4 软件的漏洞与防护

2019年4月10日 23:35

1 软件缺陷与漏洞机理概述

1.1 安全漏洞为什么存在，不能消除?

- 由于计算能力的迅速增长，人们对编程的技巧和高雅的代码关注的愈来愈少，而是把更多的精力集中在尽可能快和尽可能低成本地写出功能代码
- 重功能性，轻完备性和安全性
- 程序只能严格按照规则做编程要他做的事情。不幸的是，所编写的程序并不总是与程序员预计让程序完成的事情一致
- 计算机只能够理解机器语言
- 用高级语言（如C语言）的程序员并不总是了解整个程序执行情况
- 安全漏洞的本质：代码流程的变化
    - 任何安全漏洞都是导致程序走到别的流程上去了，而非设计流程
- 改变程序执行流程的黑客入侵实际上仍然没有违反任何编程规则，它仅仅是知道了规则的更多情况，而且以难以预料的方法使用规则
- 安全编程最主要的就是如何避免安全漏洞!

为什么漏洞难以消除?

- 根源：矛盾不可调和的产物
    - 存在的偶然性:设计、实现、配置的失误,故意埋设
    - 价值归属:军事,经济价值,时空特性

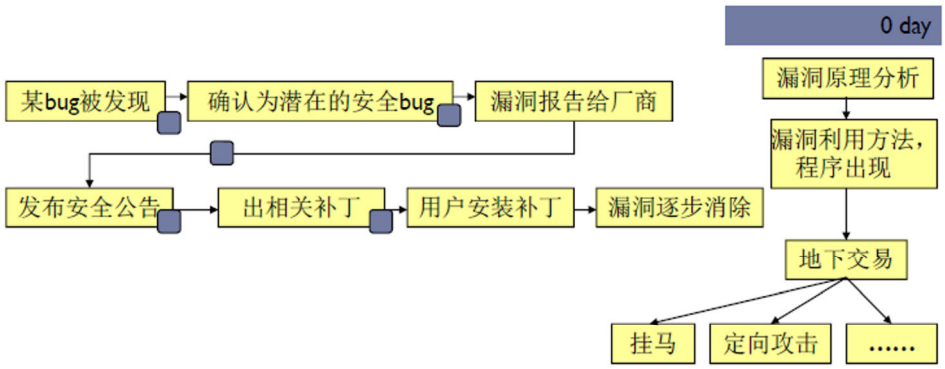
    - 漏洞存在的必然性:政治、军事、经济矛盾不可调和的产物
- 挑战：技术困难重重
    - 计算能力困境—软件规模的持续增长与有限计算能力之间的矛盾
    - 检测能力困境—现有技术不能较好地满足漏洞检测需求之间的矛盾
    - 分析能力困境—现有的自动化水平并不能与人们对漏洞检测需求相适应的矛盾

1.3 安全漏洞对系统的威胁

- 非法获取访问权限
    - 访问控制：防止未经授权使用资源，包括防止以非授权方式使用资源。
    - 访问权限：访问控制的访问规则，用来区别不同访问者对资源的访问权限。
- 权限提升
    - 攻击者通过攻击某些有缺陷的系统程序，把当前较低的账户权限提升到更高级别的用户权限。
    - 由于管理员权限较大，通常将获得管理员权限看做是一种特殊的权限提升。
- 拒绝服务
    - 拒绝服务攻击的目的是使计算机软件或系统无法正常工作、无法提供正常的服务。
    - 根据存在漏洞的应用程序的应用场景，可划分为本地拒绝服务漏洞和远程拒绝服务漏洞
    - 与网络层面的Dos相比： 侧重于软件或系统组件漏洞引发的拒绝服务攻击
- 恶意软件植入
    - 当恶意软件明确攻击目标后，需要通过特定方式将攻击代码植入到目标中。
    - 主动植入：由程序自身利用系统地正常功能或缺陷漏洞将攻击代码植入到目标中，而不需要人的任何干预
    - 被动植入：恶意软件将攻击代码植入到目标主机时需要借助用户的操作。（通常和社会工程学的攻击方法相结合，诱使用户触发漏洞）
- 数据丢失或泄露
    - 第一类漏洞是由于对文件的访问权限设置错误而导致文件被非法读取
    - 第二类漏洞常见于Web应用程序，由于没有充分验证用户的输入，导致文件被非法读取
    - 第三类漏洞主要是系统漏洞，导致服务器信息泄露。

1.4 安全漏洞的分类

从时序上看漏洞分类

- 

1.6 漏洞利用

- 漏洞利用是黑客针对已有的漏洞，根据漏洞的类型和特点而采取相应的技术方案，进行尝试性或实质性的攻击。

- 这类攻击的形式比较多样化：
  - 一个简单命令
  - 一个具体操作
  - 下载病毒木马等恶意软件
- 共同点：通过触发漏洞来隐蔽地执行恶意操作
- 触发漏洞来完成恶意操作的程序通常被称为 exploit

不同视角看漏洞利用

- 一个防火墙隔离（只允许运维部的人访问）的网络里运行一台 Windows服务器，配置如下：
  - 操作系统中只有 administration和 iis_user低权限用户。
  - 系统中运行了 IIS服务（ iis_user权限）、 Oracle服务、 Symantec病毒防火墙。
- 一个攻击者的目的是修改 Oracle数据库中的账单表中的数据。 已知：
  - IIS在接收用户页面 GET请求时 user字段大于 64字节时会发生栈溢出。
  - Symantec软件的驱动程序检测新建文件时，文件名超过 128字节时存在栈溢出。
  - Administrator用户以 sysdba角色（ oracle中的最高权限）登录数据库不需要口令。
- 问题：给出可能的攻击步骤，说明各步骤漏洞和手法属于哪些分类。
- 可能的攻击步骤为：
  a. 接入运维部的网络，获得一个运维部的 IP地址从而能通过防火墙访问被保护的服务器。
  b. 利用 IIS服务的远程缓冲区溢出漏洞，直接获得一个低权限 iis_user身份的 cmd命令行窗口。
  c. 利用 Symantec杀毒软件溢出漏洞获得 administrator权限 cmd窗口。
  d. 利用 oracle的 sysdba角色登录进入数据库（本地 administrator用户登录不需要密码）。
  e. 修改目标表的数据。
- 步骤
  a. 认证绕过
  b. 远程漏洞、代码执行（机器码）、认证绕过
  c. 权限提升、认证绕过
  d. 认证绕过、 oracle设计上一个问题
  e. 数据写

1.7 总结

- 目标

  保护软件使用者的数据安全，业务持续。

  实现安全漏洞较少的软件。
- 漏洞

  正常软件安全问题的技术方面根源。

  和安全相关的可利用的软件BUG。
- 视角

  设计、实现中的。

  典型漏洞：溢出、SQL注入、跨站等。

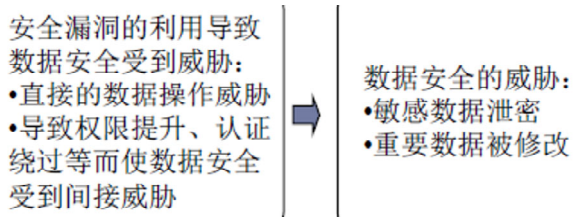
  本地、远程、被动、主动、

  高、中、低。

  老、新、Oday。

数据操作：读、写、执行。

源：内存、文件、I0等。

- 

2 典型软件漏洞机理分析

2.1 缓冲区溢出

- 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

shellcode 写开一个shell的Exploit的方法

- 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

内存

- 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
  - 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
  - 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
- 在操作系统中，系统都会给每个进程分配独立的虚拟地址空间，在真正调用时则将其映射到物理内存空间

相关的几个术语

- 缓冲区 （ buffer）
  - 内存空间中用于存储程序运行时临时数据的一片大小有限且连续的内存区域

    char buffer[256];

buffer=new int[64];

- 溢出（ overflow）
  - 数据过长导致无法存储在预期区域内，覆盖了存储其他数据的区域
    strcpy(smallBuffer,bigString)
- 缓冲区溢出
  - 当计算机向缓冲区内填充数据时超过了缓冲区本身的容量，溢出的数据覆盖到了合法数据上。
- 栈缓冲区溢出(stack overflow)
  - 溢出发生在栈区中
  - 覆盖堆栈结构
- 堆缓冲区溢出(heap overflow)
  - 溢出发生在堆区中
  - 覆盖指定的四字节指针
- 格式串问题
  - 覆盖指定的若干指针
- 非安全函数
  - 没有内嵌边界保护支持，必须要使用额外代码进行边界检查的函数
    strcat(),strcpy(),sprintf(),vsprintf(),bcopy(),gets(),scanf(),…
- 安全函数
  - 可以限制所操作的数据长度，正确使用则不会导致缓冲区问题的函数
    strncpy(),memcpy(),snprintf(),strncat(),…
    strncpy(DstBufferm,SrcBuffer,sizeof(DstBuffer)-1);
- 边界检查（Boundary Check）
  - 在向缓冲区中存储数据时，确定数据长度是否会超出缓冲区边界
    if(strlen(SrcBuffer)<sizeof(DstBuffer))
    strcpy(DstBuffer,SrcBuffer);
    else
    printf（“DstBuffer is too (“ small\n”););
- 栈不可执行
  - 某些型号的用CPU支持对内存是否可执行的标志位，操作系统可以利用该特性在进程初始化时将堆栈设置为不可执行

缓冲区溢出漏洞存在的原因和后果

- 原因
  - 没有使用安全函数，也没有进行边界检查
  - 没有正确地使用安全函数
    strcpy(DstBufferm,SrcBuffer,sizeof(SrcBuffer)-1)
  - 设计和计算失误用
    MultiByteToWideChar()
- 后果
  - 攻击者使远程服务程序或者本地程序崩溃
  - 执行本地任意程序
  - 主机被控制

为什么 STRCPY等是不安全函数

- 没有检查操作的数据长度，仅凭某些特征判断是否应结束操作，譬如“\0”
- strcpy()的实现

```
char *strcpy(char *dst,const char * src)
{
    char *cp=dst;
    char *cp=dst;
    while(*cp++=*src++);
    return(dst);
}
```

导致缓冲区溢出的几种方式

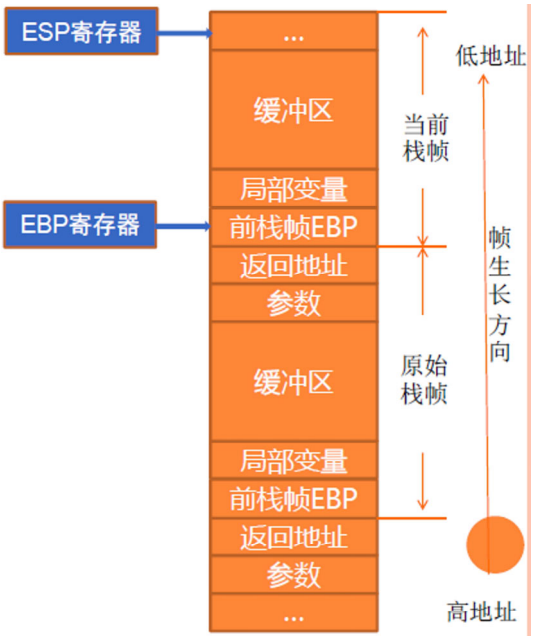
- 错误的双字节字符串长度比较
- 用源字符串长度做拷贝限制
- strncpy等字符串截断拷贝函数的陷阱
  - 如果源字符串长度大于限定长度参数，strncpy会按照限定长度拷贝到目标，但并不会在最后加上字符串结束符’\0’
  - 这个陷阱可能导致不易察觉的缓冲区溢出
- 整形变量处理不当（整形溢出）

栈

- 在程序设计中，栈通常指的是一种先进后出（first in last out, FILO）的数据结构
- 入栈（PUSH）和出栈（POP）是进行栈操作的两种常见方法。
- 为了标识栈的空间大小，同时为了更方便地访问栈中数据，栈包括栈顶（TOP）和栈底（BASE）两个栈指针
- 栈顶随入栈和出栈操作而动态变化，但始终指向栈中最后入栈的数据；栈底指向先入栈的数据
- 栈顶和栈底之间的空间存储的就是当前栈中的数据。

系统栈

- 相对于广义的栈而言，系统栈则是操作系统在每个进程的虚拟内存空间中为每个线程划分出来的一片存储空间。
- 遵守先进后出的栈操作原则。
- 由系统自动维护，用于实现高级语言中的函数的调用
- 对于类似用C语言这样的高级语言，系统栈的PUSH和POP等堆栈平衡的细节对用户是透明的
- 栈的生长方向是从**高地址向低地址**增长的
- 操作系统为进程中的每个函数调用都划分了一个栈帧空间
- 每个栈帧都是一个独立的栈结构
- 系统栈是这些函数调用栈帧的集合

- 

系统栈在工作的过程中主要用到了三个寄存器

- ESP：栈指针寄存器（extended stack pointer），其存放的是当前栈帧的栈顶指针
- EBP：基址指针寄存器（extended base pointer），其存放的是当前栈帧的栈底指针
- EIP：指令寄存器（extended instruction pointer），其存放的是下一条等待执行的指令地址。如果控制了EIP寄存器的内容，就可以控制进程行为——通过设置EIP的内容，使CPU去执行我们想要执行的命令，从而劫持了进程。

函数调用约定上的差异

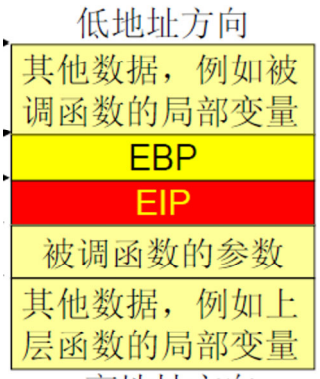
- 参数入栈的顺序：从左向右、从右向左
- 函数返回时恢复堆栈的操作：由子函数完成、由母函数完成
-

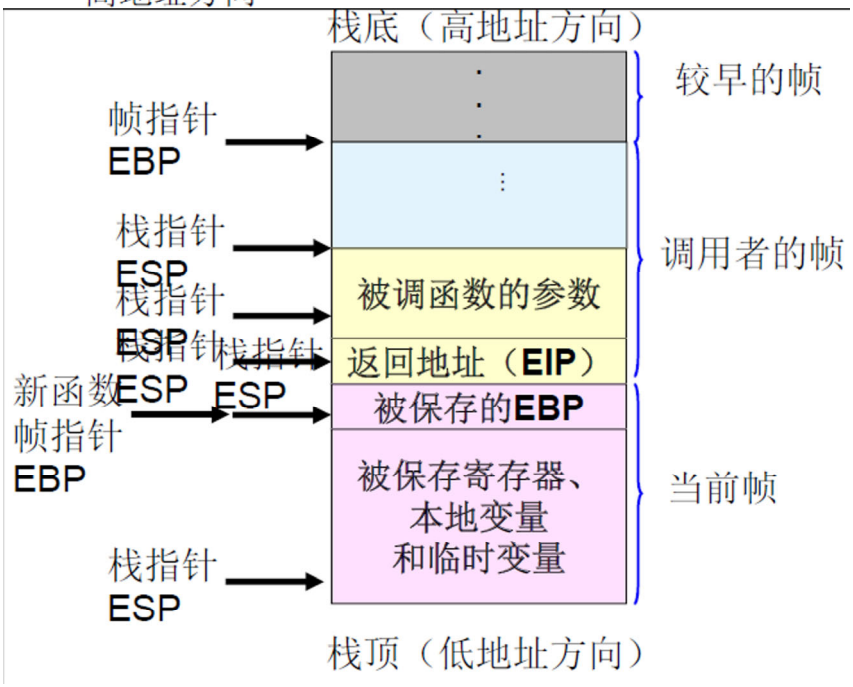
关键字	清理堆栈	参数入栈顺序
__cdecl	调用函数	右 → 左
__stdcall	被调用函数	右 → 左
__fastcall	被调用函数	右 → 左

- 对于用windows平台下的用VC++而言，一般按照用stdcall方式对函数调用

函数调用步骤

1. 参数入栈：将被调用函数的参数按照从右向左的顺序依次入栈
2. 返回地址入栈：将用call指令的下一条指令的地址入栈
3. 代码区跳转：处理器从代码区的当前位置调到被调函数的入口处
4. 栈帧调整：主要包括保存当前栈帧状态、切换栈帧和给新栈帧分配空间

- 

- 

函数调用代码

```
push arg1        #参数入栈
push arg2
call 函数地址 #返回地址入栈，跳转到函数入口处
#下面三条指令实现栈帧调整
push ebp         #保存当前栈帧的栈底
mov ebp,esp   #设置新栈帧的栈底，实现栈帧切换
sub esp,xxx      #抬高栈顶，为函数的局部变量等开辟 栈空间
```

函数返回步骤

1. 根据需要保存函数返回值到eax寄存器中（一般使用eax寄存器存储返回值）
2. 降低栈顶，回收当前栈空间
3. 恢复母函数栈帧
4. 按照函数返回地址跳转回到父函数，继续执行

函数返回代码

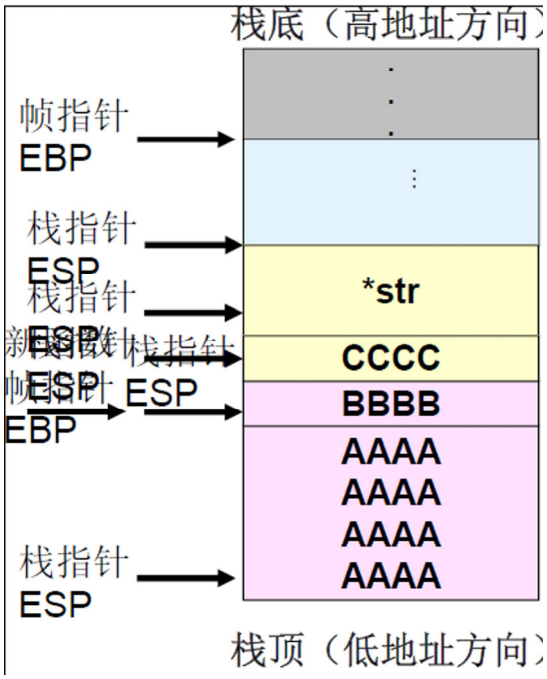
```
add esp ,xxx  #降低栈顶，回收当前的栈帧空间（堆栈平衡）
pop ebp       #还原原来的栈底指针，恢复母函数栈帧
```

retn　　#弹出栈帧中的返回地址，让CPU跳转到返回地址

栈溢出

一个典型的问题程序

```
void vulfunc(char *str){
    char buffer[16];
    strcpy(buffer,str);
}
int main(int argc,char int main(int argc,char **argv){
    if(argc>1)
        vulfunc(argv[1]);
    return 0;
}
```

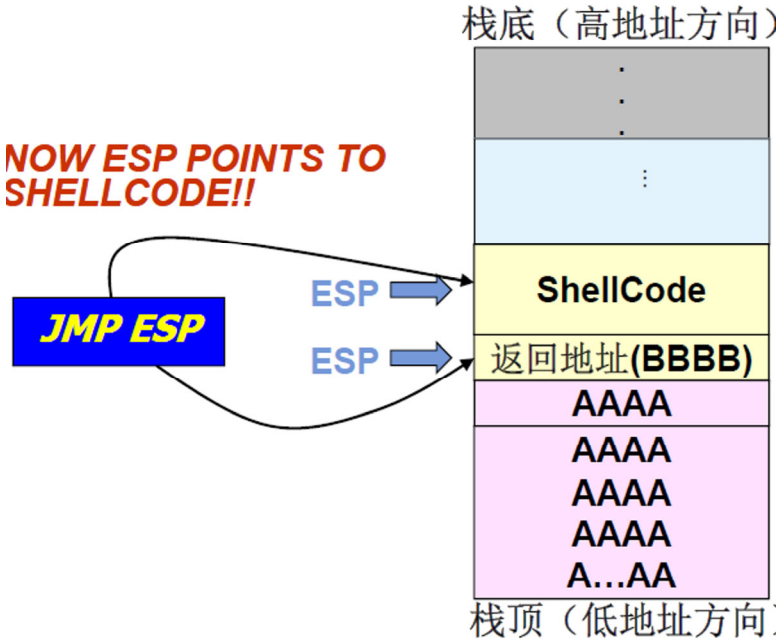
- 

修改函数返回地址

- 如返回地址被改写，函数返回时，流程会转入到被改写的地址

用什么数据覆盖返回地址?

- 将内存中的shellcode地址赋给返回地址，然后使程序直接跳转到shellcode处执行。(不可行)
- 在实际的漏洞利用过程中，由于动态链接库的装入和卸载等原因，Windows进程的函数栈帧可能发生移位，即shellcode在内存中的地址是动态变化的。所以采用这种直接赋地址值的简单方式在以后的运行过程中会出现跳转异常。
- 处于栈中shellcode开始位置的高位通常是0x00，如果用该地址覆盖返回地址，则构造的溢出字符串中0x00之后的数据可能在进行字符串操作时被截断。



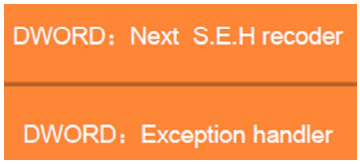
修改函数返回地址

- 用什么数据覆盖返回地址?
  - 用jmp esp的地址
- 如何找到jmp esp的地址?
  - jmp esp ff e4
  - 在覆盖返回地址的时候用系统动态链接库中某条处于高地址且位置固定的跳转指令所在的地址进行覆盖。
  - 在内存中搜索jmp esp指令是比较容易的，为了稳定性和通用性，一般选择kernel32.dll或user32.dll中的地址

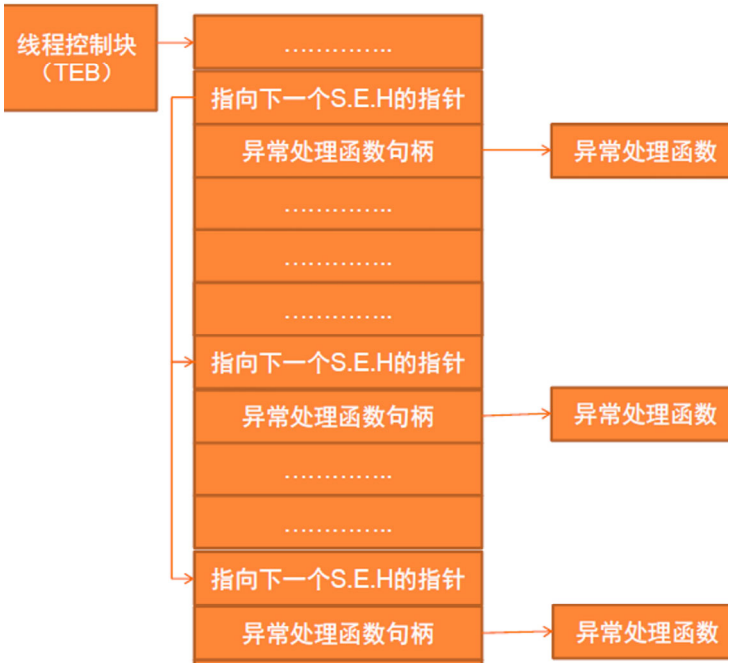
栈溢出的利用--根据被覆盖的数据位置和所要实现的目的不同分类

- 修改函数返回地址
- 修改邻接变量
- 修改S.E.H结构覆盖

Windows的异常处理机制

- 在Windows平台下，操作系统或应用程序运行时，为了保证在出现除零、非法内存访问等错误时，系统也能在正常运行而不至于崩溃或宕机，Windows会对运行在其中的程序提供一次补救机会来处理错误，这就是用Windows的异常处理机制
- S.E.H
  - Windows的异常处理机制的一个重要数据结构是位于系统栈中的异常处理结构体SEH（struct exception handler）
  - 包含两个用DWORD指针:SEH链表指针和异常处理函数句柄
  - 

- 当程序中包含异常处理块时，编译器则要生成一些特殊的代码用来实现异常处理机制。

- 编译时产生一些来支持处理用SEH数据结构的表和确保异常被处理的回调（callback）函数
- 当栈中存在多个SEH时，他们通过链表指针在栈内由栈顶向栈底串成单向链表。
- 

- 当发生异常时，操作系统会中断程序，首先从TEB的0字节偏移处取出最顶端的SEH结构地址，使用异常处理函数句柄所指向的代码来处理异常。
- 如果该异常处理函数运行失败，则顺着SEH链表依次尝试其他的异常处理函数
- 如果程序预先安装的所有异常处理函数均无法处理，系统将采用默认的异常处理函数,弹出错误对话框并强制关闭程序。

修改 SEH结构覆盖

- 修改用SEH结构覆盖就是在程序出错之后系统关闭之前，让程序去执行一个预先设定的回调函数。
- 利用缓冲区溢出覆盖SEH，将SEH中异常处理函数的入口地址更改为shellcode的起始地址或可以跳转到用shellcode的跳转指令的地址。从而导致程序发生异常时，windows异常处理机制转而执行的不是正常异常处理函数，而是已覆盖的shellcode

栈溢出检查-GS

- 什么是StackCookie
  - 在返回地址之前压入一个随机生成的StackCookie
  - 函数返回前，先检查StackCookie是否被修改
  - 始于VisualStudio2003

- 如何使用StackCookie
  - /GS选项，这是默认的
- 实现
  - 程序启动时，它是首先会计算出程序的cookie（伪随机数，4字节无符号整数）
  - 然后将cookie保存在加载模块的.data节中
  - Cookie被拷于栈中，位于返回地址、寄存器EBP之后，局部变量之前
  - 函数结尾处程序会把这个cookie和保存在data节中的cookie比较，如果相等就说明进程的系统的栈被破坏，终止程序运行
- 为了尽量减少额外代码行对性能带来的影响，VS将仔细评估程序中哪些函数需要保护
  - 一个函数包含字符串缓冲区或使用_alloca函数在栈上分配空间时
  - 缓冲区少于5字节时不保存cookie
- /GS保护机制的功能 除了在栈中加入cookie外，另外一个重要保护机制是对栈中变量进行重新排序。
- 为了防止对函数内部的局部变量和参数的攻击，编译器会进行如下操作：
  - 对函数栈帧进行重新排序，把字符串缓冲区分配在栈帧的最高地址上。因此当字符串缓冲区被溢出时，也就不能溢出任何本地局部变量了
  - 将函数参数复制到寄存器，防止参数被溢出
- GS安全机制的不足
  - 在有几个缓冲区的函数里，它们相继放在栈中，因此从一个缓冲区溢出到另一个缓冲区是有可能的
  - 结构成员的互操作性的问题不能重新排序。因此当它们包含缓冲区时，这个缓冲区将位于struct或class声明固定的位置。在缓冲区溢出发生后它们之后的字段就可以被控制。
- 绕过机制
  - 利用异常处理器绕过（GS没有保护异常处理器）
    - 缓冲区溢出覆盖异常处理器地址，在检查cookie前触发异常
  - 通过同时替换栈中和data节中的cookie来绕过
  - 通过覆盖父函数的栈数据绕过

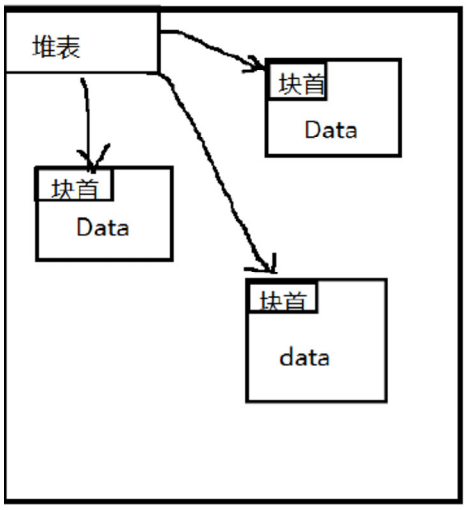
堆的特性

- 在程序运行时动态分配内存。
- 使用时需要程序员使用专有的函数进行申请
- 一般用一个堆指针来使用申请得到的内存
- 使用完毕后需要将堆指针传给堆释放函数回收这片内存，否则会造成内存泄漏

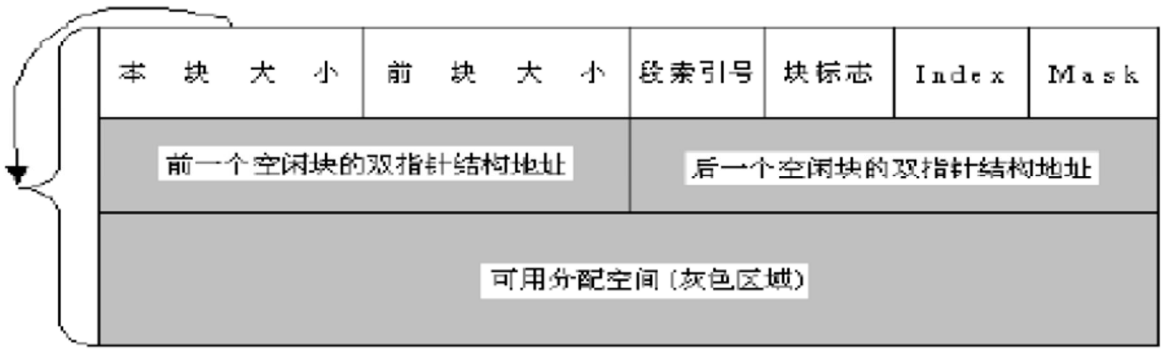
堆与栈的区别

	堆内存	栈内存
典型用例	动态增长的链表等数据结构	函数局部数组
申请方式	需要函数动态申请，通过返回的指针使用	在程序中直接声明即可
释放方式	需要专门的函数来释放，如free	系统自动回收
管理方式	由程序员负责申请与释放，系统自动合并	由系统完成
所处位置	变化范围很大	一般是0x0010xxxx
增长方向	从内存低地址向高地址排列	由内存高地址向低地址增加

堆的结构

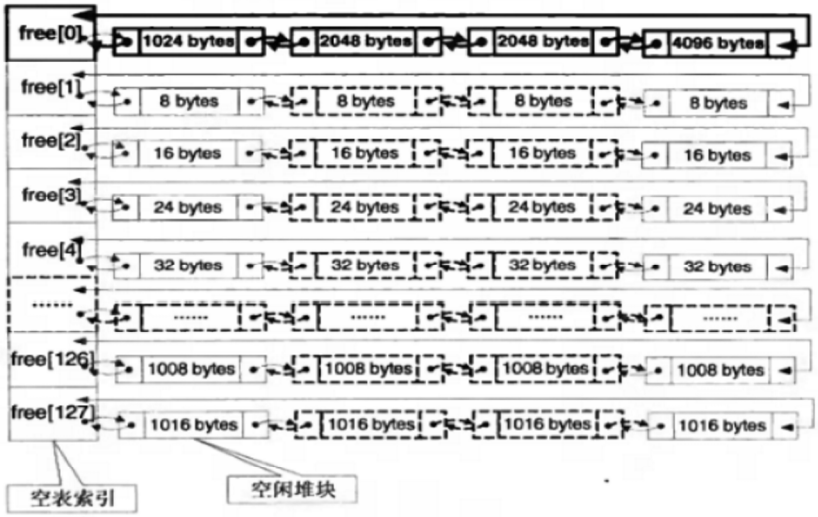


- 堆块
  - 空闲态：堆块被链入空链表中，由系统管理
  - 占有态：堆块会返回一个由程序员定义的句柄，由程序员管理



  - 空闲堆块比占有堆块多出了两个4字节的指针，这两个指针用于链接系统中的其他空闲堆块

堆表

- 

- 空表
  - 空闲堆块的块首中包含一对重要的指针，这对指针用于将空闲堆块组织成双向链表
  - 根据大小不同，空表总共被分成 128条

堆溢出利用的精髓

1. 用精心构造的数据去溢出覆盖下一个堆块的块首，使其改写块首中的flink指针和blink指针，
2. 然后在分配、释放、合并等操作发生时伺机获得一次向内存任意地址写入任意数据的机会（Arbitrary Dword Reset,又称 Dword Shoot）
3. 通过这个机会，可以控制设计的目标（任意地址），选择适当的目标数据，从而劫持进程，运行shellcode。

堆溢出

攻击目标	内容	改写后的结果
栈帧中的函数返回地址	Shellcode的起始地址	函数返回时，跳去执行shellcode
栈帧中SHE句柄	Shellcode的起始地址	异常发生时，跳去执行shellcode
重要函数调用地址	Shellcode的起始地址	函数调用时，跳去执行shellcode

2.2 整型溢出

什么是整型溢出

- int i = 0xFFFFFFFF

  i + 1 =?
- 整数的符号问题
- 整数相乘与相加的溢出
- 短整型与长整型的运算

整型溢出防范

- 不同大小类型的整数不要互相尝试存储
- 对整数的操作要格外注意
  - 加和乘的结构不一定使整数变大
  - 减和除不一定使整数变小
- 切记：安全函数如memcpy、strncpy的长度参数都是无符号整数

2.3 格式串漏洞

导致格式串漏洞的原因

- printf系列函数的问题：
  - 它们并不能确定数据参数arg1,arg2…究竟在什么地方结束，参数的个数不是确定的。只会根据format中的打印格式的数目依次打印堆栈中参数format后面地址的内容。
- 正确的写法

  printf(“%s”,buffer);
- 有问题的写法

  printf(buffer);

PRINTF()调用过程

以下面代码为例：

printf("Ais %d and is at %08x,B is %u and is at %08x\n",A,&A,B,&B)

- 参量被逆序压栈，最后是格式化字符串的地址，printf()每次遍历格式化字符串中的一个字符，
  - 如果该字符不是格式化参数的首字符（由百分号指定），复制输入该字符.栈指针下移
  - 若遇到一个格式化参数，就采取相应的动作，将栈中的变量pop()，与该参数对应。栈指针下移
- 重点：若格式化参数个数>参量个数，用printf()会从栈的当前指针开始，依次向下打印。

  printf("A is %d and is at %08x, B is %u and is at %08x.\n",A,&A,B,&B)

- 常用的格式化字符有：
- —%s：打印地址对应的字符串
- —%n：对该printf()前面已输出的字符计数，将数值存入当前栈指针指向的栈单元存储的地址中。
- —%m.nx：十六进制打印，宽度为m，精度为n，在m前加0处理为左对齐。

攻击原理

- 利用格式化函数（如prinf())的沿着堆栈指针向下打印的特性，通过只提供格式化字符串但不提供对应的变量，读取栈内空间的内容。更进一步，通过将某个要攻击的目标地址放入栈中，就可以利用格式化字符串读写里面的值。因此，它的攻击分为两步：
  a. 将目标地址放入栈中；
  b. 设计格式化字符串，读写目标地址里的值

格式串漏洞的利用原理

- 可以用%s、%x等查看堆栈中的数据
- %n能把显示内容的长度写入一个内存地址
- 可以出现多次写内存的机会

2.4 覆盖C++虚函数指针

虚函数

- 虚函数是成员函数而且是非static的成员函数。说明虚函数的方法如下： virtual 类型说明 函数名（参数表）
- 一旦在基类中指定某成员为虚函数，那么不管在派生类中是否给出virtual声明，派生类（以及派生类的派生类，依次类推）中对其重定义的成员函数均为虚函数。

运行时的多态性需要满足三个条件：

1. 类之间应该满足赋值兼容规则。
2. 要声明虚函数
3. 要由成员函数来调用或者通过指针、引用来访问虚函数。

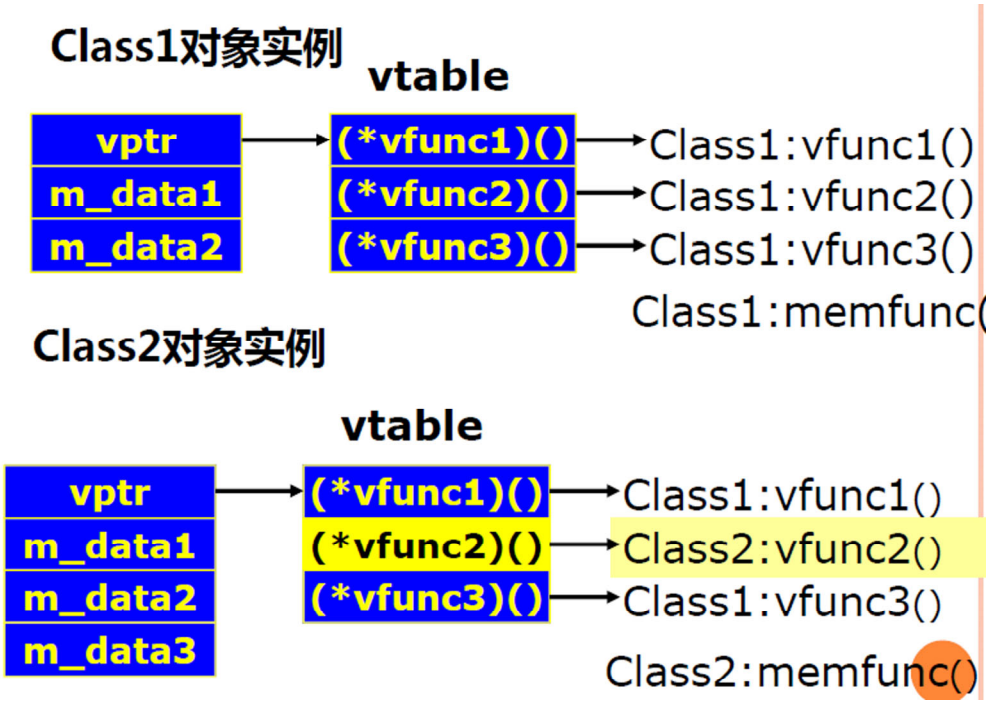
- 如果使用对象名来访问虚函数，则联编在编译过程中就可以进行，而无需在运行过程中进行

赋值兼容规则（子类型）

- 需要基类对象的任何地方都可以使用公有派生类的对象来替代包括：
  - 派生类的对象可以赋值给基类对象
  - 派生类的对象可以初始化基类的引用
  - 派生类的地址可以赋给指向基类的指针

虚函数的本质

- 为了达到动态绑定的目的，C++编译器通过某个表格，在执行期“间接”调用实际欲绑定的函数。这样的表格称为虚函数表（常被称为vtable）
- 每一个“内含虚函数的类”编译器都会为它做出一个虚函数表，表中的每个元素都指向一个虚函数的地址
- 编译器也会为类加上一项成员变量，是一个指向该虚函数表的指针（常被称为vptr）


Class1对象实例
vtable
vptr
m_data1
m_data2
(*vfunc1)()
(*vfunc2)()
(*vfunc3)()
Class1:vfunc1()
Class1:vfunc2()
Class1:vfunc3()
Class1:memfunc(
Class2对象实例
vtable
vptr
m_data1
m_data2
m_data3
(*vfunc1)()
(*vfunc2)()
(*vfunc3)()
Class1:vfunc1()
Class2:vfunc2()
Class1:vfunc3()
Class2:memfunc()