7.6.1 缓冲区溢出的原理

- 由于函数里局部变量的内存分配是发生在栈帧里的，所以如果在某一个**函数内部**定义了缓冲区变量，则这个缓冲区变量所占用的内存空间是在该函数被调用时所建立的栈帧里。
- 由于**对缓冲区的潜在操作(比如字串的复制)都是从内存低址到高址的**，而**内存中所保存的函数返回地址往往就在该缓冲区的上方(高地址)**——这是由于栈的特性决定的，这就为**覆盖函数的返回地址**提供了条件。
- 当用大于目标缓冲区大小的内容来填充缓冲区时，就可以改写保存在函数栈帧中的**返回地址**，从而改变程序的执行流程，执行攻击者的代码。
- 以下例程(attack_overflow.c)给出Linux IA32构架缓冲区溢出的实例。

IA32构架缓冲区溢出的实例

attack_overflow.c

```
// Define a large buffer with 32 bytes.
char Lbuffer[] = "01234567890123456789========ABCD";  //32Byte
// Define a large buffer with ATTACK_BUFF_LEN bytes.
#define ATTACK_BUFF_LEN 1024
char attackStr[ATTACK_BUFF_LEN];
void foo()
{
    char buff[16];
    strcpy (buff, attackStr);
}
void justCopyTheLbuffer()
{
    strcpy(attackStr, Lbuffer);
}
int main(int argc, char * argv[])
{
    justCopyTheLbuffer();  foo();    return 0;
}
```

注：蓝色字表示用户输入的信息

- 编译并运行该C程序：

```
$ gcc -fno-stack-protector -o buf attack_overflow.c
$ ./buf
Segmentation fault (core dumped)
$ gdb buf
GNU gdb (Ubuntu 7.11.1-0ubuntu1~16.5) 7.11.1
Copyright (C) 2016 Free Software Foundation, Inc.
(gdb) r
Starting program: /home/i/work/buf

Program received signal SIGSEGV, Segmentation fault.
0x44434241 in ?? ()
(gdb)
```

- 可见会发生段错误。

- 为了找出错误原因，需要用gdb对程序./buf进行调试。

  $ gdb buf

  ……

- 反汇编main和foo：

```
(gdb) disas main
Dump of assembler code for function main:
   0x08048534 <+0>:     lea    0x4(%esp),%ecx
   0x08048538 <+4>:     and    $0xfffffff0,%esp
   0x0804853b <+7>:     pushl  -0x4(%ecx)
   0x0804853e <+10>:    push   %ebp
   0x0804853f <+11>:    mov    %esp,%ebp
   0x08048541 <+13>:    push   %ecx
   0x08048542 <+14>:    sub    $0x4,%esp
   0x08048545 <+17>:    call   0x80484ad <justCopyTheLbuffer>
   0x0804854a <+22>:    call   0x804846b <foo>
   0x0804854f <+27>:    mov    $0x0,%eax
   0x08048554 <+32>:    add    $0x4,%esp
   0x08048557 <+35>:    pop    %ecx
   0x08048558 <+36>:    pop    %ebp
   0x08048559 <+37>:    lea    -0x4(%ecx),%esp
   0x0804855c <+40>:    ret End of assembler dump.
(gdb)
```

```
(gdb) disas foo
Dump of assembler code for function foo:
   0x0804846b <+0>:     push   %ebp
   0x0804846c <+1>:     mov    %esp,%ebp
   0x0804846e <+3>:     sub    $0x18,%esp
   0x08048471 <+6>:     sub    $0x8,%esp
   0x08048474 <+9>:     push   $0x804a0a0
   0x08048479 <+14>:    lea    -0x18(%ebp),%eax
   0x0804847c <+17>:    push   %eax
   0x0804847d <+18>:    call   0x8048320 <strcpy@plt>
   0x08048482 <+23>:    add    $0x10,%esp
   0x08048485 <+26>:    nop
   0x08048486 <+27>:    leave
   0x08048487 <+28>:    ret
End of assembler dump.
```

设置断点

- 在函数foo的入口、对strcpy的调用、出口及其它需要重点分析的位置设置断点：

```
(gdb) b *(foo+0)
Breakpoint 1 at 0x804846b
(gdb) b *(foo+18)
Breakpoint 2 at 0x804847d
(gdb) b *(foo+28)
Breakpoint 3 at 0x8048487
(gdb) display/i $pc
```

运行程序并在断点处观察寄存器的值

```
(gdb) r
Starting program: /home/i/work/buf

Breakpoint 1, 0x0804846b in foo ()
1: x/i $pc
=> 0x804846b <foo>:      push   %ebp
(gdb) x/x $esp
0xbfffef2c:0x0804854f
(gdb) x/i 0x0804854f
   0x804854f <main+27>: mov    $0x0,%eax
```

- 函数入口处的堆栈指针esp指向的栈（地址为0xbfffef2c）保存了函数foo()返回到调用函数(main)的地址（0x0804854f），即“函数的返回地址”。

- 为了核实该结论，可以查看main的汇编代码：

```
(gdb) disas main
Dump of assembler code for function main:
   0x08048534 <+0>:    lea    0x4(%esp),%ecx
   0x08048538 <+4>:    and    $0xfffffff0,%esp
   0x0804853b <+7>:    pushl  -0x4(%ecx)
   0x0804853e <+10>:   push   %ebp
   0x0804853f <+11>:   mov    %esp,%ebp
   0x08048541 <+13>:   push   %ecx
   0x08048542 <+14>:   sub    $0x4,%esp
   0x08048545 <+17>:   call   0x80484ad <justCopyTheLbuffer>
   0x0804854a <+22>:   call   0x804846b <foo>
   0x0804854f <+27>:   mov    $0x0,%eax
   0x08048554 <+32>:   add    $0x4,%esp
   0x08048557 <+35>:   pop    %ecx
   0x08048558 <+36>:   pop    %ebp
   0x08048559 <+37>:   lea    -0x4(%ecx),%esp
   0x0804855c <+40>:   ret
End of assembler dump.
```

- 记录堆栈指针esp的值，在此以A标记：**A=$esp=0xbfffef2c**

继续执行到下一个断点

```
(gdb) c
Continuing.

Breakpoint 2, 0x0804847d in foo ()
1: x/i $pc
=> 0x804847d <foo+18>:call   0x8048320 <strcpy@plt>
(gdb)
```

- 查看执行strcpy(des, src)之前堆栈的内容。
- 由于C语言默认将参数逆序推入堆栈，因此，src（全局变量attackStr的地址）先进栈（高地址），des（foo()中buff的首地址）后进栈（低地址）。

```
(gdb) x/x $esp
0xbfffef00:     0xbfffef10
(gdb)
0xbfffef04:     0x0804a0a0
(gdb) x/x 0x0804a0a0
0x804a0a0 <attackStr>:    0x33323130
(gdb)
```

- 可见， attackStr 的地址0x0804a0a0保存在地址为0xbfffef04的栈中，
- buff的首地址0xbfffef10保存在地址为0xbfffef00的栈中。

- 令**B= buff的首地址=0xbfffef10**，则buff的首地址与返回地址所在栈的距离=A-B= 0xbfffef2c - 0xbfffef10 =**0x1c=28**。

  (gdb) p 0xbfffef2c - 0xbfffef10
  $1 = 28
  (gdb) p/x 0xbfffef2c - 0xbfffef10
  $2 = 0x1c
  (gdb)

- 因此，如果attackStr的内容超过28字节，则将发生缓冲区溢出，并且返回地址被改写。attackStr的长度为32字节，其中最后的4个字节为“ABCD”

  (gdb) x/x 0x0804a0a0 + 0x1c
  0x804a0bc <attackStr+28>: 0x44434241
  (gdb)

- 因此，执行strcpy(des, src)之后，返回地址由原来的0x0804854f变为”ABCD”（0x44434241），即返回地址被改写。
- 继续执行到下一个断点：

  (gdb) **c**
  Continuing.
  Breakpoint 3, 0x08048487 in foo ()
  1: x/i $pc
  => 0x8048487 <foo+28>: ret

- 即将执行的指令为ret。执行ret等价于以下三条指令：

  eip的值=esp指针指向的堆栈内容

  跳转到eip执行指令

  esp=esp+4

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

  0xbfffeffc: 0x44434241

- 可见，执行ret之前的堆栈的内容为“ABCD”，即0x44434241。
- 可以推断执行ret后将跳到地址0x44434241去执行。

- 继续单步执行下一条指令：

  (gdb) **si**

  0x44434241 in ?? ()

  1: x/i $pc

  => 0x44434241: <error: Cannot access memory at address 0x44434241>

  (gdb) **x/x $eip**

  0x44434241: Cannot access memory at address 0x44434241

  (gdb)

- 可见程序指针eip的值为0x44434241，而0x44434241是不可访问的地址，因此发生段错误。eip=0x44434241，正好是“ABCD”倒过来，这是由于IA32默认字节序为little_endian（低字节存放在低地址）。
- 通过修改attackStr的内容（将“ABCD”改成期望的地址），就可以设置需要的返回地址，从而可以将eip变为可以控制的地址，也就是说可以控制程序的执行流程。

调试重点

- 在漏洞函数的3个地方设置断点：
  - ✓(1) 第一条汇编语句：在此记下函数的返回地址（A=esp的值）(动态变化)
  - ✓(2) 调用strcpy对应的汇编语句：记下smallbuf的**起始地址=$esp=B** (动态变化)，与A相减可以得到产生缓冲区溢出所需的字节数 **Offset=A-B**
  - ✓(3) ret语句：查看esp的内容，确定被修改的返回地址。

7.6.2 缓冲区溢出攻击技术

- 为了实现缓冲区溢出攻击，需要向被攻击的缓冲区写入合适的内容。为此，攻击者必须精心构造攻击串，并根据被攻击缓冲区的大小将shellcode放置在的适当位置。在此以strcpy为例，说明攻击串的构造方法。考虑如下函数：

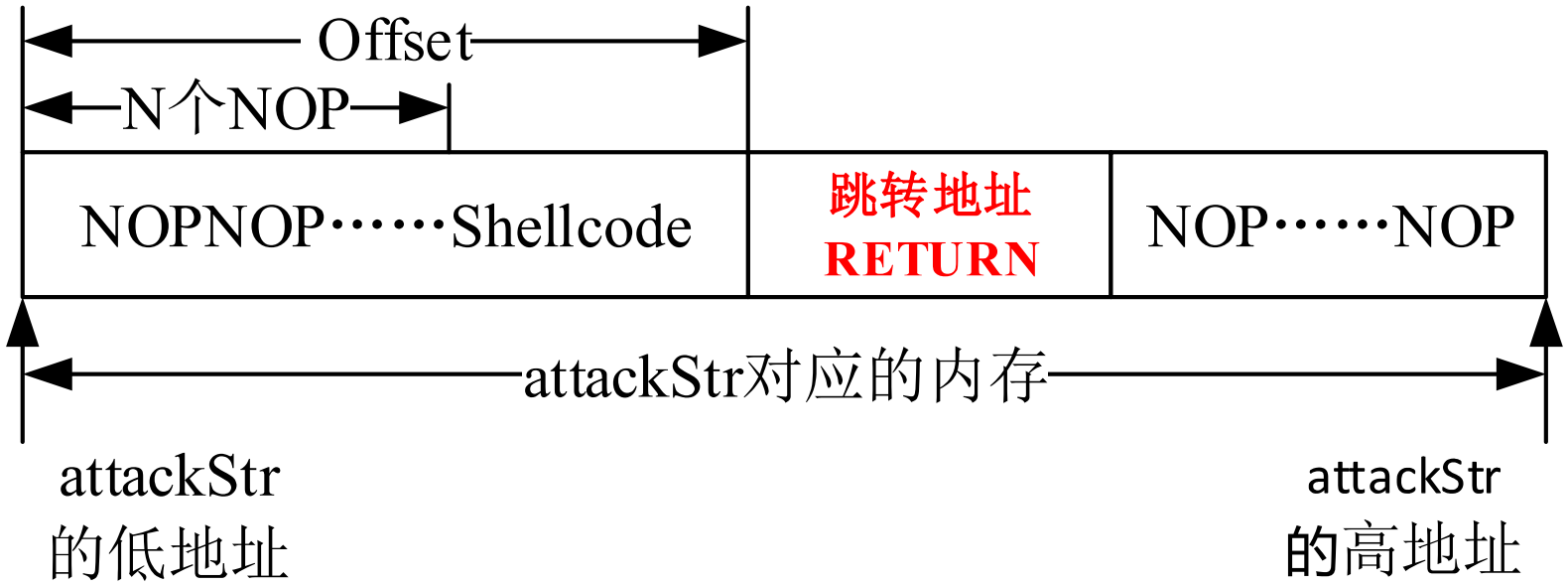
```
void foo()
{
    char buffer[LEN];
    strcpy (buffer, attackStr);
}
```

- 显然，若attackStr的内容过多，则上述代码会发生缓冲区溢出错误。在此buffer是被攻击的字符串，attackStr是攻击串。
- 假定attackStr是攻击者可以设置的，则有两种常用的方法构造attackStr。

方法一：将Shellcode放置在跳转地址(函数返回地址所在的栈)之前

- 如果被攻击的缓冲区(buffer)较大，足以容纳Shellcode，则可以采用这种方法。attackStr的内容按图7.6.2-1(a)的方式组织。
- 其中，Offset为被攻缓冲区(buffer)首地址与函数的返回地址所在栈地址的距离，需要通过gdb调试确定（见7.6.1）。对于老版本的Linux系统，跳转地址RETURN的值可通过gdb调试目标进程而确定。然而，现代的操作系统由于在内核使用了地址随机化技术，堆栈的起始地址是动态变化的，进程每次启动时均与上一次不同，只能猜测一个可能的地址。

7.6.2-1(a) 攻击串的构造



7.6.2-1(b): 即将执行strcpy之前buffer及栈的内容

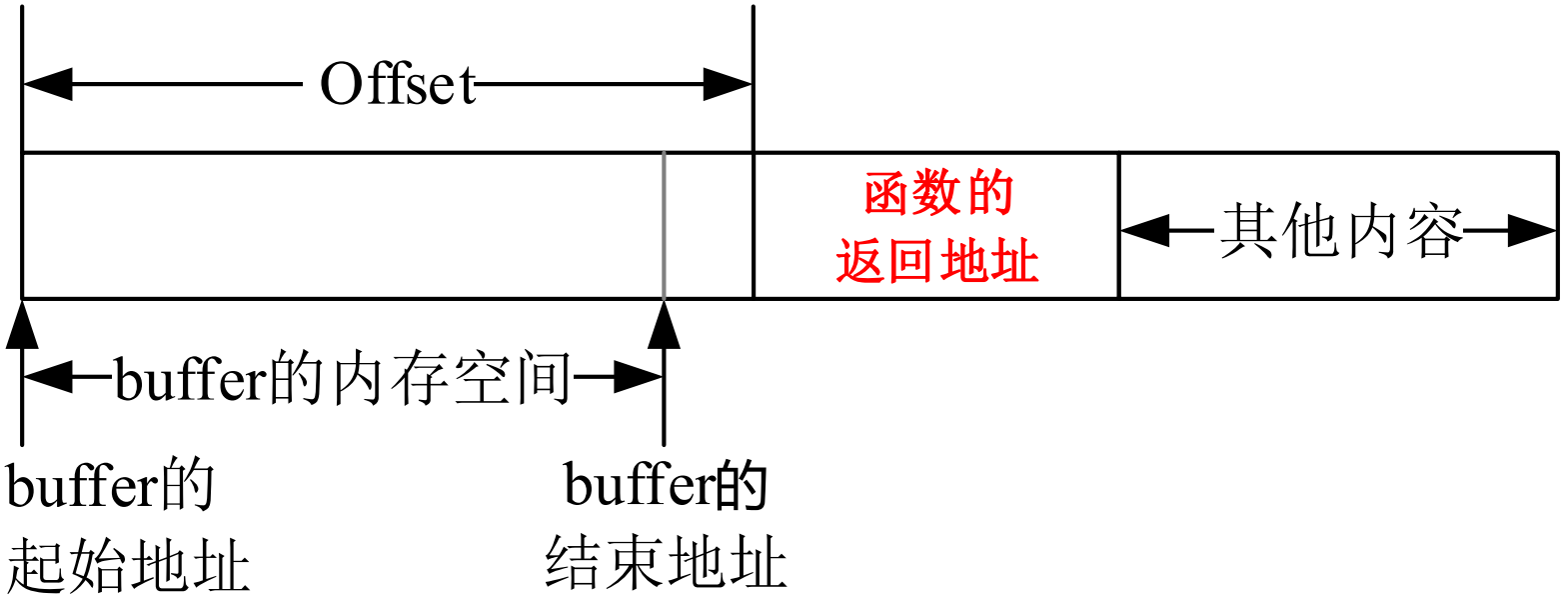


图7.6.2-2: 执行strcpy语句之后buffer及栈的内容

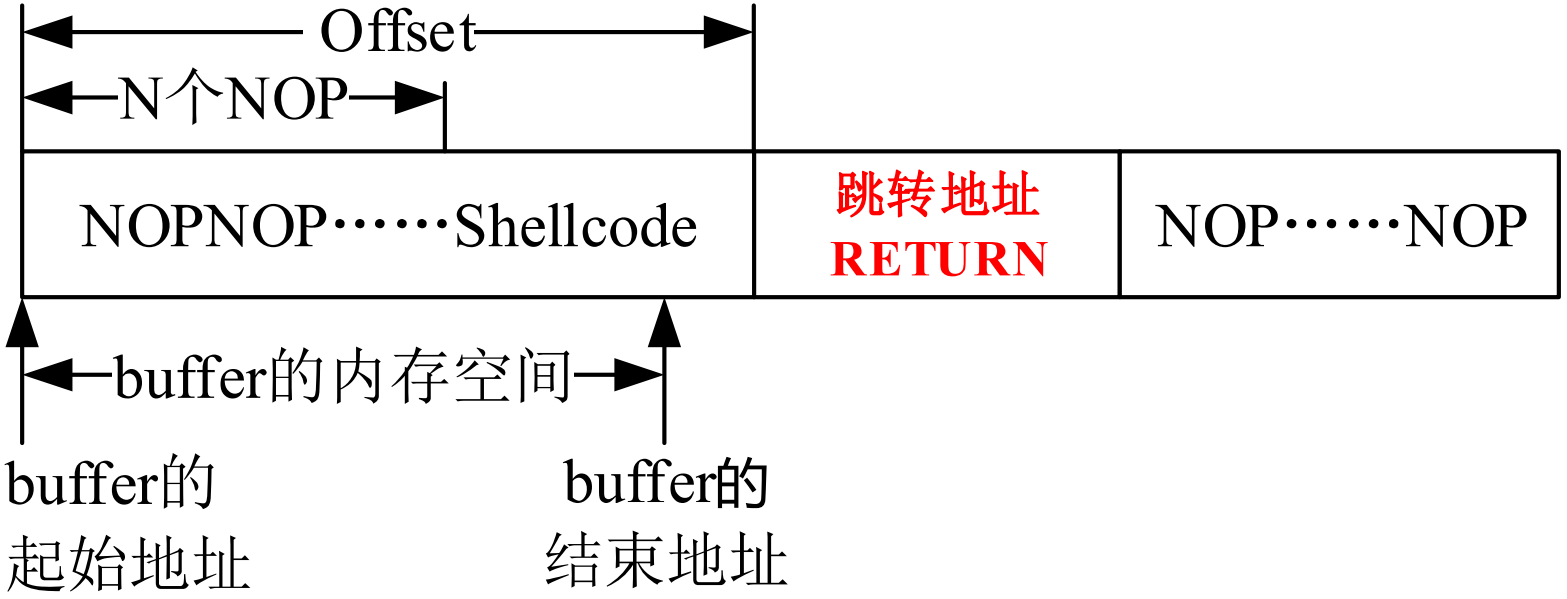


图7.6.2-1(a)中的跳转地址应按如下公式计算

RETURN=buffer的起始地址+n， 其中， 0<n<N

方法二：将Shellcode放置在跳转地址(函数返回地址所在的栈)之后

- 如果被攻击的缓冲区(buffer)的长度小于Shellcode的长度，不足以容纳shellcode，则只能将Shellcode放置在跳转地址之后。attackStr的内容按图7.6.2-3 (a)的方式组织。
- 即将执行strcpy (buffer, attackStr)语句时，buffer及栈的内容如图7.6.2-3(b)所示。执行strcpy (buffer, attackStr)语句之后，buffer及栈的内容如图7.6.2-3 -4所示。
- 图7.6.2-3 -3(a)中的跳转地址应按如下公式计算

  RETURN=buffer的起始地址+**Offset+4**+n，

  其中， 0<n<N

图7.6.2-3 (a) 攻击串的构造

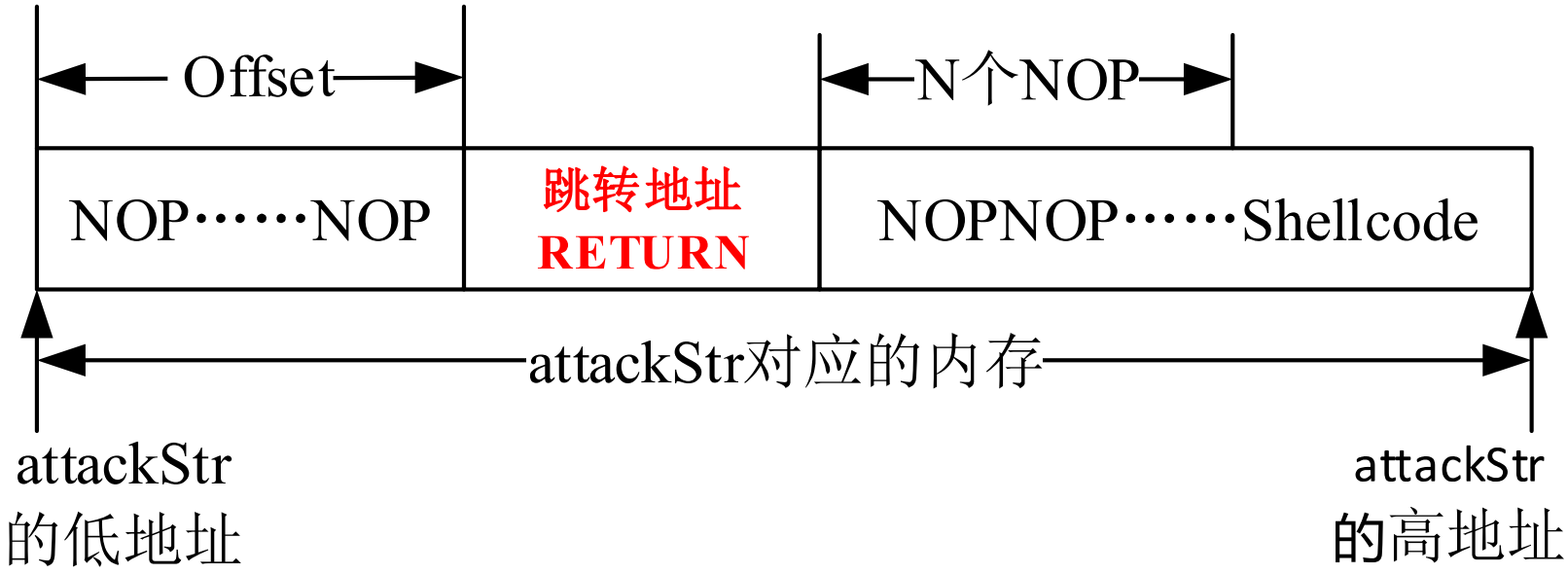


图7.6.2-3(b): 即将执行strcpy之前buffer及栈的内容

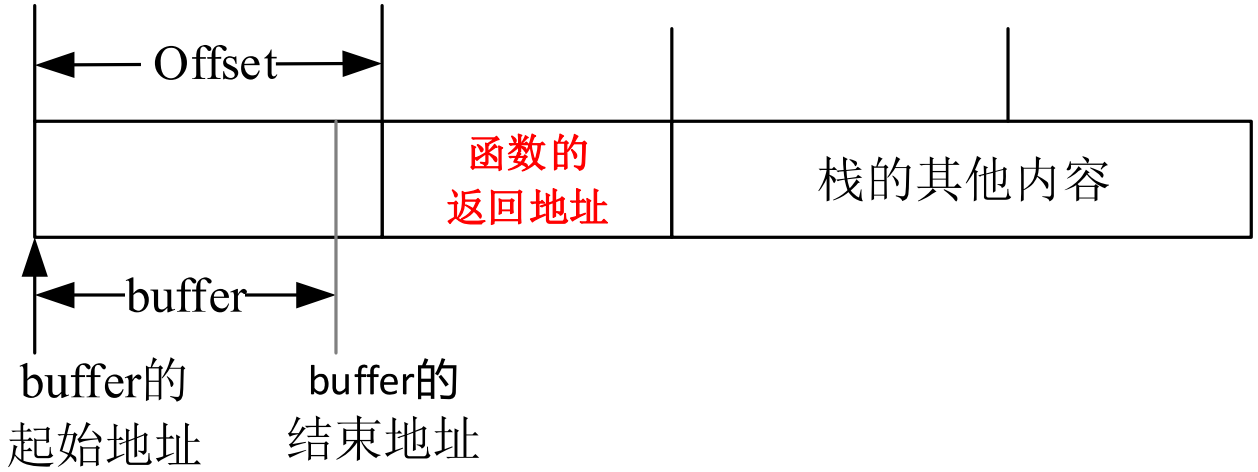
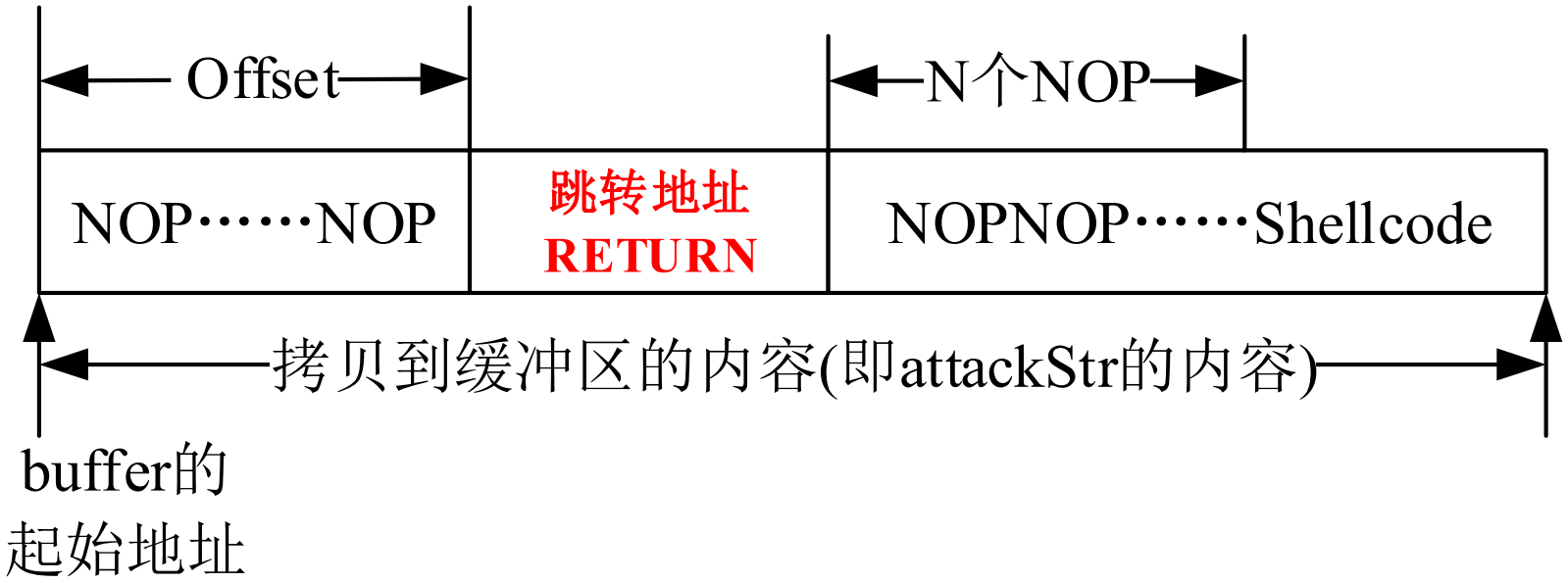


图7.6.2-4: 执行strcpy语句之后buffer及栈的内容



7.6.3 本地攻击实例

- 首先测试shellcode是否能成功运行。

```
int main(int argc, char * argv[])
{ run_shellcode();return 0; }
$ gcc -fno-stack-protector -o buf ../src/attack_overflow.c
$ ./buf
Segmentation fault (core dumped)
$ gcc -fno-stack-protector -z execstack -o buf ../src/attack_overflow.c
$ ./buf
$
```

- 组织攻击代码进行攻击
  - int main(int argc, char * argv[])
  - { SmashSmallBuf(); foo(); return 0; }
  - 见函数SmashSmallBuf()

第7章作业

- 作业
  1．简述网络攻击的一般流程。
  3．简述缓冲区溢出的基本原理。
  4．简述常见的拒绝服务攻击方法的原理。
  7．僵尸网络攻击大致包括哪些阶段？每个阶段的任务是什么？
  8.将attack_overflow.c中foo函数的char buff[16]改成char buff[163]，其他代码不变。通过调试确定buff的首地址与返回地址所在栈的距离Offset。

- 实践（自己研究，不考核）
  - 从http://www.nsfocus.net/index.php?act=sec_bug查阅几种缓冲区溢出漏洞。