软件安全实验

北京邮电大学信息安全中心
张淼
zhangmiao@bupt.edu.cn

第一讲 缓冲区溢出

- 缓冲区溢出概述
- 系统栈的工作原理
- 堆栈溢出实例分析：修改邻接变量

一、缓冲区溢出概述

- 缓冲区溢出原理简介
- 缓冲区溢出问题历史
- 缓冲区溢出的影响
- 缓冲区溢出的预防

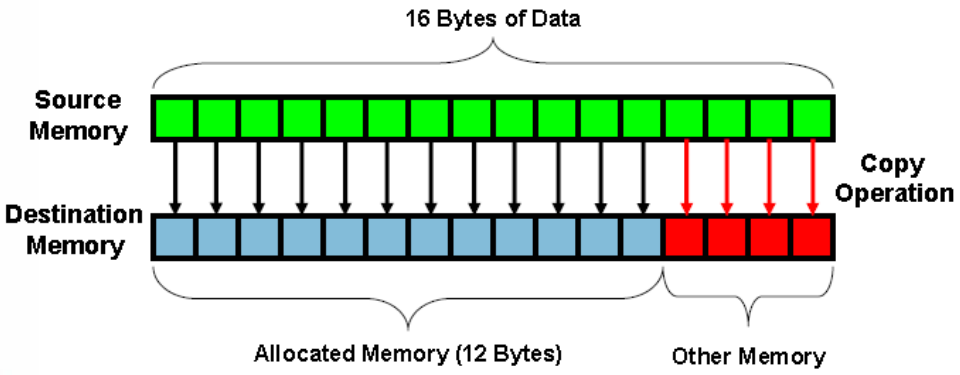
一、缓冲区溢出概述—缓冲区溢出原理简介

缓冲区溢出是指当计算机向缓冲区内填充数据位数时超过了缓冲区本身的容量溢出的数据覆盖在合法数据上。

这种错误的状态发生在写入内存的数据超过了分配给缓冲区的大小的时候,就像一个杯子只能盛一定量的水,如果放到杯子中的水太多,多余的水就会一出到别的地方。由于缓冲区溢出，相邻的内存地址空间被覆盖，造成软件出错或崩溃。如果没有采取限制措施，可以使用精心设计的输入数据使缓冲区溢出，从而导致安全问题。

一、缓冲区溢出概述—缓冲区溢出原理简介

缓冲区溢出是最常见的内存错误之一，也是攻击者入侵系统时所用到的最强大、最经典的一类漏洞利用的方式



缓冲区溢出图示

一、缓冲区溢出概述—缓冲区溢出相关历史

- 很长一段时间以来,缓冲区溢出都是一个众所周知的安全问题，C程序的缓冲区溢出问题早在70年代初就被认为是C语言数据完整性模型的一个可能的后果。这是因为在初始化、拷贝或移动数据时，C语言并不自动地支持内在的数组边界检查。虽然这提高了语言的执行效率，但其带来的影响及后果却是深远和严重的。
- 1988年 Robert T. Morris的finger蠕虫程序.这种缓冲区溢出的问题使得Internet几乎限于停滞,许多系统管理员都将他们的网络断开,来处理所遇到的问题.
- 1989年 Spafford提交了一份关于运行在VAX机上的BSD版UNIX的fingerd的缓冲区溢出程序的技术细节的分析报告，引起了部分安全人士对这个研究领域的重视

一、缓冲区溢出概述—缓冲区溢出相关历史

- 1996年 出现了真正有教育意义的第一篇文章，Aleph One在Underground发表的论文详细描述了Linux系统中栈的结构和如何利用基于栈的缓冲区溢出。
- Aleph One的贡献还在于给出了如何写开一个shell的Exploit的方法，并给这段代码赋予shellcode的名称，而这个称呼沿用至今，我们现在对这样的方法耳熟能详--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。
- 1997年 Smith综合以前的文章，提供了如何在各种Unix变种中写缓冲区溢出Exploit更详细的指导原则。

一、缓冲区溢出概述—缓冲区溢出相关历史

- 1998年　来自“Cult of the Dead Cow”的Dildog在Bugtrq邮件列表中以Microsoft Netmeeting为例子详细介绍了如何利用Windows的溢出，这篇文章最大的贡献在于提出了利用栈指针的方法来完成跳转，返回地址固定地指向地址，将Windows下的溢出Exploit推进了实质性的一步。

一、缓冲区溢出概述—缓冲区溢出的影响

- 不要小看缓冲区溢出问题,它可能会产生很恶劣的影响:
  - 发布安全报告以及相关补丁的时间和费用开销
  - 数以千计的系统管理员安装补丁的时间开销
  - 系统被攻击者破坏所造成的损失
  - 重要资料被盗取造成的损失
  - 开发者的信誉….

- 粗略的算下来,一个马虎的错误就可能需要花费几百万美元的代价才能弥补

一、缓冲区溢出概述—缓冲区溢出的预防

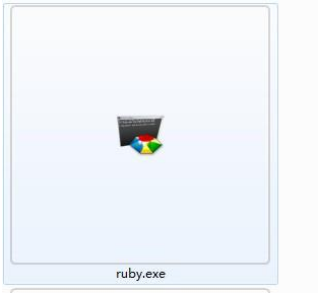
- 面临越来越多的来自缓冲区溢出攻击的威胁,例如Immunix根据自动侦测和预防技术开发的StackGuard系统之类的侦测和防止缓冲区溢出发生的自适应技术被研发出来.防范溢出问题正受到越来越多的关注.
- 目前对于缓冲区溢出,主要分为静态保护和动态保护:
- **静态保护:**不执行代码,通过静态分析来发现代码中可能存在的漏洞.静态的保护技术包括编译时加入限制条件,返回地址保护,二进制改写技术,基于源码的代码审计等.
- **动态保护**:通过执行代码分析程序的特性,测试是否存在漏洞,或者是保护主机上运行的程序来防止来自外部的缓冲区溢出攻击.

二、系统栈的工作原理

- PE文件格式简介
- 虚拟内存相关知识
- 内存的不同用途
- 栈与系统栈
- 函数调用时发生了什么
- 寄存器与函数栈帧
- 函数调用约定与相关指令

PE文件格式简介

- PE(Portable Executable)是Win32平台下可执行文件遵守的数据格式。常见的可执行文件(如“*.exe”文件和“*.dll”文件)都是典型的PE文件。



- 一个可执行文件不光包含了二进制的机器代码，还会子弹许多其他信息，如字符串、菜单、图标、位图、字体等。

PE文件格式简介

- PE文件格式规定了所有的这些信息在可执行文件中如何组织。在程序被执行时，操作系统会按照PE文件格式的约定去相应的地方准确地定位各种类型的资源，并分别装入内存的不同区域。

- PE文件格式把可执行文件分成若干个数据节(section)，不通的资源被存放在不同的节中。一个典型的PE文件包含的节如下：

PE文件格式简介

- .text　　由编译器产生，存放着二进制的机器代码，也是我们反汇编和调试的对象。
- .data　　初始化的数据块，如宏定义、全局变量、静态变量等。
- .idata　　可执行文件所使用的动态链接库等外来函数与文件的信息。
- .rsrc　　存放程序的资源，如图标、菜单等。
- 除此之外，还可能出现的节包括“.reloc”、“.edata”、“.tls”、“.rdata”等。

三、PE文件格式简介

PE文件简单构成

虚拟内存相关知识

- 虚拟内存简介
- 内存管理与银行的类比
- PE文件与虚拟内存之间的映射

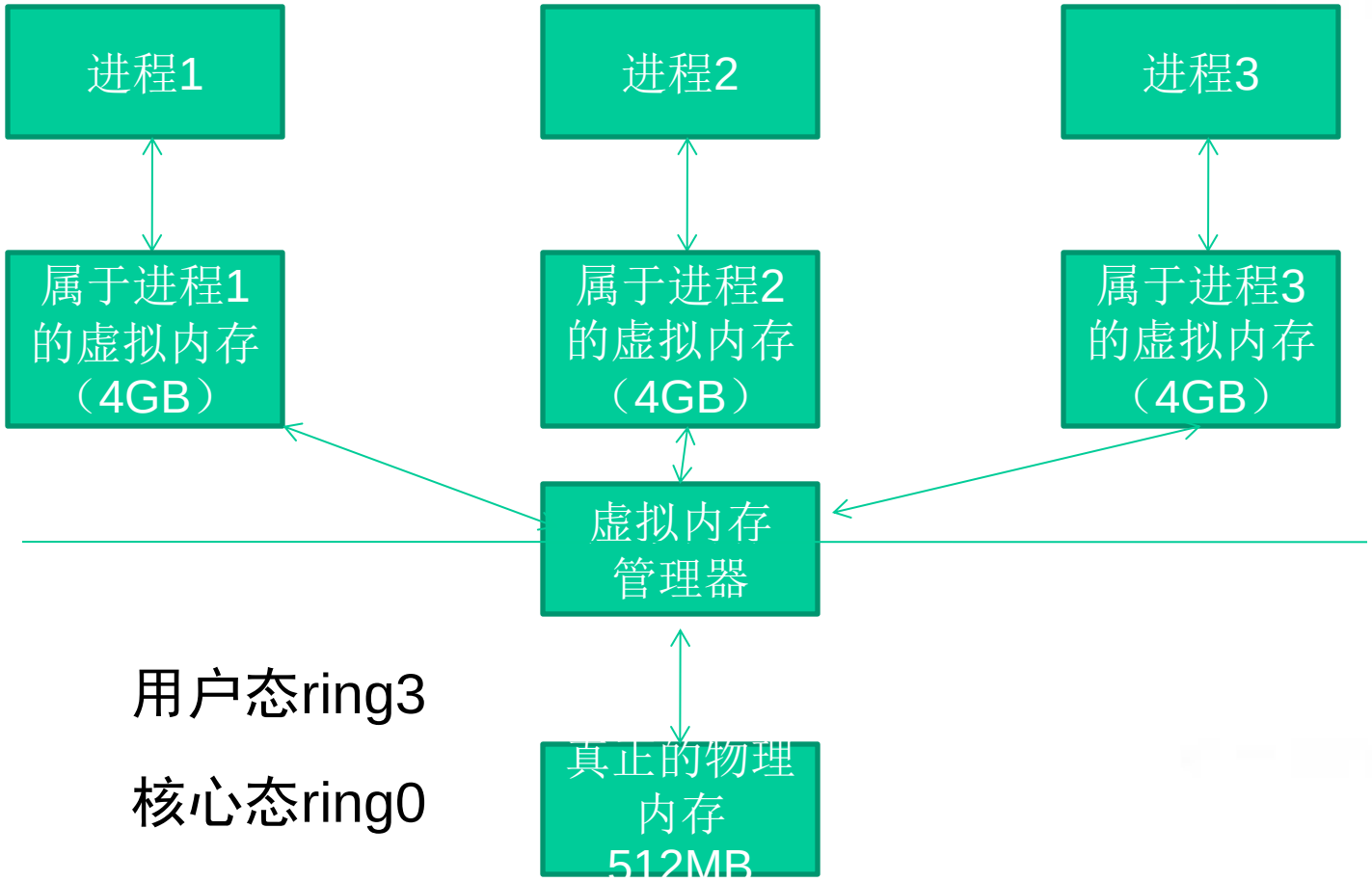
虚拟内存相关知识—虚拟内存简介

- Windows的内存可以被分为两个层面：物理内存和虚拟内存。其中，物理内存比较复杂，需要进入Windows内核级别ring0才能看到。通常，在用户模式下，我们用调试器看到的地址都是虚拟内存。

虚拟内存相关知识—虚拟内存简介

- Windows让所有的进程都“相信”自己用友独立的4GB内存空间。但是我们计算机中那跟实际的内存条可能只有512MB,怎么能为所有进程都分配4GB的内存呢？这一切都是通过虚拟内存管理器的映射做到的。

虚拟内存相关知识—虚拟内存简介



用户态ring3

核心态ring0

虚拟内存相关知识—虚拟内存简介

- 虽然每个进程都“相信”自己拥有4GB的空间，但实际上它们运行时真正能用到的空间根本没有那么多。
- 内存管理器只是分给进程一片“假地址”，或者说是“虚拟地址”，它们对进程来说只是一笔“无形的数字财富”；
- 当需要实际的内存操作时，内存管理器才会把“虚拟地址”和“物理地址”联系起来。

虚拟内存相关知识—内存管理与银行的类比

我们将银行与内存管理机制进行一下类比来帮助大家理解

内存管理	银行类比
进程	储户
内存管理器	银行
物理内存	钞票
虚拟内存	存款
进程可能拥有大片内存，但使用往往很少	储户拥有大笔存款，但实际生活中的开销并没多少

虚拟内存相关知识—内存管理与银行的类比

我们将银行与内存管理机制进行一下类比来帮助大家理解

内存管理	银行类比
进程不使用虚拟内存时，这些内存只是些地址，是虚拟存在的，是一笔无形的数字财富。	用户不使用储蓄时，储蓄也只是一些数字，是无形的数字财富。
进程使用内存时，内存管理器会为器会为这个虚拟地址映射实际的物理地址，虚拟内存地址和最终被映射到的物理内存地址之间没有什么必然联系	储户需要钱时，银行才会兑换一定的现金给储户，但物理钞票的号码与储户心目中的数字存款之间可能并没有任何联系。

虚拟内存相关知识—内存管理与银行的类比

我们将银行与内存管理机制进行一下类比来帮助大家理解

内存管理	银行类比
操作系统的实际物理内存空间可以远远小于进程的虚拟内存空间之和，仍能正常调度	银行的现金准备可以远远小于所有储户的储蓄额总和，仍能正常运转
很少出现所有程序要申请出全部物理内存	很少会出现所有储户同时要取出全部存款的现象
物理内存可以远小于虚拟内存之和	社会上实际流通的钞票可以远远小于社会的财富总额

虚拟内存相关知识— PE文件与虚拟内存之间的映射

- 静态反汇编工具看到的PE文件中某条指令的位置是相对于磁盘文件而言的，即所谓的文件偏移，我们可能还需要知道这条指令在内存中所处的位置，即虚拟内存的位置

- 反之，在调试时看到的某条指令的地址是虚拟内存地址，我们也经常需要回到PE文件中找到这条指令对应的机器码

虚拟内存相关知识— PE文件与虚拟内存之间的映射

我们首先要弄清楚几个概念

- 文件偏移地址(File Offset)
  - 数据在PE文件中的地址叫做文件偏移地址。这是文件在磁盘上存放时相对于文件开头的偏移。

- 装载基址(Image Base)
  - PE装入内存时的基地址。默认情况下，EXE文件在内存中的基地址是0x00400000, DLL文件是0x10000000。这些位置可以通过修改编译选项更改。

虚拟内存相关知识— PE文件与虚拟内存之间的映射

我们首先要弄清楚几个概念

- 虚拟内存地址(Virtual Address,VA)
  - PE文件中的指令被装入内存后的地址。

- 相对虚拟地址(Relative Virtual Address,RVA)
  - 相对虚拟地址是内存地址相对于映射基址的偏移量。

➔ 虚拟内存地址、映射基址、相对虚拟内存地址三者有如下关系

VA=Image Base + RVA

虚拟内存相关知识— PE文件与虚拟内存之间的映射

文件偏移地址在与他们计算时还需要考虑存放方式的不同。

- PE文件中的数据按照磁盘数据标准存放，以0x200字节为基本单位进行组织。当一个数据节不足0x200字节时，不足的地方将被0x00填充；当一个数据节超过0x200字节时，下一个0x200块将分配给这个节使用。因此PE数据节的大小永远是0x200的整数倍。

虚拟内存相关知识— PE文件与虚拟内存之间的映射

文件偏移地址在与他们计算时还需要考虑存放方式的不同。

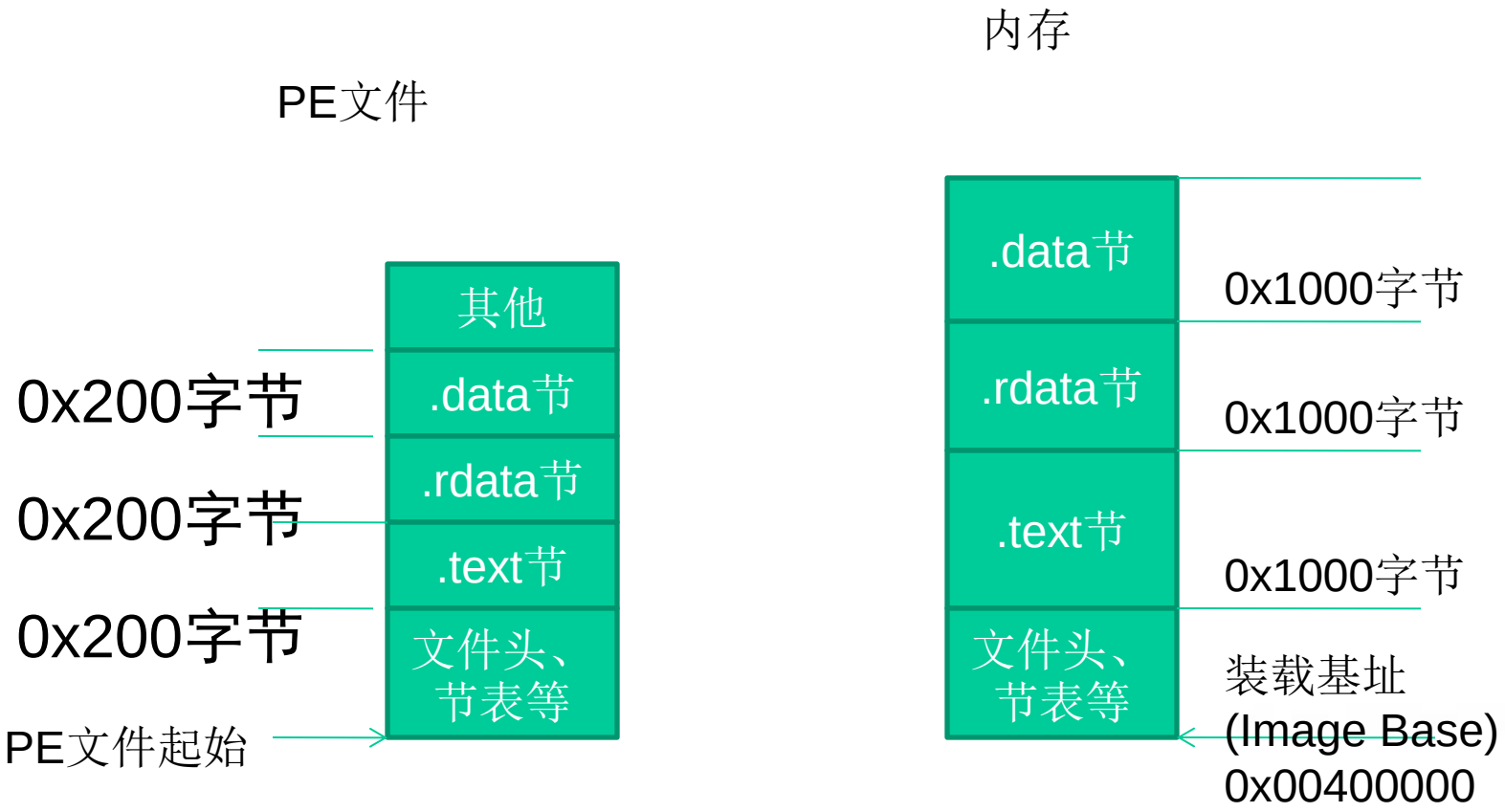
- 当代码装入内存后，将按照内存数据标准存放，并以0X1000字节为基本单位进行组织。类似的，不足将被不全，若超出将分配下一个0x1000为其所用。因此，内存中的节总是0x1000的整数倍。

虚拟内存相关知识— PE文件与虚拟内存之间的映射

文件偏移地址在与他们计算时还需要考虑存放方式的不同。

- 当代码装入内存后，将按照内存数据标准存放，并以0X1000字节为基本单位进行组织。类似的，不足将被不全，若超出将分配下一个0x1000为其所用。因此，内存中的节总是0x1000的整数倍。

虚拟内存相关知识— PE文件与虚拟内存之间的映射



PE文件与虚拟内存的映射关系

虚拟内存相关知识— PE文件与虚拟内存之间的映射

节(section)	相对虚拟偏移量(RVA)	文件偏移量
.text	0x00001000	0x0400
.rdata	0x00007000	0x6200
.data	0x00009000	0x7400
.rsrc	0x0002D000	0x7800

我们把这种由存储单位差异引起的节基址差称做节偏移，在上图例中：

.text节偏移 = 0x1000-0x400 = 0xc00

.rdata节偏移 = 0x7000-0x6200 = 0xE00

.data节偏移 = 0x9000-0x7400 = 0x1c00

.rsrc节偏移 = 0x2D000-0x7800 = 0x25800

文件偏移地址 = 虚拟内存地址(VA) – 装载基址(Image Base)-节偏移

= RVA – 节偏移

以上表为例，如果在调试时遇到虚拟内存中0x00404141处的一条指令，那么要换算出这条指令在文件中的偏移量，有：

文件偏移量= 0x00404141-0x00400000-(0x1000-0x400)=0x3541

二、系统栈的工作原理—内存的不同用途

- 成功地利用缓冲区溢出漏洞可以修改内存中的变量的值，甚至可以劫持进程，执行恶意代码，最终获得主机的控制券。要透彻地理解这种攻击方式，我们需要回顾一些计算机体系架构的基础知识，搞清楚CPU、寄存器、内存是怎样协同工作而让程序流畅执行的。

寄存器

Intel x86的寄存器可以分为下述的几类：

- 通用寄存器
  - 32位的通用寄存器有*EAX*、*EBX*、*ECX*、*EDX*、*ESP*、*EBP*、*ESI*和*EDL*，它们的使用方法不总是相同的。一些指令赋予它们特殊的功能。
- 段寄存器
  - 段寄存器被用于指向进程地址空间不同的段。
  - *CS*指向一个代码段的开始；
  - *SS*是一个堆栈段；
  - *DS*、*ES*、*FS*、*GS*和各种其他数据段，例如存储静态数据的段。
- 程序流控制寄存器
- 其他寄存器

- 在通用寄存器里面有很多寄存器虽然他们的功能和使用没有任何的区别，但是在长期的编程和使用中，在程序员习惯中已经默认的给每个寄存器赋上了特殊的含义，比如：
  - EAX一般用来做返回值
  - ECX用于记数
  - EIP：扩展指令指针。在调用一个函数时，这个指针被存储在堆栈中，用于后面的使用。在函数返回时，这个被存储的地址被用于决定下一个将被执行的指令的地址。
  - ESP：扩展堆栈指针。这个寄存器指向堆栈的当前位置，并允许通过使用push和pop操作或者直接的指针操作来对堆栈中的内容进行添加和移除。
  - EBP：扩展基指针。主要用与存放在进入call以后的ESP的值，便于退出的时候回复ESP的值，达到堆栈平衡的目的。

二、系统栈的工作原理—内存的不同用途

根据不同的操作系统，一个进程可能被分配到不同的内存区域去执行。但是不管什么样的操作系统、什么样的计算机架构，进程使用的内存都可以按照功能大致分成以下4个部分。

二、系统栈的工作原理—内存的不同用途

名称	用途
代码区	这个区域存储着被装入执行的二进制机器代码，处理器会到这个区域取指并执行。
数据区	用于存储全局变量等。
堆区	进程可以在堆区动态地请求一定大小的内存，并在用完之后归还给堆区。动态分配和回收是堆区的特点。
栈区	用于动态地存储函数之间的调用关系，以保证被调用函数在返回时恢复到父函数中继续执行。

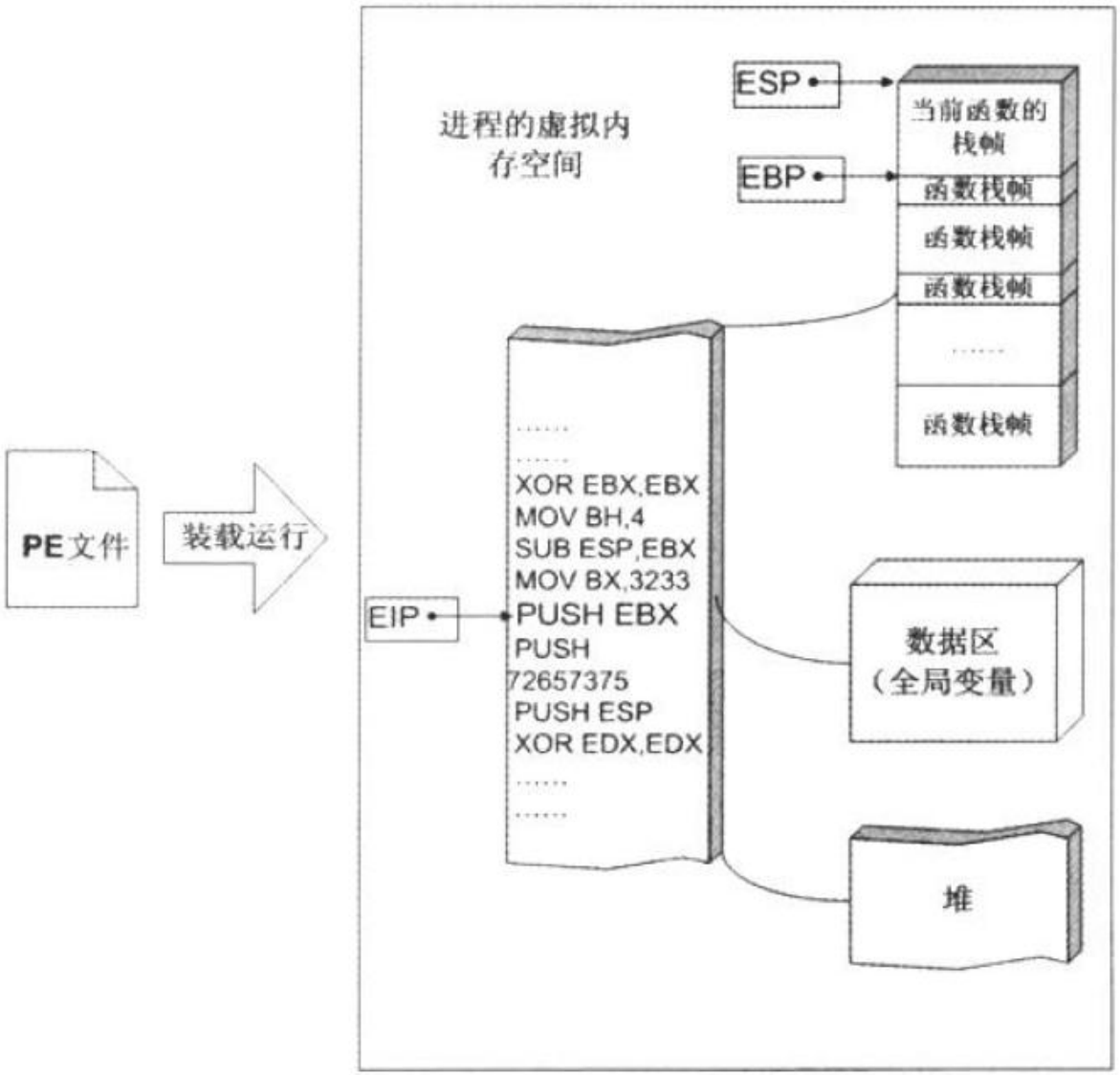
内存的4个部分和相关用途

二、系统栈的工作原理—内存的不同用途

如果把计算机看成一个有条不紊的工厂，我们可以得到如下类比。

名称	类比
CPU	完成工作的工人
数据区、堆区、栈区等	存放原料、半成品、成品等各东西的场所。
存在代码区的指令	告诉CPU要做什么，怎么做，到哪里去领原料，用什么工具来做，做完以后把成品放到哪个货舱去。
栈区	栈除了扮演存放原料、半成品的仓库之外，它还是车间调度主任的办公室。

内存与工厂的类比关系


进程的虚拟内存空间
ESP
EBP
当前函数的栈帧
函数栈帧
函数栈帧
函数栈帧
.....
函数栈帧
PE文件
装载运行
EIP
.....
.....
XOR EBX,EBX
MOV BH,4
SUB ESP,EBX
MOV BX,3233
PUSH EBX
PUSH 72657375
PUSH ESP
XOR EDX,EDX
.....
.....
数据区（全局变量）
堆

二、系统栈的工作原理—内存的不同用途

程序中所使用的缓冲区可以是堆区、栈区和存放静态变量的数据区。缓冲区溢出的利用方法和缓冲区到底属于上面哪个内存区域密不可分，本课主要介绍在系统栈发生一出的情形。

二、系统栈的工作原理—栈与系统栈

从计算机科学的角度来看，栈指的是一种数据结构，是一种先进后出的数据表。

栈的最常见操作有两种：压栈(PUSH)、弹栈(POP)；用于标识栈的属性也有两个：栈顶(TOP)、栈底(BASE)。

二、系统栈的工作原理—栈与系统栈

为了便于理解，我们把栈和扑克牌进行一下类比

操作名称	类比效果
PUSH	为栈增加一个元素的操作叫做PUSH，这相当于在这摞扑克牌的最上面再放上一张牌。
POP	从栈中取出一个元素的操作叫做POP,相当于从这摞扑克牌取出最上面的一张。

栈操作与扑克的类比

二、系统栈的工作原理—栈与系统栈

假如我们把栈想象成一摞扑克牌：

位置名称	类比效果
TOP	标识栈顶的位置，并且是动态变化的。每做一次PUSH操作，它都会自增1；相反，每做一次POP操作，它会自减1。栈顶元素相当于扑克牌最上面一张，只有这张牌的花色是当前可以看到的。
BASE	标识栈底的位置，它记录着扑克牌最下面一张的位置。BASE用于防止栈空后继续弹栈。一般情况下它的值是不会改变的。

栈位置属性与扑克牌的类比

二、系统栈的工作原理—栈与系统栈

假如我们把栈想象成一摞扑克牌：

PUSH操作

POP操作

栈位置属性与扑克牌的类比

二、系统栈的工作原理—栈与系统栈

内存的栈区实际上指的就是系统栈。系统栈由系统自动维护，它用于实现高级语言中函数的调用。对于类似C语言这样的高级语言，系统栈的PUSH/POP等堆栈平衡细节是透明的。

一般说来，只有在使用汇编语言开发程序的时候，才需要和它直接打交道。

二、系统栈的工作原理—函数调用时发生了什么

我们假定有如下程序

```
int funcb()
{…
}
int funca()
{…
  funcb();
…
}
```

二、系统栈的工作原理—函数调用时发生了什么

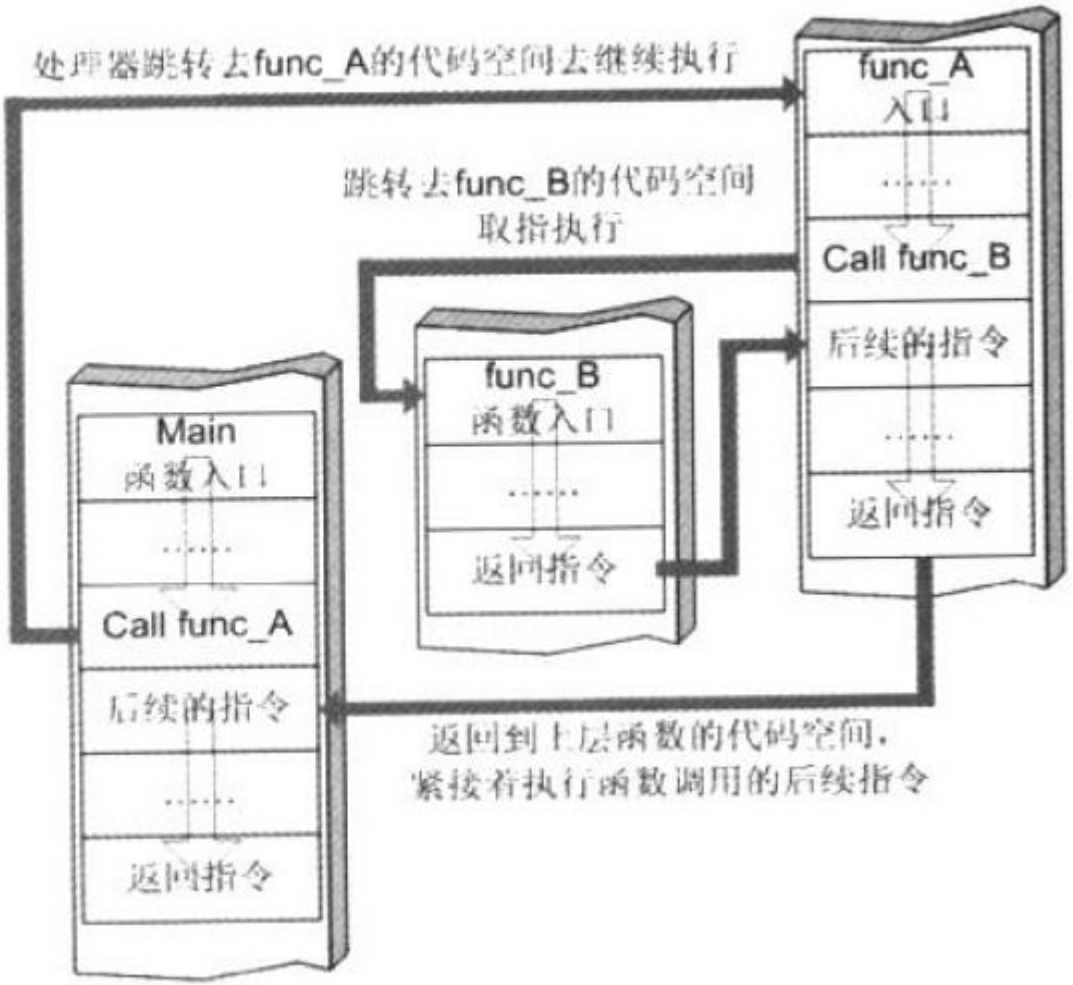
我们假定有如下程序

```
int main()
{
  funca();
}
```

系统栈在函数调用时的变化是什么样的呢？

代码空间调用关系


……
（其他指令）
……
（func_B的指令）
……
……
（main函数指令）
……
……
……
……
（其他指令）
……
（func_A的指令）
……
……
……


处理器跳转去func_A的代码空间去继续执行
func_A
入口
……
Call func_B
后续的指令
……
返回指令
跳转去func_B的代码空间
取指执行
func_B
函数入口
……
返回指令
Main
函数入口
……
Call func_A
后续的指令
……
返回指令
返回到上层函数的代码空间，
紧接着执行函数调用的后续指令

二、系统栈的工作原理—函数调用时发生了什么

系统栈在函数调用时的变化

代码空间：程序被装入，由main函数代码空间依次取指执行

系统栈空间：系统栈栈顶为当前正在执行的main函数栈帧

代码空间：执行到main代码区的call指令时，跳转到funca的代码区继续执行

系统栈空间：为配合funca的执行，在系统栈中为其开辟新的栈帧并压入

二、系统栈的工作原理—函数调用时发生了什么

系统栈在函数调用时的变化

代码空间：执行到funca代码区的call指令时，跳转到funcb的代码区继续执行

系统栈空间：为配合funcb的执行，在系统栈中为其开辟新的栈帧并压入

代码空间：funcb代码执行完毕，弹出自己的栈帧并从中获得返回地址，跳回funca代码区继续执行

系统栈空间：弹出funcb的栈帧。对应于当前正在执行的函数，当前栈顶栈帧重新恢复成funca函数栈帧

二、系统栈的工作原理—函数调用时发生了什么

系统栈在函数调用时的变化

代码空间：funca代码执行完毕，弹出自己的栈帧并从中获得返回地址，跳回main代码区继续执行

系统栈空间：弹出funca的栈帧。对应于当前正在执行的函数，当前栈顶栈帧重新恢复成main函数栈帧

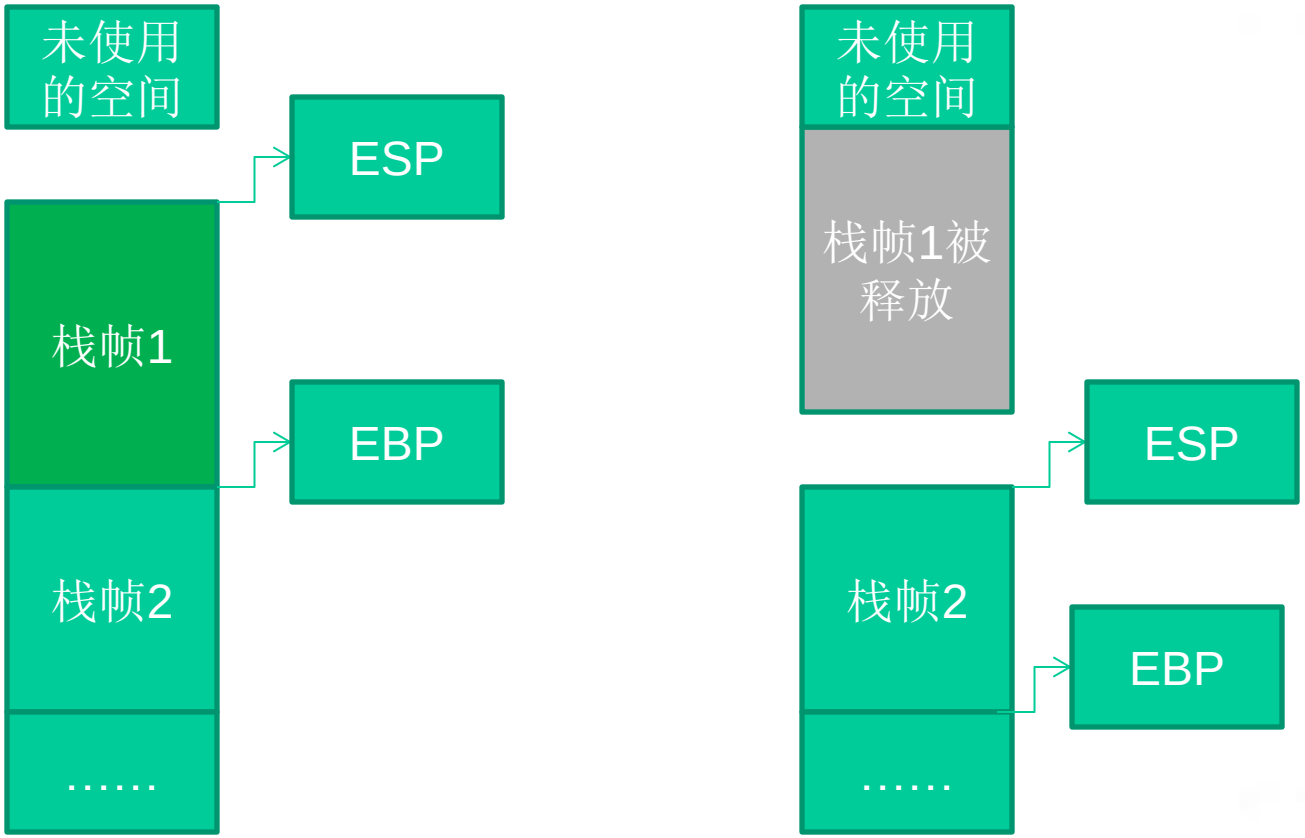
二、系统栈的工作原理—寄存器与函数栈帧

每一个函数独占自己的栈帧空间。当前正在运行的函数的栈帧总是在栈顶。Win32系统提供两个特殊的寄存器用于标识位于系统栈顶端的栈帧。

（1）ESP：栈指针寄存器(extended stack pointer),其内存放着一个指针，该指针永远指向系统栈最上面的一个栈帧的栈顶。

（2）EBP：基址指针寄存器(extended base pointer),其内存放着一个指针，该指针永远指向系统栈最上面的一个栈帧的底部。

二、系统栈的工作原理—寄存器与函数栈帧



栈帧寄存器ESP与EBP的作用

二、系统栈的工作原理—寄存器与函数栈帧

在函数栈帧中，一般包含以下几类重要信息。

（1）局部变量：为函数局部变量开辟的内存空间。

（2）栈帧状态值：保存前栈帧的顶部和底部（实际上只保存前栈帧的底部，前栈帧的顶部可以通过堆栈平衡计算得到），用于在本帧被弹出后恢复出上一个栈帧。

（3）函数返回地址：保存当前函数调用前的“断点”信息，也就是函数调用前的指令位置，以便在函数返回时能够恢复到函数被调用前的代码区中继续执行指令。

二、系统栈的工作原理—寄存器与函数栈帧

除了与栈相关的寄存器外，我们还需要记住另一个至关重要的寄存器。

EIP：指令寄存器(Extended Instruction Pointer),其内存放着一个指针，该指针永远指向一条等待执行的指令地址。

可以说如果控制了EIP寄存器的内容，就控制了进程---我们让EIP指向哪里，CPU就会去执行哪里的指令。

二、系统栈的工作原理—函数调用约定与相关指令

函数调用大致包括以下几个步骤。

（1）参数入栈：将参数从右向左一次压入系统栈中。

↓

（2）返回地址入栈：将当前代码区调用指令的下一跳指令地址压入栈中，供函数返回时继续执行。

↓

（3）代码区跳转：处理器从当前代码区跳转到被调用函数的入口处。

↓

（4）栈帧调整

二、系统栈的工作原理—函数调用约定与相关指令

第四步栈帧调整具体包括如下几个步骤。

保存当前栈帧的状态值，以备后面恢复本栈帧时使用（EBP入栈）；

↓

将当前栈帧切换到新栈帧（将ESP值装入EBP，更新栈帧底部）；

↓

给新栈帧分配空间（把ESP减去所需空间的大小，抬高栈帧）；

二、系统栈的工作原理—函数调用约定与相关指令

对于__stdcall调用约定，函数调用时用到的指令序列大致如下。

序列	备注
push 参数 3	假设该函数有3个参数，将从右向左依次入栈
push 参数 2	
push 参数 1	
call 函数地址	call指令将同时完成两项工作:a)向栈中压入当前指令在内存中的位置，即保存返回地址。 b)跳转到所调用函数的入口地址函数入口处

__stdcall调用约定的指令序列

二、系统栈的工作原理—函数调用约定与相关指令

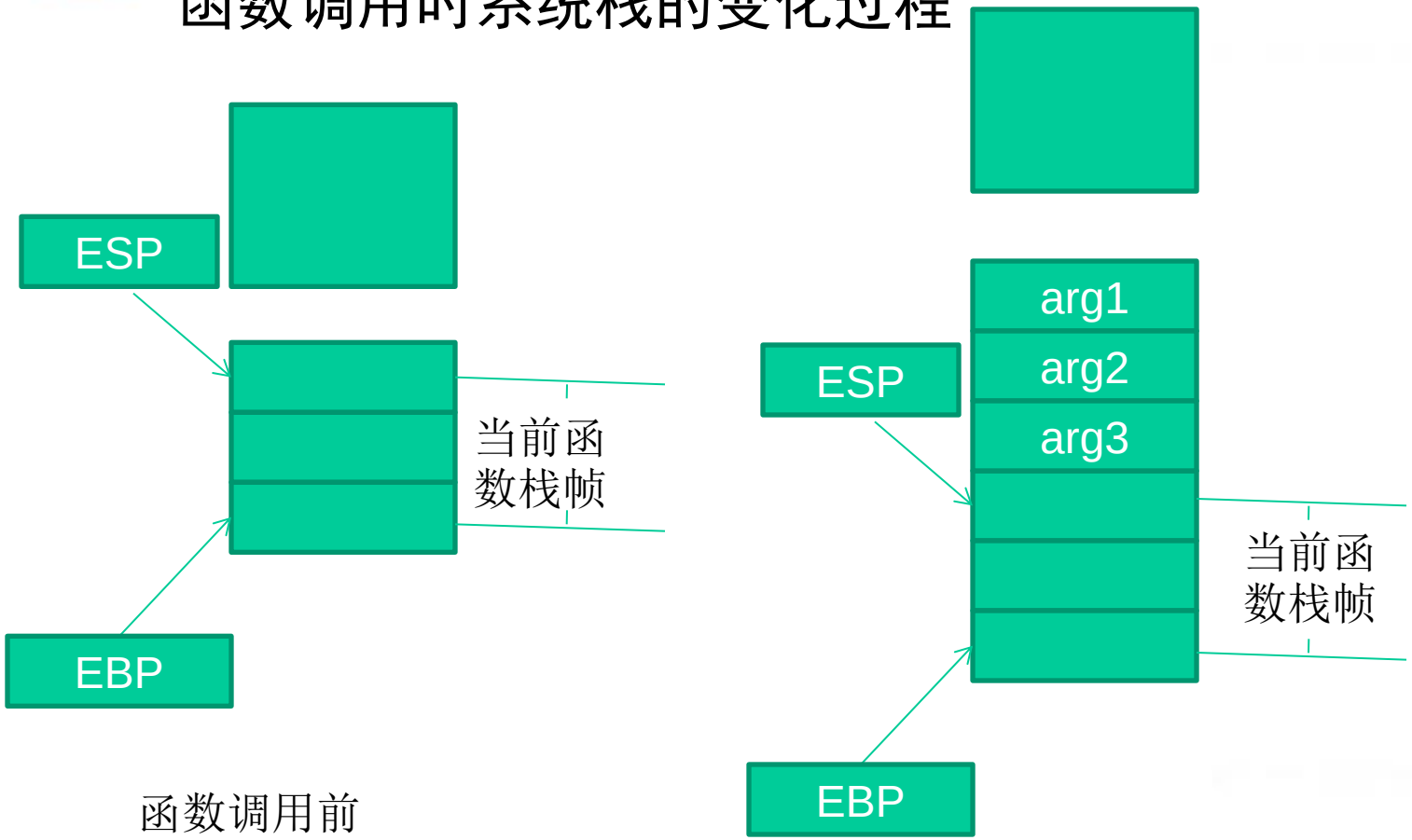
对于__stdcall调用约定，函数调用时用到的指令序列大致如下。

序列	备注
push ebp	保存旧栈帧的底部
mov ebp,esp	设置新栈帧的底部(栈帧切换)
sub esp,xxx	设置新栈帧的顶部(抬高栈顶，为新栈帧开辟空间）

__stdcall调用约定的指令序列

二、系统栈的工作原理—函数调用约定与相关指令

函数调用时系统栈的变化过程

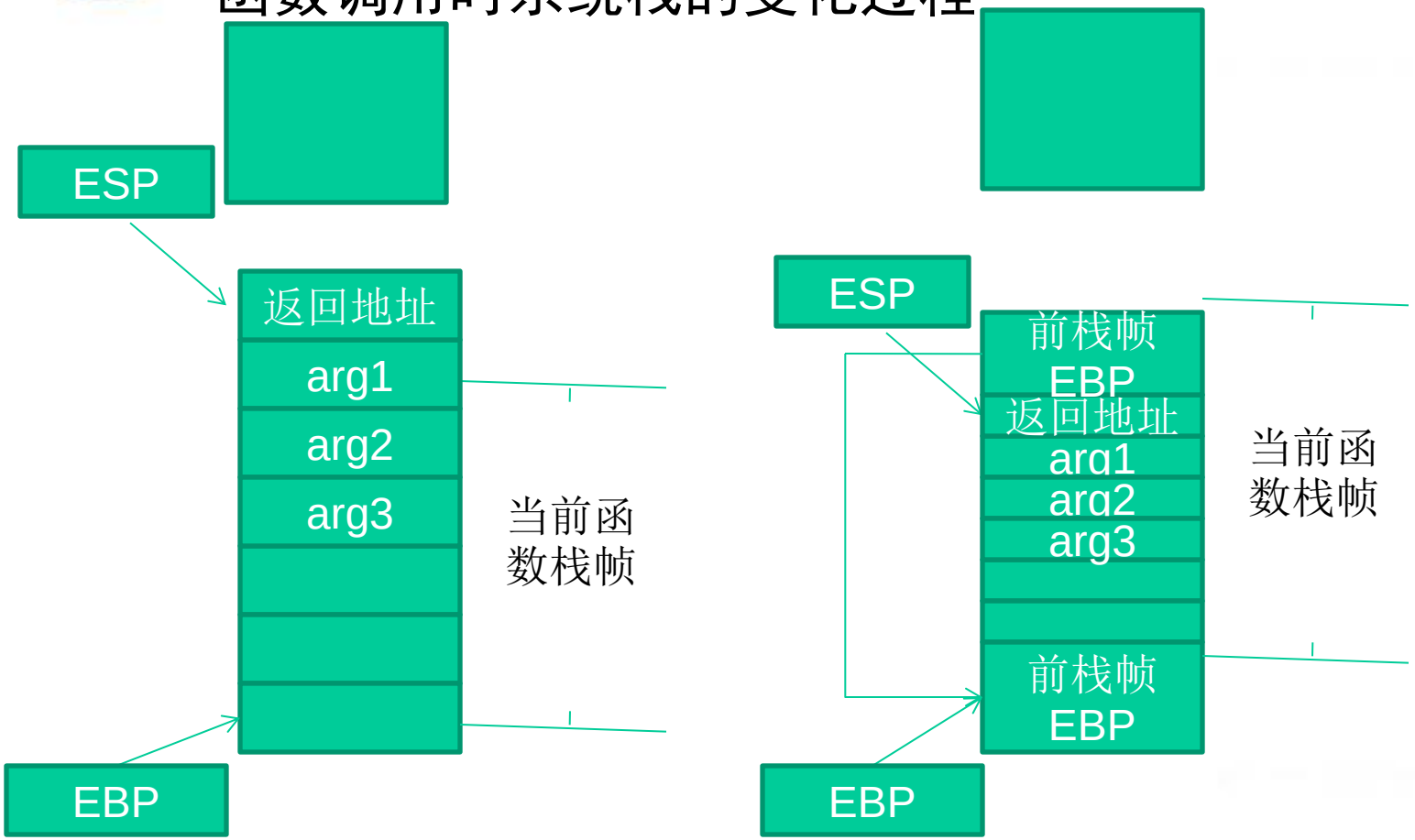


PUSH arg_x
参数从右向左一次入栈

__stdcall调用约定的指令序列

二、系统栈的工作原理—函数调用约定与相关指令

函数调用时系统栈的变化过程



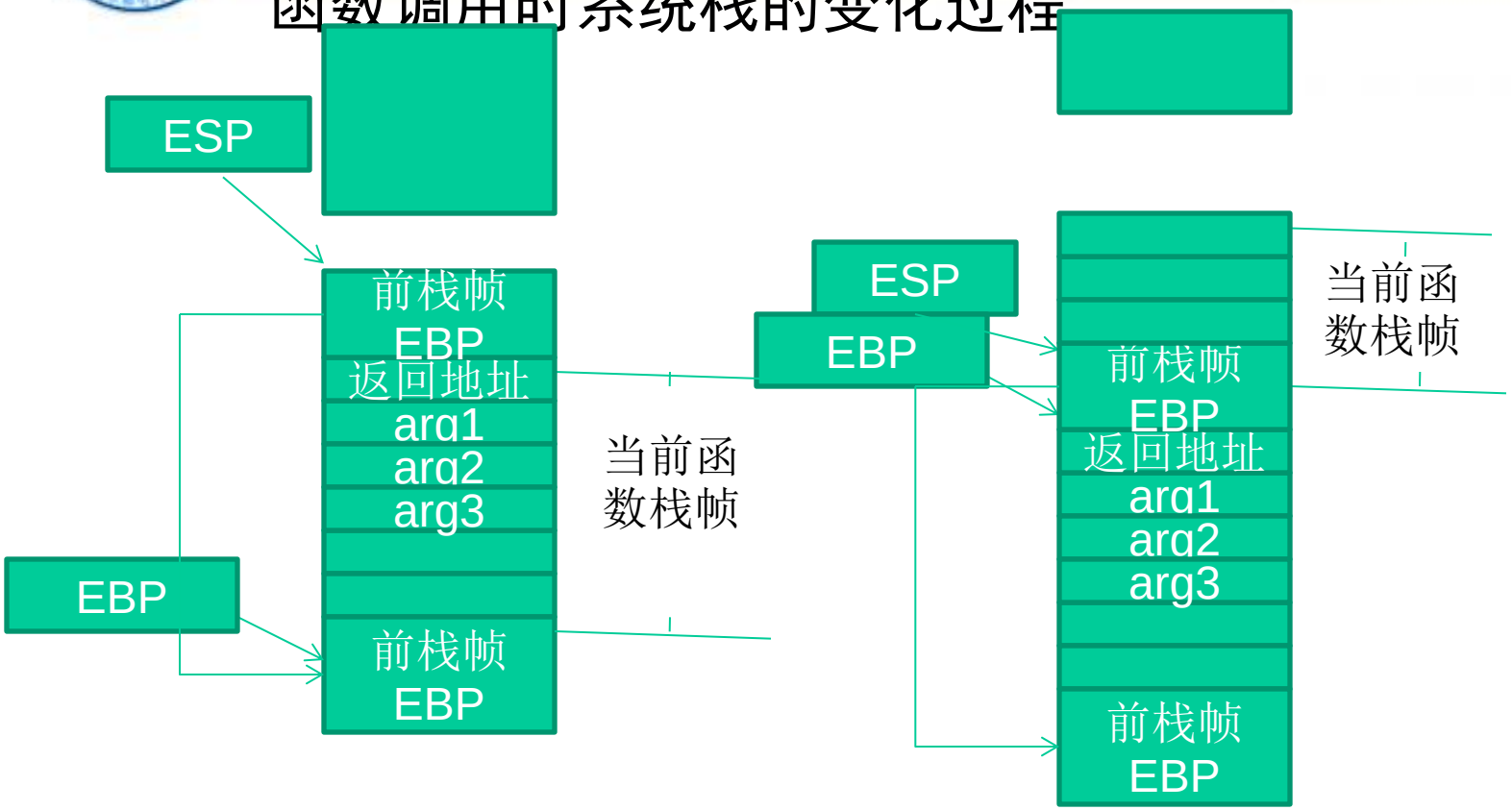
CALL指令引起的压栈操作。
返回地址入栈

PUSH EBP
保存当前栈帧底部位置，以备栈帧恢复时用

__stdcall调用约定的指令序列

二、系统栈的工作原理—函数调用约定与相关指令

函数调用时系统栈的变化过程



MOV EBP,ESP

设置新栈帧的底部开始栈帧切换

SUB ESP,XX

设置新栈帧的顶部，新栈帧切换完毕

__stdcall调用约定的指令序列

二、系统栈的工作原理—函数调用约定与相关指令

类似的，函数返回的步骤如下。

（1）保存返回值：通常将函数的返回值保存在寄存器EAX中。

（2）弹出当前栈帧，恢复上一个栈帧，具体包括：

在堆栈平衡的基础上，给ESP加上栈帧的大小，降低栈顶，回收当前栈帧的空间。

将当前栈帧底部保存的前栈帧EBP值弹入EBP寄存器，恢复上一个栈帧。

将函数返回地址弹给EIP寄存器。

（3）跳转：按照函数返回地址跳回母函数中继续执行。

二、系统栈的工作原理—函数调用约定与相关指令

以C语言和Win32平台为例，函数返回时的相关的指令序列如下。

指令	备注
add esp,xxx	降低栈顶，回收当前的栈帧
pop ebp	将上一个栈帧底部位置恢复到ebp
retn	这条指令有两个功能: a)弹出当前栈顶元素，即弹出栈帧中的返回地址。至此，栈帧恢复工作完成。 b)让处理器跳转到弹出的返回地址，恢复调用前的代码区

三、修改邻接变量

- 修改邻接变量所用程序源代码
- 修改邻接变量的原理
- 突破密码验证程序

三、堆栈溢出实例分析：修改邻接变量

通过之前的学习，我们已经知道了函数调用的细节和栈中数据的分布情况。

如果这些局部变量中有数组之类的缓冲区，并且程序中存在数组越界的缺陷，那么越界的数组元素就有可能破坏栈中相邻变量的值，甚至破坏栈帧中所保存的EBP值、返回地址等重要数据。

下面我们用一个简单的例子来说明破坏栈内局部变量对程序的安全性有何影响。

三、修改邻接变量—源代码part1

```
#include<stdio.h>
#define PASSWORD “1234567”
int verify_password(char* password)
{
   int authenticated;
   char buffer[8];
   authenticated = strcmp(password,PASSWORD);
   strcpy(buffer,password);//这里会发生溢出
   return authenticated;
}
```

三、修改邻接变量—源代码part2

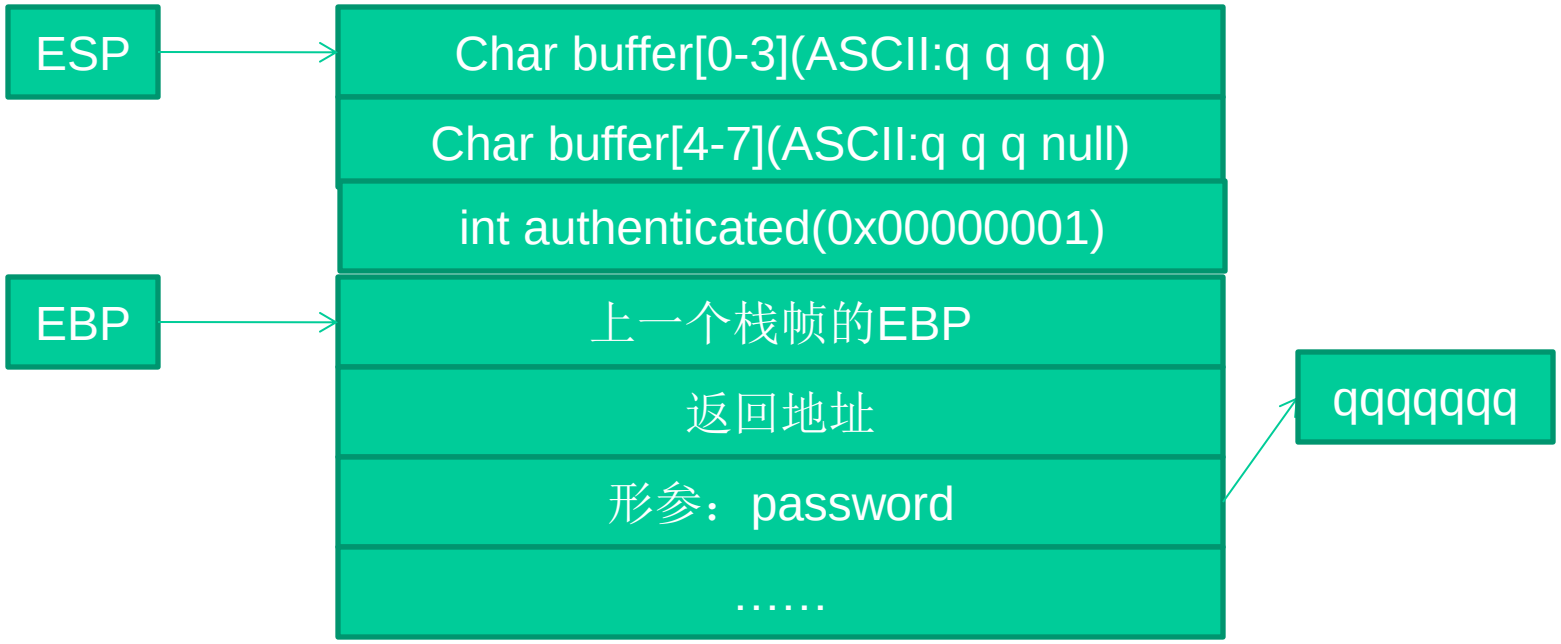
```
void main()
{
   int valid_flag=0;
   char password[1024];
   while(1)
   {
            printf(“please input password:”);
            scanf(“%s”,password);
            valid_flag = verify_password(password);
```

三、修改邻接变量—源代码part3

```
        if(valid_flag)
        {
                printf("incorrect password!\n");
        }
        else
        {
                printf("Congratulations!You have passed the
                        verification!\n");
                break;
        }
    }
}
```

三、修改邻接变量—修改邻接变量的原理

我们结合之前所学来看看栈帧的布局应该是什么样的



三、修改邻接变量—修改邻接变量的原理

authenticated为int类型，在内存中是一个DWORD，占4个字节。所以，如果让buffer数组越界，buffer[8]、 buffer[9]、 buffer[10]、 buffer[11]将写入相邻的变量authenticated中。

观察一下源代码不难发现，authenticated变量的值来源于strcmp函数的返回值，之后会返回给main函数作为密码验证成功与否的标志变量；当authenticated为0时，标识验证成功；反之则不成功。

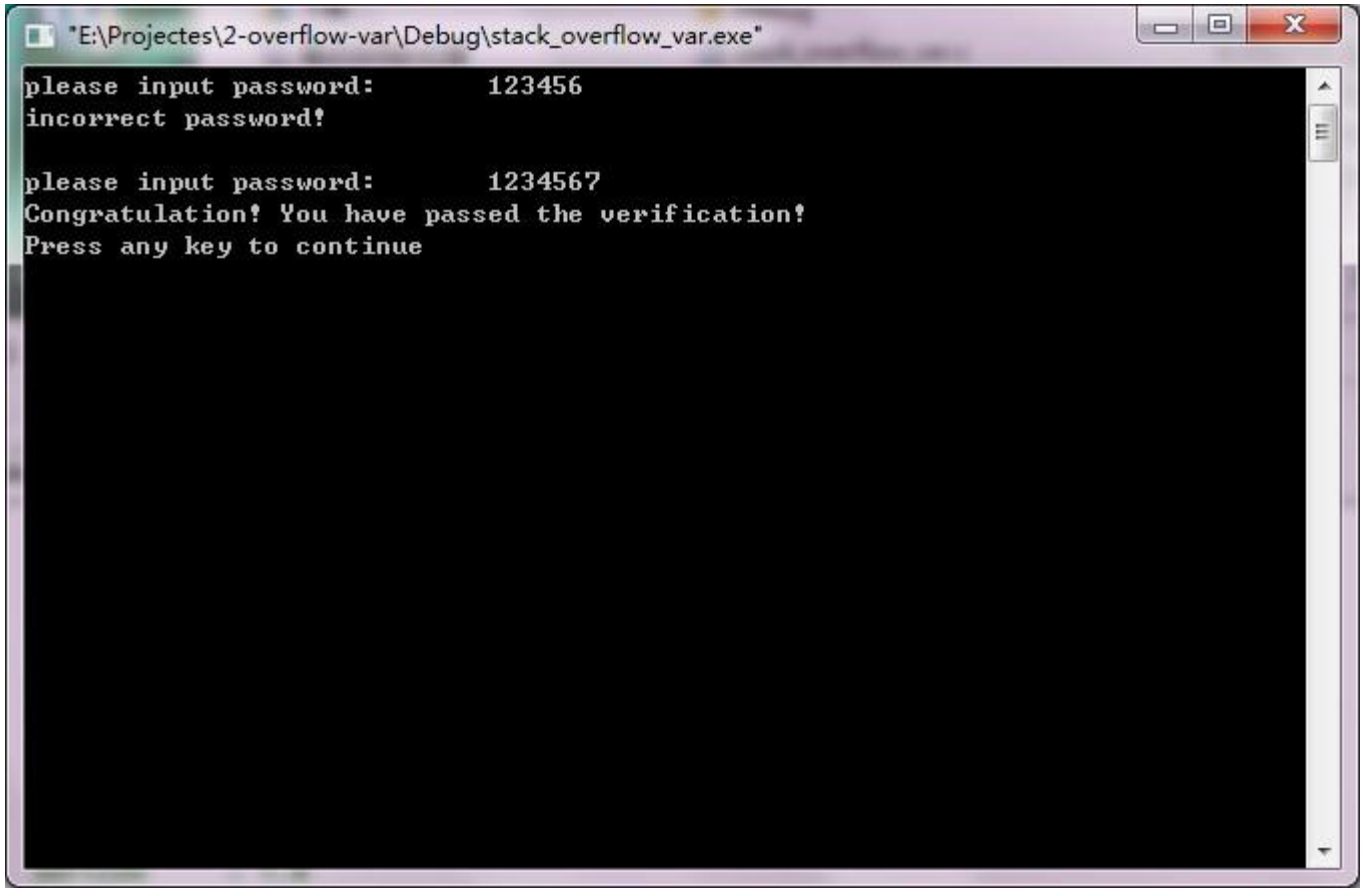
三、修改邻接变量—修改邻接变量的原理

如果我们输入的密码超过了7个字符（注意：字符串截断符NULL将占用一个字节），则越界字符的ASCII码会修改掉authenticated的值。如果这段溢出数据恰好把authenticated改为0，则程序流程将被改变。本节实验要做的就是研究怎样用非法的超长密码去修改buffer的邻接变量authenticated从而绕过密码验证程序这样有趣的事情。

三、修改邻接变量—突破密码验证程序

	推荐使用的环境	备注
操作系统	Xp sp2，win7	其他win32系统也可以进行本实验
编译器	Vc 6.0	如用其他编译器，需重新调试
编译选项	默认编译选项	Visual Studio 新版本中的GS编译选项会使栈溢出实验失败
Build版本	Debug版本	如使用release版本，则需要重新调试

三、修改邻接变量—突破密码验证程序



程序正常运行时的情况

三、修改邻接变量—突破密码验证程序

- 假如我们输入密码为7个英文字母”q”，按照字符串的关系”qqqqqqq”>”1234567”,strcmp应该返回1，即authenticated为1。

- 我们用OllyDbg动态调试的内存情况如下图所示（见下页）

程序正常运行时的情况

三、修改邻接变量—突破密码验证程序

```
0040101A     CC           INT3
0040101B     CC           INT3
0040101C     CC           INT3
0040101D     CC           INT3
0040101E     CC           INT3
0040101F     CC           INT3
00401020  /$ 55           PUSH EBP                                stack_overflow_var.verify_password(void)
00401021  |. 8BEC         MOV EBP,ESP
00401023  |. 83EC 4C      SUB ESP,4C
00401026  |. 53           PUSH EBX
00401027  |. 56           PUSH ESI
00401028  |. 57           PUSH EDI
00401029  |. 8D7D B4       LEA EDI,[LOCAL.19]
0040102C  |. B9 13000000  MOV ECX,13
00401031  |. B8 CCCCCCCC  MOV EAX,CCCCCCCC
00401036  |. F3:AB        REP STOS DWORD PTR ES:[EDI]
00401038  |. 68 1C504200  PUSH OFFSET 0042501C                    ASCII "1234567"
0040103D  |. 8B45 08      MOV EAX,DWORD PTR SS:[ARG.1]
00401040  |. 50           PUSH EAX
00401041  |. E8 0A020000  CALL strcmp                             Cstrcmp
00401046  |. 83C4 08      ADD ESP,8
00401049  |. 8945 FC      MOV DWORD PTR SS:[LOCAL.1],EAX
0040104C  |. 8B4D 08      MOV ECX,DWORD PTR SS:[ARG.1]
0040104F  |. 51           PUSH ECX
00401050  |. 8D55 F4      LEA EDX,[LOCAL.3]
00401053  |. 52           PUSH EDX
00401054  |. E8 07010000  CALL strcpy                             Cstrcpy
00401059  |. 83C4 08      ADD ESP,8
0040105C  |. 8B45 FC      MOV EAX,DWORD PTR SS:[LOCAL.1]
0040105F  |. 5F           POP EDI
00401060  |. 5E           POP ESI
00401061  |. 5B           POP EBX
00401062  |. 83C4 4C      ADD ESP,4C
00401065  |. 3BEC         CMP EBP,ESP
00401067  |. E8 74020000  CALL __chkesp
0040106C  |. 8BE5         MOV ESP,EBP
0040106E  |. 5D           POP EBP
0040106F  \. C3           RETN
00401070     CC           INT3
00401071     CC           INT3
00401072     CC           INT3
00401073     CC           INT3
00401074     CC           INT3
00401075     CC           INT3
00401076     CC           INT3
00401077     CC           INT3
00401078     CC           INT3
00401079     CC           INT3
0040107A     CC           INT3
0040107B     CC           INT3
0040107C     CC           INT3
0040107D     CC           INT3
0040107E     CC           INT3
0040107F     CC           INT3
```

在strcpy设置断点（F2），运行strcpy之后观察缓冲区中的结果

三、修改邻接变量—突破密码验证程序

```
0012FA8C  0012FAE0  ∝·▲.  ASCII "qqqqqqq"
0012FA90  0012FB44  D♂▲.  ASCII "qqqqqqq"
0012FA94  0012FF48  H ▲.
0012FA98  00000000  ....
0012FA9C  7FFDE000  .∝²⌂
0012FAA0  CCCCCCCC  ╠╠╠╠
0012FAA4  CCCCCCCC  ╠╠╠╠
0012FAA8  CCCCCCCC  ╠╠╠╠
0012FAAC  CCCCCCCC  ╠╠╠╠
0012FAB0  CCCCCCCC  ╠╠╠╠
0012FAB4  CCCCCCCC  ╠╠╠╠
0012FAB8  CCCCCCCC  ╠╠╠╠
0012FABC  CCCCCCCC  ╠╠╠╠
0012FAC0  CCCCCCCC  ╠╠╠╠
0012FAC4  CCCCCCCC  ╠╠╠╠
0012FAC8  CCCCCCCC  ╠╠╠╠
0012FACC  CCCCCCCC  ╠╠╠╠
0012FAD0  CCCCCCCC  ╠╠╠╠
0012FAD4  CCCCCCCC  ╠╠╠╠
0012FAD8  CCCCCCCC  ╠╠╠╠
0012FADC  CCCCCCCC  ╠╠╠╠
0012FAE0  71717171  qqqq
0012FAE4  00717171  qqq.
0012FAE8  00000001  ☺...
0012FAEC  0012FF48  H ▲.
0012FAF0  004010EB  δ►@.  RETURN from stack_overflow_var.verify_password to stack_overflow_var.main+5B
0012FAF4  0012FB44  D♂▲.  ASCII "qqqqqqq"
0012FAF8  00000000  ....
0012FAFC  00000000  ....
0012FB00  7FFDE000  .∝²⌂
0012FB04  CCCCCCCC  ╠╠╠╠
0012FB08  CCCCCCCC  ╠╠╠╠
0012FB0C  CCCCCCCC  ╠╠╠╠
0012FB10  CCCCCCCC  ╠╠╠╠
0012FB14  CCCCCCCC  ╠╠╠╠
0012FB18  CCCCCCCC  ╠╠╠╠
0012FB1C  CCCCCCCC  ╠╠╠╠
0012FB20  CCCCCCCC  ╠╠╠╠
0012FB24  CCCCCCCC  ╠╠╠╠
```

我们注意到有 qqqqqqq+null这个字符串，而比较后的结果为01就在它地址的后面

三、修改邻接变量—突破密码验证程序

局部变量名	内存地址	偏移3处的值	偏移2处的值	偏移1处的值	偏移0处的值
Buffer[0-3]	0x0012FAE0	q	q	q	q
Buffer[4-7]	0x0012FAE4	NULL	q	q	q
authenticated	0x0012FAE8	0x00	0x00	0x00	0x01

栈帧数据分布的情况

在观察内存的时候应当注意“内存数据”与“数值数据”的区别。在我们的调试环境中，内存由低到高分布，你可以简单地把这种情形理解成Win32系统在内存中由地位向高位存储一个4字节的双字，但在作为“数值”应用的时候，确实按照由高位字节向低位字节进行解释。这样一来，在我们的调试环境中，“内存数据”中的DWORD和我们逻辑上使用的“数值数据”是字节序逆序过的。

三、修改邻接变量—突破密码验证程序

- 我们可以联想到，如果输入超过7个，算上NULL会超过8个字节，会发生什么事情呢？

- 我们不妨来试试，输入　qqqqqqqqrst　之后和之前一样，观察调试器的数据，会发现如下的数据。

三、修改邻接变量—突破密码验证程序

```
0012FACC  CCCCCCCC  ╠╠╠╠
0012FAD0  CCCCCCCC  ╠╠╠╠
0012FAD4  CCCCCCCC  ╠╠╠╠
0012FAD8  CCCCCCCC  ╠╠╠╠
0012FADC  CCCCCCCC  ╠╠╠╠
0012FAE0  71717171  qqqq
0012FAE4  71717171  qqqq
0012FAE8  00747372  rst.
0012FAEC  0012FF48  H ↕.
0012FAF0  004010EB  §►@.  RETURN from stack_overflow_var.verify_password to stack_overflow_var.main+5B
0012FAF4  0012FB44  D▼↕.  ASCII "qqqqqqqqqrst"
0012FAF8  00000000  ....
0012FAFC  00000000  ....
0012FB00  7FFDE000  .α²⌂
0012FB04  CCCCCCCC  ╠╠╠╠
0012FB08  CCCCCCCC  ╠╠╠╠
0012FB0C  CCCCCCCC  ╠╠╠╠
0012FB10  CCCCCCCC  ╠╠╠╠
0012FB14  CCCCCCCC  ╠╠╠╠
0012FB18  CCCCCCCC  ╠╠╠╠
0012FB1C  CCCCCCCC  ╠╠╠╠
0012FB20  CCCCCCCC  ╠╠╠╠
0012FB24  CCCCCCCC  ╠╠╠╠
0012FB28  CCCCCCCC  ╠╠╠╠
0012FB2C  CCCCCCCC  ╠╠╠╠
0012FB30  CCCCCCCC  ╠╠╠╠
0012FB34  CCCCCCCC  ╠╠╠╠
0012FB38  CCCCCCCC  ╠╠╠╠
0012FB3C  CCCCCCCC  ╠╠╠╠
0012FB40  CCCCCCCC  ╠╠╠╠
0012FB44  71717171  qqqq
0012FB48  71717171  qqqq
0012FB4C  00747372  rst.
0012FB50  CCCCCCCC  ╠╠╠╠
0012FB54  CCCCCCCC  ╠╠╠╠
0012FB58  CCCCCCCC  ╠╠╠╠
0012FB5C  CCCCCCCC  ╠╠╠╠
0012FB60  CCCCCCCC  ╠╠╠╠
0012FB64  CCCCCCCC  ╠╠╠╠
0012FB68  CCCCCCCC  ╠╠╠╠
```

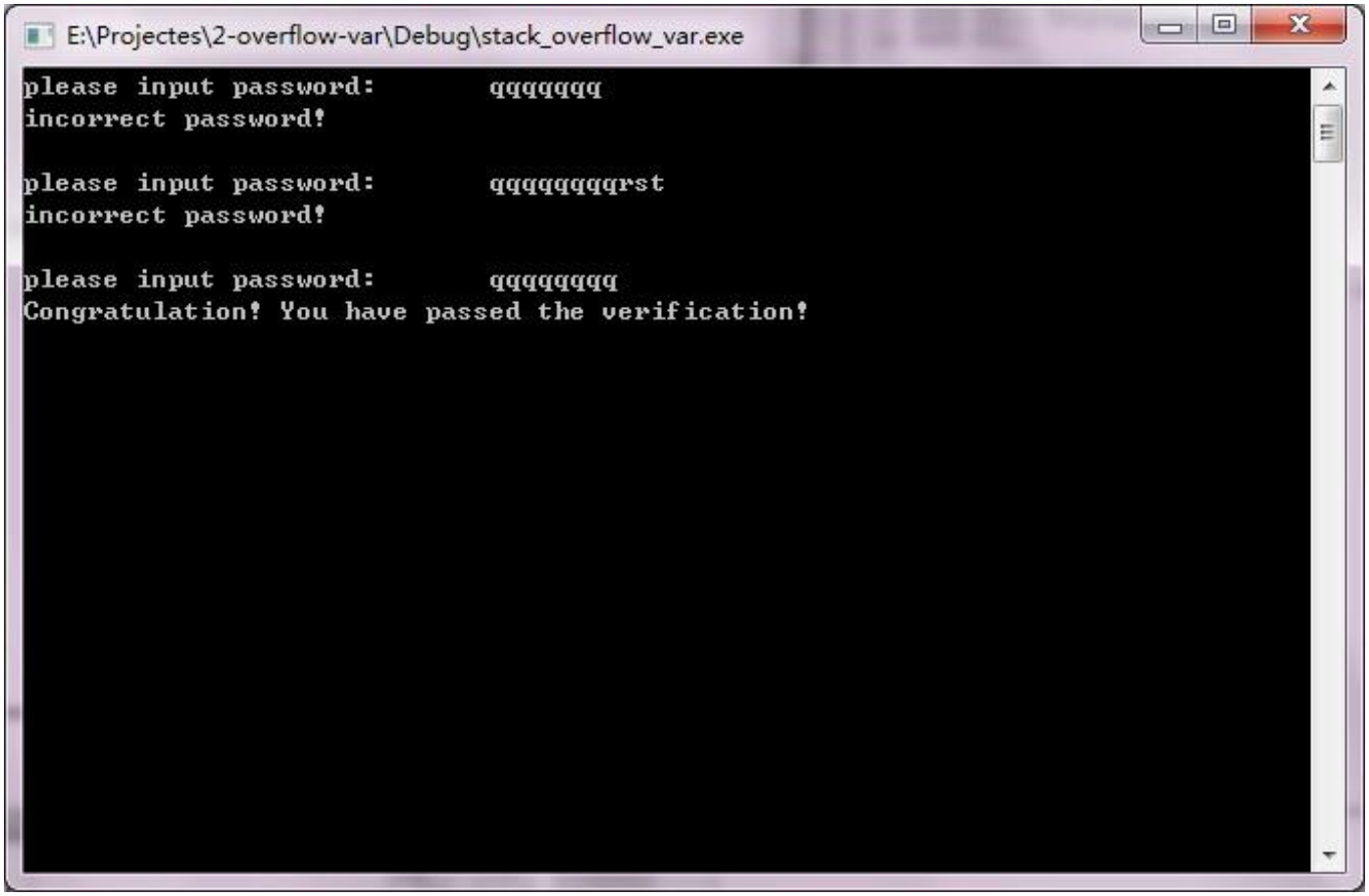
在strcpy设置断点（F2），运行strcpy之后观察缓冲区中的结果，我们发现authenticated变成了0x00747372。

三、修改邻接变量—突破密码验证程序

- 那么我们经过分析可以发现，如果我们输入的是8个q，那么他的最后以为NULL会淹没authenticated的数据

- Authenticated会变成0x00000000，那么判断分支则会走向正确的方向，就会绕过密码验证程序。

三、修改邻接变量—突破密码验证程序



果然当我们输入8个q的时候程序发生了程序员预料之外的结果，我们这个演示也成功运行得到了想要的结果。

三、修改邻接变量—突破密码验证程序

- 当大家做了这个实验后，发现并不是所有的8位字符串都能得到想要的结果，这是为什么呢？

  - 由于authenticated的值来源于字符串比较函数strcmp的返回值。按照字符串的序关系，当输入字符串大于“1234567”时，会返回1，内存中的值为0x00000001，可以淹没地位突破验证；那么当输入字符串小于“1234567”时，会返回-1，而变量在内存中的值按照双字-1的补码存放，为0xFFFFFFFF，低位被淹没后便成了0xFFFFFF00，这是的值是不能冲破验证程序的。如“01234567”就不可以。

栈溢出原理

栈溢出我们仅仅举了一个简单的修改邻接变量的例子，其实它能做的远不止这些，它还可以修改函数的返回地址，控制程序的执行流程，甚至能够进行代码的植入，希望有兴趣的同学能够结合自己所学进行深入的研究。

讲义及实验说明下载

- http://mail.126.com
- 用户名：　　softsecurity
- 口令：　　123456
- 6-7次实验，地点：主楼910、919房间，双周，周二晚上（6：30-9：00）
- 作业提交方式：电子版，上传到网盘，在下一次上课前提交
- 网盘地址：http://k.115.com/330319852/softsecurity

谢谢观赏！
Thanks！