Explanation for Project 1

Ran Liao, SID 3034504227

February 5, 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.

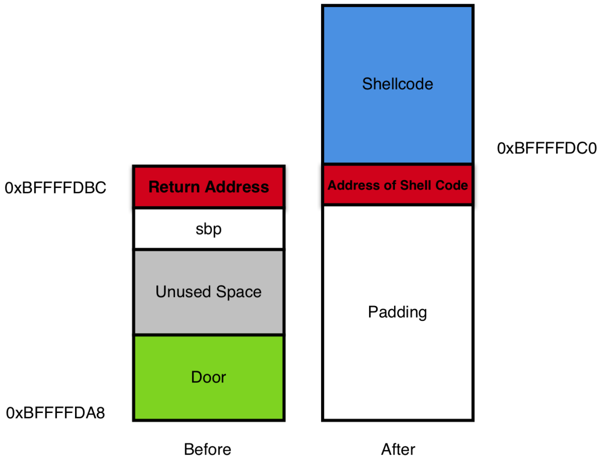
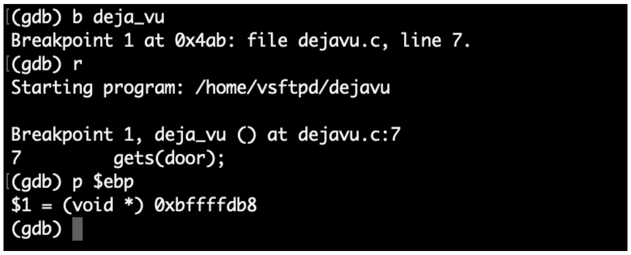


Figure 1: Stack Layout

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.

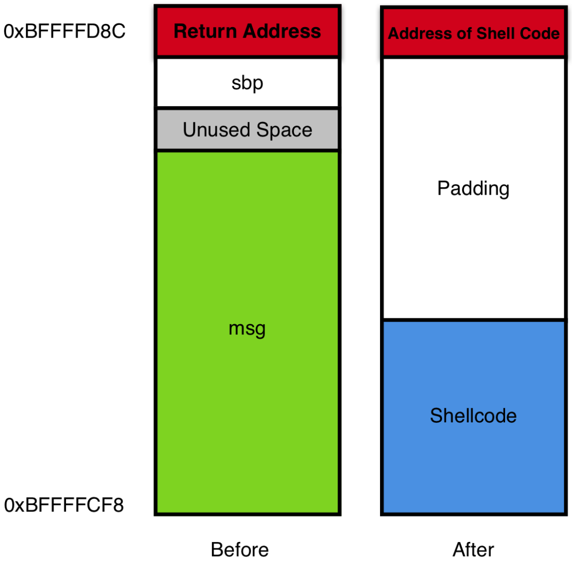
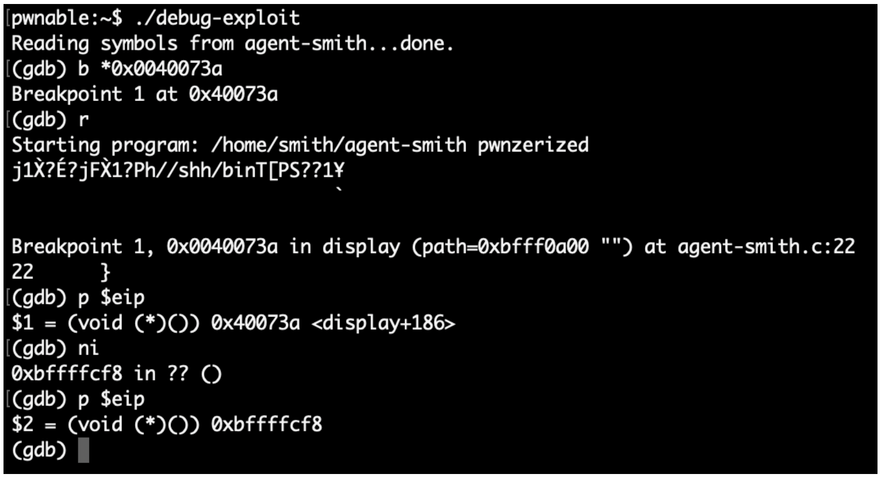


Figure 3: Stack Layout

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

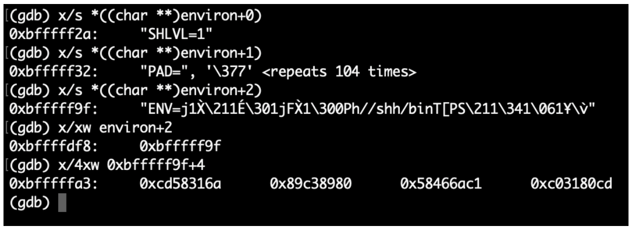
In figure 4, I use gdb to print the value in *eip* register. As you can see, it changed to 0xBFFFFCF8, which means I hijacked the control flow.

3 Deep Infiltration

In line 9, this buggy code miscalculated the length of array when checking the boundary. Therefore, it allows us to initiate an overflow attack and overwrite 1 byte.

There're two stage in this attack.

First, the shellcode is injected into environment variable. This is done by **egg** script. Figure 5 shows how to determine the address of injected shellcode in environment variable.



```
(gdb) x/s *((char **)environ+0)
0xbfffff2a:     "SHLVL=1"
(gdb) x/s *((char **)environ+1)
0xbfffff32:     "PAD=", '\377' <repeats 104 times>
(gdb) x/s *((char **)environ+2)
0xbfffff9f:     "ENV=j1X̀\211É\301jFX̀1\300Ph//shh/binT[PS\211\341\061¥\v̀"
(gdb) x/xw environ+2
0xbffffdf8:     0xbfffff9f
(gdb) x/4xw 0xbfffff9f+4
0xbfffffa3:     0xcd58316a      0x89c38980      0x58466ac1      0xc03180cd
(gdb) 
```

Figure 5: Retrive Shellcode Address

Second, I initiate an off-by-one overflow attack. Variable *buf* is adjacent to *sbp*. Therefore, the least significant byte in *sbp* will be overwrote. I let this byte be 0x10, which is the least significant part of *buf*'s starting address.

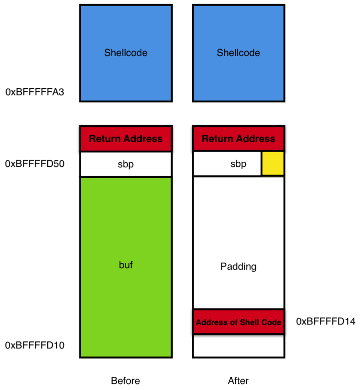
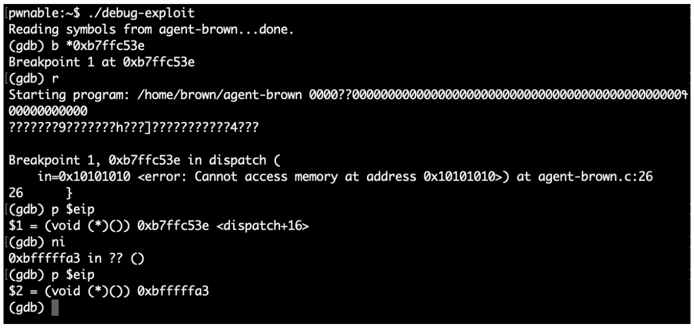


Figure 6: Stack Layout

After *flip*() finish and return control to function *invoke*(). The *ebp* register will have the wrong value pointing to variable *buf*. Then, the malicious address injected into *buf* will be pop into *eip* register when function *invoke*() finish. Figure 7 demonstrate how *eip* register changed after hijacking.



```
pwnable:~$ ./debug-exploit
Reading symbols from agent-brown...done.
(gdb) b *0xb7ffc53e
Breakpoint 1 at 0xb7ffc53e
(gdb) r
Starting program: /home/brown/agent-brown 0000??000000000000000000000000000000000000000000000ᵠ
00000000000
???????9???????h???]???????????4???

Breakpoint 1, 0xb7ffc53e in dispatch (
    in=0x10101010 <error: Cannot access memory at address 0x10101010>) at agent-brown.c:26
26      }
(gdb) p $eip
$1 = (void (*)()) 0xb7ffc53e <dispatch+16>
(gdb) ni
0xbfffffa3 in ?? ()
(gdb) p $eip
$2 = (void (*)()) 0xbfffffa3
(gdb) 
```

Figure 7: EIP value

4 Secret Exfiltration

This program will not behave properly if invalid input data is given to it. Specifically, if the last 4 bytes of input data is '\', 'x', '\n' and '\0' respectively, the bound checking of line 20 will fail. Therefore, more data behind this address can be printed.

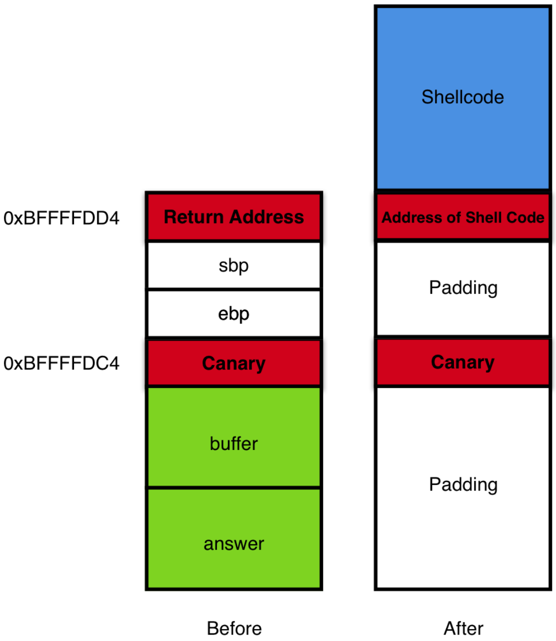


Figure 8: Stack Layout

As demonstrated in figure 8, the **canary** is right behind variable $buffer$. Therefore this overflow attack can be conducted in two steps. First, construct an invalid input string described early to retrieve the value of **canary**. Second, inject shellcode and overwrite **canary** correctly to avoid program from crashing. Again, the relative position of return address can be calculated after reading assembly codes carefully. And the value of *ebp* register can be printed by gdb.

The first attack string should like this, '\ x30 \ x30 \ x30 \ x'. The 4th to 8th value in response string will be the **canary**'s value.

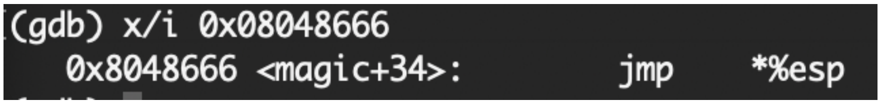
The second attack string will be like this,

$$'A' * 32 + \textbf{canary} +' A' * 8 +' \backslash xd4\backslash xfd\backslash xff\backslash xbf' + \textbf{SHELLCODE} +' \backslash n'$$

The length of A padding is 32. This is because all result will be copied to *answer* by this program, thus, 16 more padding is needed.

5 The Last Bastion

There's a magic number 58623 in the buggy program, which is $ffe4$ in hexadecimal. And it is $jmp * \%esp$ when interpreted as executable instruction. This instruction is not protected by ASLR, which means we can obtain its exact address. Figure 9 demonstrate how to find this instruction is gdb.



```
(gdb) x/i 0x08048666
   0x8048666 <magic+34>:        jmp    *%esp
```

Figure 9: Address of $jmp * \%esp$

This vulnerability makes it possible to initiate a **ret2esp** attack, which can bypass ASLR.

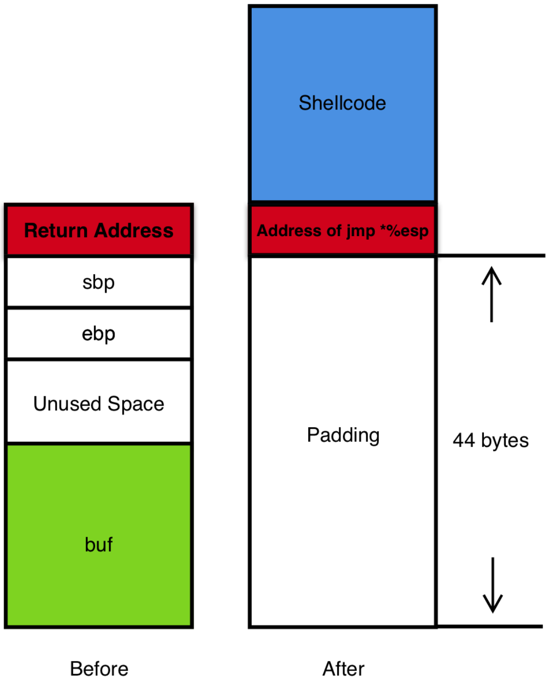


Figure 10: Stack Layout

Figure 10 demonstrate the memory layout before and after attack. Shellcode is placed right behind RIP, and RIP is redirected to $jmp * \%esp$ instruction.