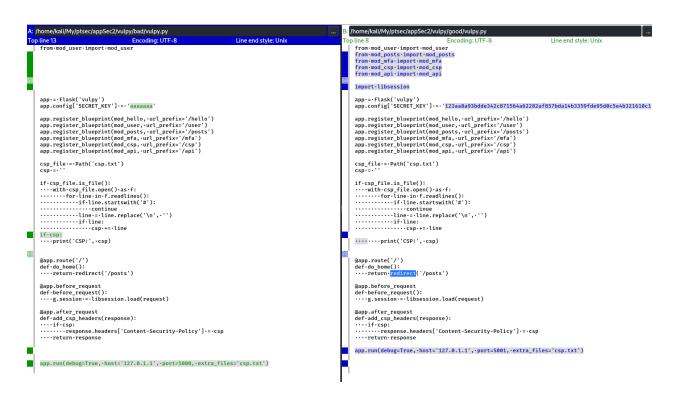
Time of work 17.208496212s
```

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

```
vulpy.py
合 libuser.py
    #!/usr/bin/env-python3
    from pathlib import Path
    from flask import Flask, g, redirect, request
     from mod hello import mod hello
    from mod user import mod user 🚄
    from mod_posts import mod_posts
    from mod mfa import mod mfa
    from mod csp import mod csp
11
12
    from mod api import mod api
    import libsession
    app = Flask('vulpy')
    app.config['SECRET KEY'] =
     '123aa8a93bdde342c871564a62282af857bda14b3359fde95d0c5e4b321610c1'
    app.register blueprint(mod hello, url prefix='/hello')
     app.register blueprint(mod user, url prefix='/user')
20
21
    app.register blueprint(mod posts, url prefix='/posts')
    app.register blueprint(mod mfa, url prefix='/mfa')
    app.register blueprint(mod csp, url prefix='/csp')
     app.register_blueprint(mod_api, url_prefix='/api')
```

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

```
/home/kali/My/ptsec/appSec2/vulpy/good/mod_user.py
                                                                                                                                                   i<mark>mport·sqlite3</mark>
from·flask·import·Blueprint,·render_template,·redirect,·request,·g,·session,·make_response
from·flask·import·Blueprint,·render_template,·redirect,·request,·g,·session,·make_response
mod_user·=·Blueprint('mod_user',·__name__,·template_folder='templates')
                                                                                                                                                  mod_user·=·Blueprint('mod_user',·__name__,·template_folder='templates')
                                                                                                                                                  @mod_user.route('/login',.methods=['GET',.'POST'])
def.do_login():
@mod_user.route('/login',.methods=['GET',.'POST'])
def.do_login():
····session.pop('username',·None)
                                                                                                                                                  ····session.pop('username', None)
····if·request.method·=·'POST':
                                                                                                                                                   ····if·request.method·=·'POST':
.....username·=·request.form.get('username')
.....password·=·request.form.get('password')
.....otp·=·request.form.get('otp')
                                                                                                                                                   .....username·=·request.form.get('username')
.....password·=·request.form.get('password')
.....otp·=·request.form.get('otp')
·····username·=·libuser.login(username,·password)
                                                                                                                                                  ·····username·=·libuser.login(username.·password)
.....if·not·username:
......flash("Invalid·user·or·password");
.....return·render_template('user.login.mfa.html')
                                                                                                                                                   .....if·not·username:
......flash('Invalid·user·or·password');
.....return·render_template('user.login.mfa.html')
....if·libmfa.mfa_is_enabled(username):
...if·not·libmfa.mfa_validate(username, otp):
...flash("Invalid-OTP");
...return·render_template('user.login.mfa.html')
                                                                                                                                                  ....if·libmfa.mfa_is_enabled(username):
...if·not·libmfa.mfa_validate(username,·otp):
...flash("Invalid-OTP"):
...return·render_template('user.login.mfa.html')
                                                                                                                                                  ·····response·-·make_response(redirect('/'))
·····response·-·libsession.create(request-request, response=response, ·username=username)
·····return response
·····response·=·make_response(redirect('/'))
·····response·=·libsession.create(response=response,·username=username)
·····return-response
····return·render_template('user.login.mfa.html')
                                                                                                                                                   ····return·render_template('user.login.mfa.html')
```

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

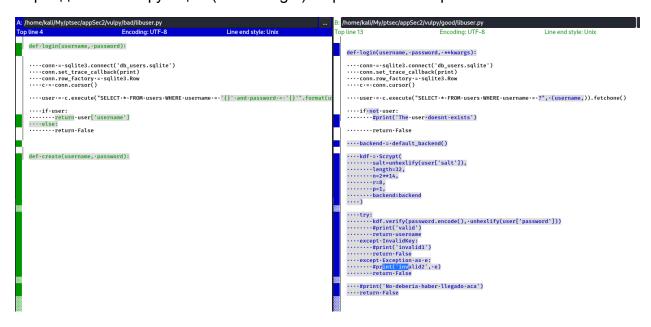


Рис.3 сравнение файла libuser.py в разных версиях

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 с.execute)

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

PoC:

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

```
def login(username, password, **kwargs):
        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 = ?", (username,)).
        fetchone()
23
       if not user:
       #print('The user doesnt exists')
      return False
        backend = default_backend()
        kdf = Scrypt(
            salt=unhexlify(user['salt']),
           length=32,
           n=2**14
           r=8,
           p=1,
           backend=backend
      try:
           kdf.verify(password.encode(), unhexlify(user['password']))
          #print('valid')
      return username
       except InvalidKey:
       except Exception as e:
      return False
```

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

<u>Именно от кол-ва итераций зависит скорость отклика сервера, поэтому выросло время ответа сервера</u>

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)