Windows内核池溢出漏洞利用方法

kinvis@hotmail.com
SoBeIt
Beihang University

内存池的机制和算法

- 内存池的机制：
  - Ø内存池机制的概述
  - ØPoolDescriptor的介绍
- 内存请求处理的算法：
  - Ø请求不同大小内存时的分情况处理方法
  - ØLookAsideList及其算法

内存池的机制和算法

X'con 2005

----内存池的机制

- 内存池(Pool)用于内核态内存的分配，类似于用户态的堆(Heap)。分配和释放函数是ExAllocatePool()和ExFreePool().
- Pool的组织通过池描述符(PoolDescriptor)来完成，他的结构将在后面介绍到。
- Pool分为两类：非分页池(NonPagedPool)与可分页池(PagedPool)，区别在于可分页池允许换出内存，而非分页池必须常驻内存。

内存池的机制和算法

----内存池的机制

- 非分页池由两部分组成，分别由变量对**(MmNonPagedPoolStart, MnNonPagedPoolEnd)**和变量对**(MmNonPagedPoolExpansionStart, MmNonPagedPoolExpansionEnd)**决定，一般位于**0x8xxxxxxx**和**0xfxxxxxxx-0xffbe0000**。
- 而分页池则由变量对**(MmPagedPoolStart, MmPagedPoolEnd)**决定，一般位于**0xexxxxxxx**。

内存池的机制和算法

----内存池的机制

Pool的组织通过池描述符(PoolDescriptor)来完成，它的结构为:

```
typedef struct _POOL_DESCRIPTOR {
    POOL_TYPE PoolType;
    ULONG PoolIndex;
    ULONG RunningAllocs;
    ULONG RunningDeAllocs;
    ULONG TotalPages;
    ULONG TotalBigPages;
    ULONG Threshold;
    PVOID LockAddress;
    LIST_ENTRY ListHeads[POOL_LIST_HEADS];
} POOL_DESCRIPTOR, *PPOOL_DESCRIPTOR;
```

内存池的机制和算法

----内存池的机制

池描述符中有几项较为重要:

- **PoolType**：池的类型，可以是**PagedPool, NonPagedPool, NonPagedPoolMust**等，它们实际上是**PoolVector**的索引。
- **PoolIndex**：用于**PagedPool**，在可分页池数组中索引池描述符。
- **ListHeads**：池的分配粒度，也就是最小分配单位是**32**个字节。为了管理这些块，把相同块大小的空闲块串在同一个双向链表上。所以就会有**4096 / 32 = 128**条链表，也就是**POOL_LIST_HEADS**的值。

内存池的机制和算法

----内存池的机制

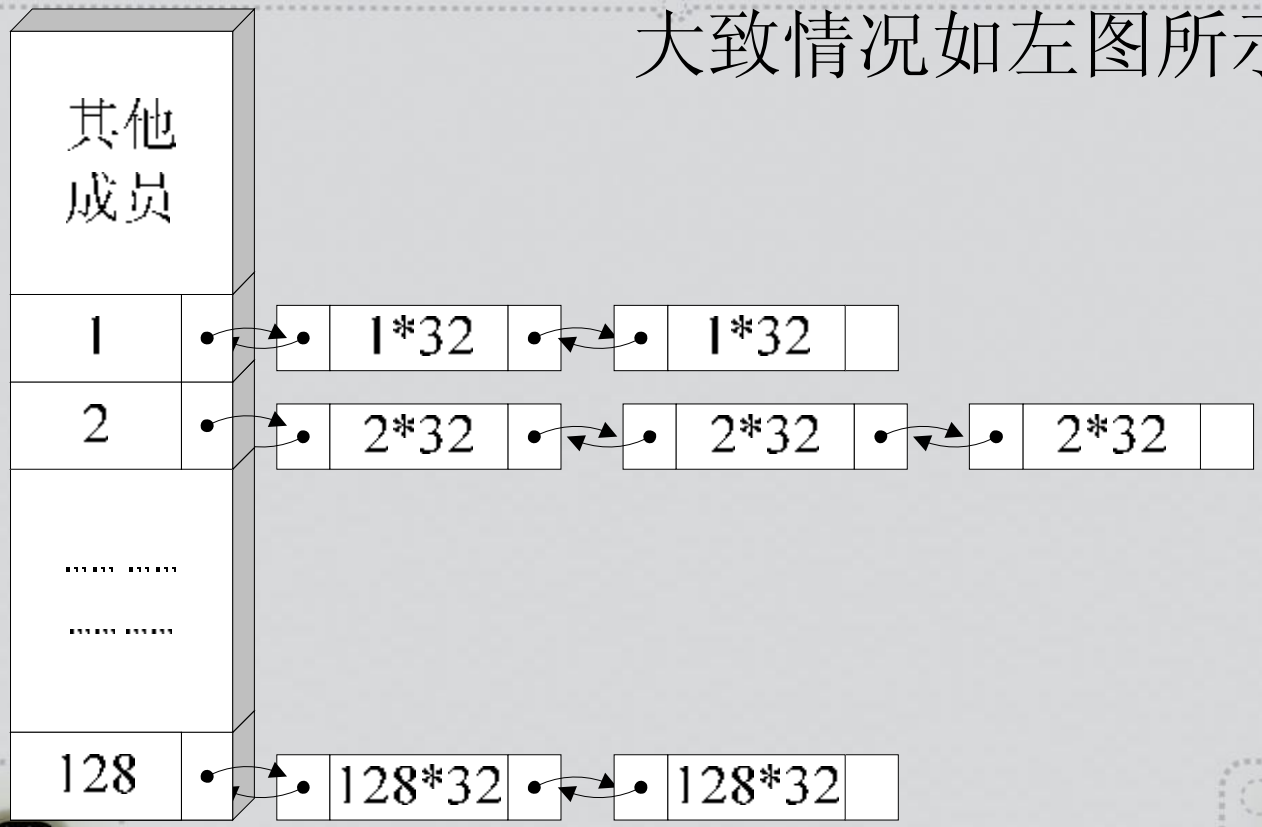
系统的PoolDescriptor由一个PoolVector全局数组来组织，它一般包括3个成员：

指向两个静态分配的非分页池描述符NonPagedPoolDescriptor与NonpagedPoolDescriptorMS的指针，以及一个指向可分页池描述符指针数组的指针。

内存池的机制和算法

----内存池的机制

大致情况如左图所示：



内存池的机制和算法

----内存池的机制

```
typedef struct _POOL_HEADER {
   union {
      struct {
         UCHAR PreviousSize;
         UCHAR PoolIndex;
         UCHAR PoolType;
         UCHAR BlockSize;
      };
      ULONG Ulong1;
   };
   union {
      EPROCESS *ProcessBilled;
      ULONG PoolTag;
      struct {
         USHORT
     AllocatorBackTraceIndex;
         USHORT PoolTagHash;
      };
   };
} POOL_HEADER, *PPOOL_HEADER;
```

每个申请到的池和堆一样，都有一个管理结构，它的定义如左:

内存池的机制和算法

----内存池的机制

- **PreviouSize:** 前一个块的大小，这里的**Size**都是实际的大小除于**32**。若为每个页的第一个块，则为**0**。
- **PoolIndex:** 对于**PagedPool**，将在可分页池描述符中循环分配，**PoolIndex**是该块在所属的可分页描述符数组中的索引。空闲的块**PoolIndex**为实际的索引，已分配的块**PoolIndex**是实际的索引加上**0x80**。系统在进行释放通过该成员**&0x80**来判断该池块是否被释放。
- **PoolType:** 空闲时为**0**，分配时为该池类型加上**1**。系统在进行释放合并时通过判断相邻块该成员是否为**0**来判断是否空闲。
- **BlockSize:** 当前块大小，它是请求的大小加上管理结构**8**个字节和一个**7**字节后除于**32**。
- **PoolTag:** 正常的块申请，这里是**4**个字节的字符，根据申请的内容不一样而不同。

内存池的机制和算法

----内存请求处理的算法

对内存的请求根据请求的大小分为3种情况:

情况一:

当请求大小大于**0xfb8**，也就是大于一个页大小减一个池管理结构大小减一个块的单位**(4096-8-32)**，通过页面分配器**MiAllocatePoolPage**，直接分配对齐的一个或几个整页。

内存池的机制和算法

----内存请求处理的算法

情况二：

请求大小大于0x100字节小于0xfd8字节时，将从池描述符的ListHeads链表中摘除合适的块返回给请求者。分配的算法使用了buddy算法的简化版，通过遍历链表，直到找到满足条件的块为止，把该块从原链表上摘除并不断进行切割直到符合请求的大小为止，把切割下来的块插入到对应大小的链表中。当释放块时，可以把前后的相邻空闲块摘除并进行合并，然后插入更大块所对应的链表中。

内存池的机制和算法

----内存请求处理的算法

情况二（续）：

当是在**PagedPool**中分配时，分配算法通过轮转法**(Round-Robin)**在可分页池描述符数组中除第**0**项外剩下几项进行循环，当获得某个池描述符的锁时，就从该池分配内存，下次则会从下一个池描述符开始申请锁。所以当两次连续的调用**ExAllocatePool**请求可分页池时，绝大多数情况下获得的内存分别属于两个不同的池。

内存池的机制和算法

----内存请求处理的算法

情况三：

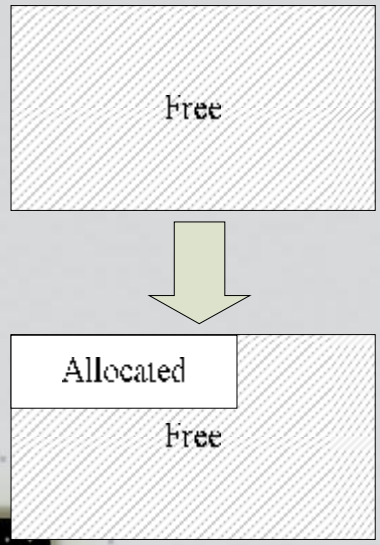
请求大小小于等于0x100字节时，由于这种大小的池块申请频繁，为提高效率则优先从Lookaside链表中分配和释放。Lookaside是通过单向链表实现的一个堆栈数据结构，位于KPCR中，PagedPool和NonPagedPool各8个PP_LOOKASIDE_LIST，对应从32到256的大小。每个这样的结构包含两条链表，通过二叉树组织，每次分配和释放都会自动平衡深度。

内存池的机制和算法

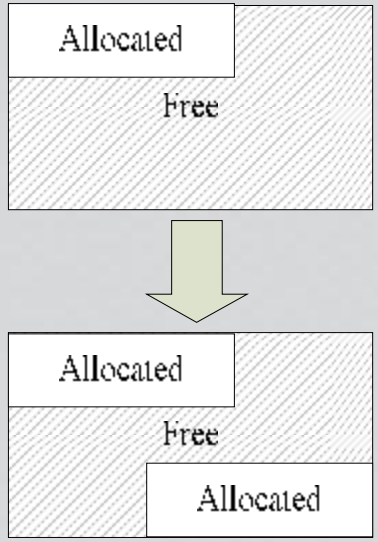
----内存请求处理的算法

分配顺序：

空闲页的分配：
---从头开始分配



非空闲页的分配：
---从尾开始分配



内存池的机制和算法

----内存请求处理的算法

LookasideList及其算法：

LookasideList 基于**pool**分配器，通过把常用大小的池块事先调用**ExAllocatePool**分配好串在链表上，这样下次分配时就从链表中直接摘取池块即可。
LookasideList通过单向链表组织，可以自平衡深度，在频繁地从数上摘取池块时**Lookaside**平衡函数会调用**ExAllocatePool**，反之链表上池块过多时会调用**ExFreePool**。

内存池的机制和算法

----内存请求处理的算法

LookasideList及其算法（续）：

可以调用**ExInitializePagedLookasideList**或**ExInitializeNPagedLookasideList**建立可分页池或不可分页池的**LookasideList**，并指定该**LookasideList**中池块大小。之后就可以通过调用**ExAllocateFromPagedLookasideList**或**ExAllocateFromNPagedLookasideList**从**Lookaside**获取前面指定大小的池块，通过调用**ExFreeToPagedLookasideList**或**ExFreeToNPagedLookasideList**将池块释放给**LookasideList**。

系统中还有几个自己使用的**LookasideList**存放在**KPCR**的**PPLookasideList**数组中，该数组有**16**项，而系统只使用了其中的**7**项。

与堆溢出相比利用的难度

- 没有用户态的每进程的默认堆，更不能象用户态创建自己的堆。所有内核态程序共用那几个池，增加了地址分配的不确定性，尤其是**PagedPool**，在两个池描述符中轮流分配，要想控制连续地址分配和释放比中六合彩难度还大。因此堆溢出的绝大部分方法无法使用。
- 当池被溢出后，无法象用户态通过创建一个堆来替换默认堆，只能手工修复池，所以要尽可能小地破坏池描述符。
- 池没有堆那样的通过构造标记**LAST_ENTRY**的堆来精确定位**shellcode**，这样**shellcode**的定位比较困难。

与堆溢出相比利用的难度

- 池溢出发生在内核态，IRQL很可能是DISPATCH_LEVEL，拿到控制权的时候是在发生了异常之后。内核态的异常比用户态异常要致命，若恢复不好将直接导致蓝屏。所以要根据实际情况进行恢复。
- 该覆盖哪些函数指针才可以拿到控制权。

溢出利用方法

我选择溢出覆盖KiDebugRoutine，它是一个函数指针，是系统内置的内核调试机制的接口。每次异常发生由内核核心异常分配函数KiDispatchException都会判断KiDebugRoutine是否为空并调用它。所以可以覆盖它拿到控制权并恢复正常流程。异常会在系统释放伪造池或者释放伪造池的下一个池时被触发。

溢出利用方法-Ⅰ

----构造空闲池法（不推荐）

通过在被溢出池后面构造一个空闲池块，当被溢出池被释放时，发生与后面空闲池的合并操作，将可以覆盖任意4个字节。通过覆盖KiDebugRoutine地址里的函数指针，将可以拿到控制权。由于被溢出池地址放在了堆栈中，可以用包含一个跳转该地址的指令的地址进行覆盖。

溢出利用方法-Ⅰ

----构造空闲池法（不推荐）

- 优点：适用于**PagedPool**和**NonPagedPool**
- 缺点：不能是页中最后一个池块，否则不会向后合并；该溢出池的地址距离当前堆栈指针距离较远，在正常系统中很难找到合适跳转指令。

溢出利用方法-Ⅱ

----跨页合并空闲池法（推荐）

在被溢出池后面构造一个空闲池块，并使该池大小加上被溢出池大小超过一页，这样在合并完空闲块覆盖任意4字节后，接下来的**AddListTail**操作所插入的链表头是一段有限的我们可以控制的地址，这时还会有一个任意4字节的覆盖。可以精确定位**SHELLCODE**

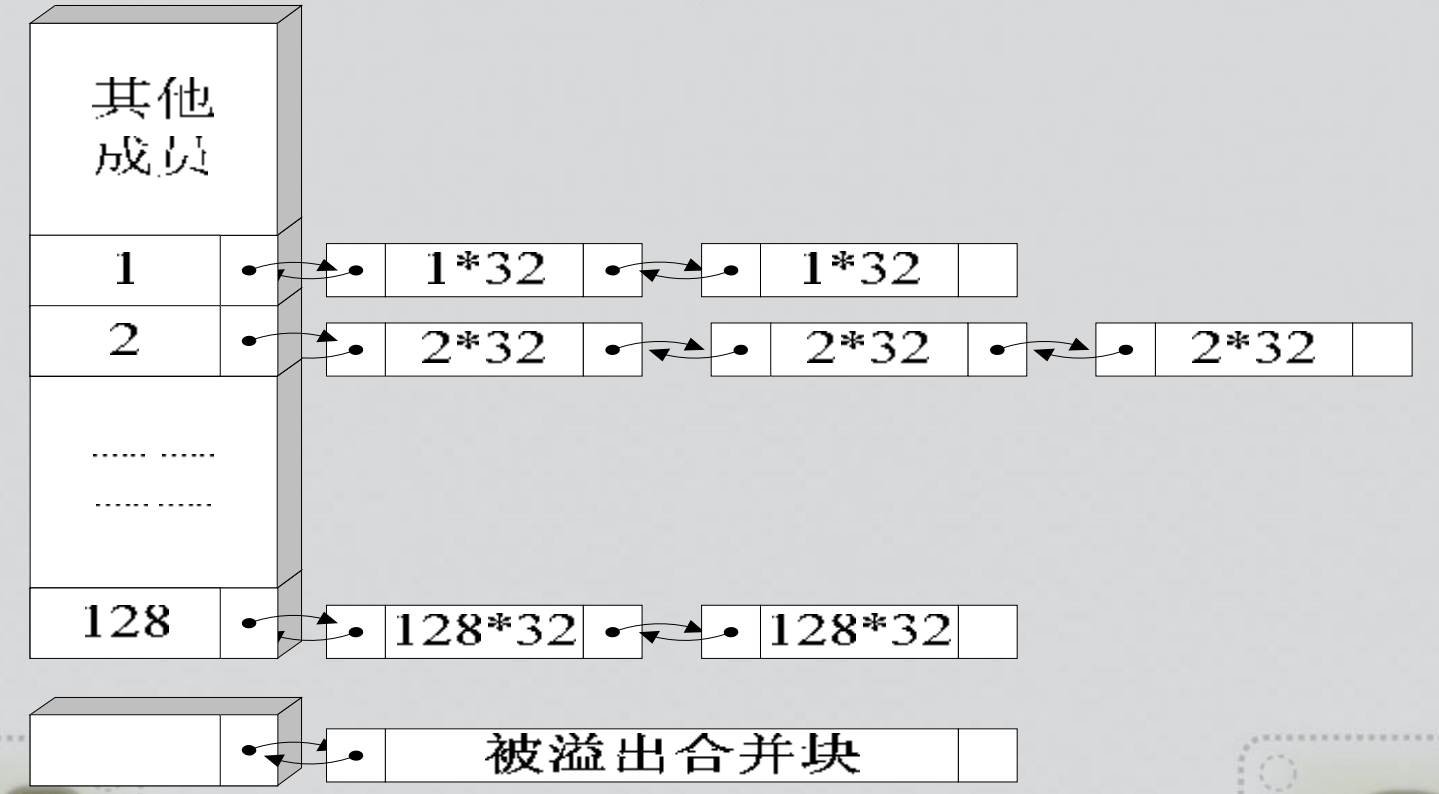
溢出利用方法-Ⅱ

----跨页合并空闲池法（推荐）

链表头地址将会被写入被溢出池头结构之后的地址，所以该地址的最后一个字节决定了指令，该字节是**0xx0**或**0xx8**。可用以下指令实现跳转:**0xe0(loopnz 0xxxxxxxxx)**、**0x70(jo 0xxxxxxxxxx)**、**0x78(js 0xxxxxxxxxx)**。

溢出利用方法-II

----跨页合并空闲池法（推荐）



溢出利用方法-Ⅱ

----跨页合并空闲池法（推荐）

- 第一次指针互写(RemoveEntry())

```
8046b6de 890a          MOV    [EDX],ECX
8046b6e0 895104        MOV    [ECX+0X4],EDX
```

- 第二次指针互写(AddListTail())

```
8046b7b5 8b54cf1c      MOV
    EDX,[EDI+ECX*8+0x1c]
8046b7b9 8d44cf18      LEA
    EAX,[EDI+ECX*8+0x18]
8046b7bd 8d4e08        LEA    ECX,[ESI+0X8]
8046b7c0 89560c        MOV    [ESI+0XC],EDX
8046b7c3 8901          MOV    [ECX],EAX
8046b7c5 890a          MOV    [EDX],ECX
8046b7c7 894804        MOV    [EAX+0X4],ECX
```

溢出利用方法-Ⅱ

----跨页合并空闲池法（推荐）

- 优点：精确定位**shellcode**；恢复池管理结构容易。
- 缺点：只适用于静态分配池描述符的**NonPagedPool**，而不适用于动态分配池描述符的**PagedPool**。不能是一页中最后一个池块，否则不会向后合并。必须假设被溢出池前一个池不为空闲池，因为不能向前合并。

溢出利用方法

以上两种利用溢出的方法的共同缺点：

不稳定，太依赖系统SP版本。

溢出利用方法

----ShellCode做了些什么

- (1)修复池描述符的池块链表。将池描述符中所有池链表初试化，并遍历被溢出池同一页内所有池块，根据前后池块的大小来修正每个池块大小，并把PoolIndex设为0x80，把PoolType设为1，防止池块被向前向后合并。
- (2)搜索NTOSKRNL导出表中需要用到的函数。
- (3)搜索SYSTEN权限的进程如lsass.exe、csrss.exe、serverice.exe等进程里处于可报警(Alertable)状态的线程(肯定能找到)，并把要执行用户态shellcode的APC插入该线程等待它执行。
- (4)恢复异常。在shellcode里循环调用ZwYieldExcution，阻止异常分派函数在该异常的返回。

演示

Any questions?

Thanks!