

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

## 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)

```
A: /home/kali/My/ptsec/appSec2/vulpy/bad/vulpy.py
Top line 13 Encoding: UTF-8 Line end style: Unix
from mod_user import mod_user

app = Flask('vulpy')
app.config['SECRET_KEY'] = 'aaaaaa'

app.register_blueprint(mod_hello, url_prefix='/hello')
app.register_blueprint(mod_user, url_prefix='/user')
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')

csp_file = Path('csp.txt')
csp = ''

if csp_file.is_file():
    with csp_file.open() as f:
        for line in f.readlines():
            if line.startswith('#'):
                continue
            line = line.replace('\n', '')
            if csp:
                csp += line
            else:
                csp = line

@app.route('/')
def do_home():
    return redirect('/posts')

@app.before_request
def before_request():
    g.session = libsession.load(request)

@app.after_request
def add_csp_headers(response):
    if csp:
        response.headers['Content-Security-Policy'] = csp
    return response

app.run(debug=True, host='127.0.1.1', port=5000, extra_files='csp.txt')
```

```
B: /home/kali/My/ptsec/appSec2/vulpy/good/vulpy.py
Top line 8 Encoding: UTF-8 Line end style: Unix
from mod_user import mod_user
from mod_posts import mod_posts
from mod_mfa import mod_mfa
from mod_csp import mod_csp
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')
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')

csp_file = Path('csp.txt')
csp = ''

if csp_file.is_file():
    with csp_file.open() as f:
        for line in f.readlines():
            if line.startswith('#'):
                continue
            line = line.replace('\n', '')
            if csp:
                csp += line
            else:
                csp = line

@app.route('/')
def do_home():
    return redirect('/posts')

@app.before_request
def before_request():
    g.session = libsession.load(request)

@app.after_request
def add_csp_headers(response):
    if csp:
        response.headers['Content-Security-Policy'] = csp
    return response

app.run(debug=True, host='127.0.1.1', port=5001, extra_files='csp.txt')
```

Между собой файлы не сильно отличаются

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

```
libuser.py  vulpy.py  script.py
1  #!/usr/bin/env python3
2
3  from pathlib import Path
4
5  from flask import Flask, g, redirect, request
6
7  from mod_hello import mod_hello
8  from mod_user import mod_user ← 3
9  from mod_posts import mod_posts
10 from mod_mfa import mod_mfa
11 from mod_csp import mod_csp
12 from mod_api import mod_api
13
14 import libsession
15
16 app = Flask('vulpy')
17 app.config['SECRET_KEY'] =
    '123aa8a93bdde342c871564a62282af857bda14b3359fde95d0c5e4b321610c1'
18
19 app.register_blueprint(mod_hello, url_prefix='/hello')
20 app.register_blueprint(mod_user, url_prefix='/user') ← 1
21 app.register_blueprint(mod_posts, url_prefix='/posts')
22 app.register_blueprint(mod_mfa, url_prefix='/mfa')
23 app.register_blueprint(mod_csp, url_prefix='/csp')
24 app.register_blueprint(mod_api, url_prefix='/api')
25
```

Рис.1 листинг файла vulpy.py

Из рисунка 1 видно, что запросы на /user обрабатывает функция mod\_user из файла mod\_user

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

```
A: /home/kali/My/ptsec/appSec2/vulpy/bad/mod_user.py
Top line 1 Encoding: UTF-8 Line end style: Unix
from flask import Blueprint, render_template, redirect, request, g, session, make_response, f
import libmfa
import libuser
import libsession

mod_user = Blueprint('mod_user', __name__, template_folder='templates')

@mod_user.route('/login', methods=['GET', 'POST'])
def do_login():
    session.pop('username', None)
    if request.method == 'POST':
        username = request.form.get('username')
        password = request.form.get('password')
        otp = request.form.get('otp')

        username = libuser.login(username, password)

        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')

        response = make_response(redirect('/'))
        response = libsession.create(response=response, username=username)
        return response
    return render_template('user.login.mfa.html')

B: /home/kali/My/ptsec/appSec2/vulpy/good/mod_user.py
Top line 1 Encoding: UTF-8 Line end style: Unix
import sqlite3
from flask import Blueprint, render_template, redirect, request, g, session, make_response, f
import libuser
import libsession
import libmfa

mod_user = Blueprint('mod_user', __name__, template_folder='templates')

@mod_user.route('/login', methods=['GET', 'POST'])
def do_login():
    session.pop('username', None)
    if request.method == 'POST':
        username = request.form.get('username')
        password = request.form.get('password')
        otp = request.form.get('otp')

        username = libuser.login(username, password)

        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')

        response = make_response(redirect('/'))
        response = libsession.create(request=request, response=response, username=username)
        return response
    return render_template('user.login.mfa.html')
```

Рис.2 сравнение файла mod\_user.py в разных версиях

В обеих 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

```
A: /home/kali/My/ptsec/appSec2/vulpy/bad/libuser.py
Top line 4 Encoding: UTF-8 Line end style: Unix
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 = '{0}' and password = '{1}'".format(u
    if user:
        return user['username']
    else:
        return False

def create(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 = '{0}' and password = '{1}'".format(u
    if user:
        return user['username']
    else:
        return False

B: /home/kali/My/ptsec/appSec2/vulpy/good/libuser.py
Top line 13 Encoding: UTF-8 Line end style: Unix
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 = '{0}' and password = '{1}'".format(u
    if not user:
        print('The user doesnt exists')
        return False

    backend = default_backend()

    kdf = Crypt(
        salt=unhexlify(user['salt']),
        length=32,
        ns=2**14,
        prf=PRF,
        p=1,
        backend=backend
    )

    try:
        kdf.verify(password.encode(), unhexlify(user['password']))
        print('valid')
        return username
    except InvalidKey:
        print('Invalid')
        return False
    except Exception as e:
        print('Invalid', e)
        return False

    print('No deberia haber llegado aca')
    return False
```

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 libuser.py bad libsession.py vulpy.py
1 import sqlite3
2 import libuser
3
4
5 def login(username, password):
6
7     conn = sqlite3.connect('db_users.sqlite')
8     conn.set_trace_callback(print)
9     conn.row_factory = sqlite3.Row
10    c = conn.cursor()
11
12    user = c.execute("SELECT * FROM users WHERE username = '{}' and password = '{}'".format(username, password)).fetchone()
13    print(str(user['password']))
14
15    if user:
16        return user['username']
17    else:
18        return False
19
20
```

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

```
* 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
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 captcha, чтобы значительно увеличить время подбора.

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