##重点整理堆的

# 前置知识

## 堆栈溢出:

事实上，堆和栈是不同的数据结构概念，堆栈溢出也可细化为堆溢出和栈溢出两种。栈有两个特性：只能从栈的顶端存取数据；数据的存取符合后进先出的原则。所谓后进先出，其实就如同自助餐中餐盘在桌面上一个一个往上叠放，在取用时先拿最上面的餐盘，这是典型的堆栈概念的应用。 堆是一种树结构，准确地说是一个完全二叉树。

在内存中，当一个可执行程序被装入到内存时，主要包括两个部分 ：代码和数据。代码会被装入到内存中的代码区，数据区又由 3 部分组成 ：①全局变量：根据其是否有初始值，被装入到内存中的未初始化数据区和初始化数据区；②局部变量：在函数调用发生时存放在栈中；③动态内存空间：在程序运行时申请的动态内存空间存放在堆中。

栈区(stack)是后进先出的结构，向低地址进行扩展，是一块连续的内存区域，栈顶的地址和栈的最大容量是系统预先规定的，只要栈的剩余空间大于所申请空间，系统将为程序提供内存，否则将报异常来提示栈发生溢出。栈空间是系统自动分配、释放的，存放函数的参数值、局部变量的值等。一般来说，进栈的顺序首先为主函数中的下一条指令（函数调用语句的下一条可执行语句）的地址先进栈，其次是参数由右往左依次进栈，最后是函数中的局部变量进栈，出栈顺序与进栈顺序相反，对于程序来说，出栈就意味着函数执行完毕，函数空间将被系统完全释放掉。

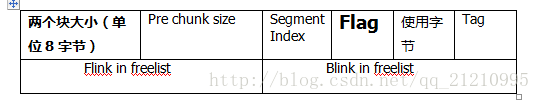
堆区一般由程序员自己申请，并指明大小，程序最后进行释放，若程序员不释放，程序结束时可能由操作系统回收（注意，如果是C/C++语言，程序不进行对空间回收，而Java语言中有专门的垃圾回收器进行回收），堆区与数据结构中的堆有所不同，分配方式类似于链表。堆区向高地址扩展。

溢出原理

堆栈溢出是说堆区和栈区的溢出，二者同属于缓冲区溢出。从上面关于堆区和栈区的解释可以看出，一旦程序确定，堆栈内存空间的大小就是固定的，当数据已经把堆栈的空间占满时，再往里面存放数据就会超出容量，发生上溢；当堆栈中的已经没有数据时，再取数据就无法取到了，发生下溢。需要注意的是，栈分为顺序栈和链栈，链栈不会发生溢出，顺序栈会发生溢出。

堆的数据结构

堆块  
1） 堆块分为块首和块身  
2） 块首大小为（8字节



注：flag 01 表示堆块占用

3） 空闲态的堆块在块首后有两个指针

堆表（空表和快表）

1） 空表：双向回环链表

2） 快表：单向链表

3） 区别

a) 以上两个都为128大小的指针数组 （空表每一项有两个指针，快表每一项有一个指针）

b) 快表最多只有四个节点

c) 空表除了数组的第一个元素外其他分别链接：数组下标\*8 大小的堆块，数组的第一个元素链接着大于1kb的堆块，并升序排序

d) 快表的堆块处于占用状态，不会发生堆块合并

e) 快表的只存在精确分配，快表优先空表分配

内存块的大小

小块（小于1kb）

分配方式：优先快表，其次空表非零元素（free[0]），然后堆缓存，最后空表零元素

大块（大于1kb小于512kb）

分配方式：优先堆缓存，其次空表零元素

巨快（大于512kb）

分配方式：虚分配

分配和释放是在程序提交申请和执行的，而堆块合并则是由堆管理系统自动完成的。

## 原因分析：

堆栈尺寸设置过小、递归调用过深、函数调用层次过深等程序设计不当之处都可能导致堆栈溢出。

1、 堆栈尺寸设置过小

由堆栈溢出的定义便可知，堆栈尺寸设置过小时，其能储存的内容过小，容易发生溢出。

2、递归层次太深或函数调用层次过深导致堆栈溢出

调用函数时，系统将为调用者构造一个由参数表返回地址组成的活动记录，并将其押入到由系统提供的运行时刻栈的栈顶，然后将程序的控制权转移到被调函数。若被调函数有局部变量，则在运行时刻，在栈的栈顶也要为其分配相应的空间，因此，活动记录和这些局部变量形成了一个可供被调函数使用的活动结构。被调函数执行完毕时，系统将运行时刻栈的栈顶的活动结构退栈，并根据退栈的活动结构中所保存的返回地址将程序的控制权转移给调用者继续执行。由此可见，当递归层次太深时或者函数调用层次过深时会产生大量的活动记录和局部变量，当超过栈的空间长度时，即发生溢出。

例如C/C++语言中的无限递归：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | int foo()  {      return foo()   //重复无限的自我调用  }    (define (foo) (foo)) //这样定义会造成死循环，但不会导致堆栈溢出  (define (foo) (+(foo)1)) //这样的定义会产生堆栈溢出 |

如果是C语言，由于没有垃圾资源自动回收机制，因此，需要程序主动释放已经不再使用的动态地址空间，如果不释放，程序结束后该部分空间依然存在，还可以继续访问，也就是说这部分依然占据着堆空间，剩余的堆空间减少，就可能造成堆区溢出。 而如果是Java语言则因为有专门的垃圾回收器回收则不会有此问题。

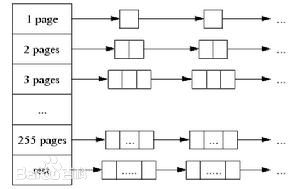
动态申请空间使用之后没有释放。由于C语言中没有垃圾资源自动回收机制，因此，需要程序主动释放已经不再使用的动态地址空间。申请的动态空间使用的是堆空间，动态空间使用不会造成堆溢出。

数组访问越界。C语言没有提供数组下标越界检查，如果在程序中出现数组下标访问超出数组范围，在运行过程中可能会内存访问错误。

指针非法访问。指针保存了一个非法的地址，通过这样的指针访问所指向的地址时会产生内存访问错误。

堆溢出原理：

堆(heap)也是一种基本的数据结构，它可由开发人员自行分配、释放堆是向高地址扩展的，不连续的内存区域。由于系统使用链表来管理空闲的内存块，堆自然是不连续的。 其中，链表的遍历是由低地址向高地址进行的。堆的存储结构如图：



在标准C语言上，使用malloc等内存分配函数获取内存即是从堆中分配内存，而在一个函数体中例如定义一个数组之类的操作是从栈中分配内存。从堆中分配的内存需要程序员手动释放，如果不释放，而系统内存管理器又不自动回收这些堆内存的话（实现这一项功能的系统很少）

，那就一直被占用。如果一直申请堆内存，而不释放，内存会越来越少，很明显的结果是系统变慢或者申请不到新的堆内存。而过度的申请堆内存（可以试试在函数中申请一个1G的数组！），会导致堆被压爆，结果是灾难性的。

我们掌握堆内存的权柄就是返回的指针，一旦丢掉了指针，便无法在我们视野内释放它。这便是内存泄露。而如果在函数中申请一个数组，在函数体外调用使用这块堆内存，结果将无法预测。　我们知道在c/c++ 中定义的数组大小必需要事先定义好，他们通常是分配在静态内存空间或者是在栈内存空间内的，但是在实际工作中，我们有时候却需要动态的为数组分配大小，这时就要用到堆内存分配的概念。在堆内存分配时首先应该知道操作系统有一个记录空闲内存地址的链表，当系统收到程序的申请时，会遍历该链表，寻找第一个空间大于所申请空间的堆结点，然后将该结点从空闲结点链表中删除，并将该结点的空间分配给程序，另外，对于大多数系统，会在这块内存空间中的首地址处记录本次分配的大小，这样，代码中的delete语句才能正确的释放本内存空间。另外，由于找到的堆结点的大小不一定正好等于申请的大小，系统会自动的将多余的那部分重新放入空闲链表中。堆内存是向高地址扩展的数据结构，是不连续的内存区域。这是由于系统是用链表来存储的空闲内存地址的，自然是不连续的，而链表的遍历方向是由低地址向高地址。堆内存的大小受限于计算机系统中有效的虚拟内存。由此可见，堆内存获得的空间比较灵活，也比较大。堆内存是由new分配的内存，一般速度比较慢，而且容易产生内存碎片,不过用起来最方便.另外，在WINDOWS下，最好的方式是用VirtualAlloc分配内存，它直接在进程的地址空间中保留一快内存，虽然用起来最不方便。但是速度快，也最灵活。

溢出危害：

从小处看，堆栈溢出会改变临近堆栈的空间中的内容，从而导致程序运行异常，发生故障；从大处看，堆栈溢出和计算机网络安全密切相关。堆栈溢出攻击是计算机被攻击的最为常见的一种形式，远程网络的攻击绝大多数是针对堆栈溢出的漏洞，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控制权。

一般后果

堆栈溢出时会访问不存在的RAM空间，造成代码跑飞，这时无法得到溢出时的上下文数据，也无法对后续的程序修改提供有用信息。

安全威胁

堆栈溢出常见的攻击类型有：修改函数的返回地址，使其指向攻击代码，当函数调用结束时程序跳转到攻击者设定的地址而不是原先的地址，修改函数指针，长跳转缓冲区来找到一个可供溢出的缓冲区。

攻击者通过缓冲区溢出来重写存储在返回地址内的值从而达到控制程序的执行流程的目的。程序函数就像是一个大程序中的小程序。它是相对独立的，对传给它的数据做相应的处理然后将处理的结果返回给主函数。因为数据在一个函数内进行处理，因此它用栈作为数据的临时存储区域。当一个程序调用函数时，它将所有的数据压栈，包括返回地址，如图所示：



当函数被调用时，指令指针指向的就是函数的返回地址。这一点很重要，因为当被调用函数执行结束以后，主程序要回到被调用函数的返回地址处，接着执行下一条指令。返回地址存储在RET中，当被调用函数执行结束，该返回地址传递给指令指针，以便主函数能够回到函数调用之前的地址继续执行。如果攻击者能够使缓冲区溢出并且重写存储在RET中的值，将恶意代码的地址赋值给RET，那么指令指针将指向恶意代码，从而执行恶意代码。

堆是程序中动态分配的内存空间，因为程序在执行前所需要的内存数量是未知的，因此堆内存在程序需要时进行动态分配，不需要时进行动态释放。堆和栈的主要区别在于，堆没有像栈那样的返回地址，这使得在栈溢出中用于控制程序执行流程的相关技术不可用。堆溢出可能导致重写数据或者指向其他函数的指针。这样，攻击者可以重写这些指针使其指向恶意代码，而不是指向原来的内存区域。

利用堆栈溢出攻击计算机的最典型的例子是1988年利用fingerd漏洞进行攻击的蠕虫病毒。

想写出堆溢出exploit，需要对堆中的重要数据结构掌握到字节级别。

# 利用思路

堆溢出利用（上）——DWORD SHOOT

链表“拆卸”中的问题

堆溢出利用的精髓就是用精心构造的数据去溢出下一个堆块的块首，改写块首中的前向指针flink和后向指针blink，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。

我们把这种能够向内存任意位置写入任意数据的机会称为DWORD SHOOT或arbitrary DWORD reset。注意：DWORD SHOOT 发生时，我们不但可以控制射击的目标（任意地址），还可以选用适当的子弹（4 字节恶意数据）。

通过DWORD SHOOT，攻击者可以进而劫持进程，运行shellcode，例如，下图中列出的几种情形。

int remove (ListNode \* node)

{

node -> blink -> flink = node -> flink;

node -> flink -> blink = node -> blink;

return 0;

}

防护绕过