



# Reliable Windows Heap Exploits

---

Matt Conover & Oded Horovitz  
翻译: ICBM

- 堆溢出利用的介绍
- Windows 堆管理内部机制
- 覆盖写任意内存的说明
- 写任意内存的应用和一个利用的演示
- 对 heap shellcodes 的特殊说明
- XP SP2 堆保护机制
- Q & A

- 堆溢出攻击成为主流
  - DCOM (seems to be the inflection point), Messenger, MSMQ, Script Engine
- 从事堆溢出的研究者:
  - David Litchfield – “Windows 堆溢出”
  - LSD – “Microsoft windows RPC 安全弱点”
  - Dave Aitel – “MSRPC 堆溢出利用I,II”
  - Halvar – “第三代溢出利用”

- 尽管许多专家利用各种巧妙的技术作为exploit的主要因素
  - 做4字节的覆盖（后面讨论）是一个猜测的工作
  - 失败的原因不是很清楚
- 可用的利用程序都是有版本依赖的
  - Shellcode地址未知，
  - 不同的sp版本中，SEH地址各不相同
  - 异常处理中，指向buffer的指针可以在栈中找到（在异常的记录中）
  - 需要一个访问栈的指令的地址，这也是和sp版本相关的

- 发现了在各个阶段堆溢出的技术，极大地改善了（堆溢出利用的）可靠性
- 更好地理解了windows堆的内部机制和内部过程
- 弄清楚了为什么现有的技术不可靠
- XP SP2 极大地增强了保护性，并且使当前所有的技术均不再可用

- 包括的内容

- 有助于堆溢出利用的堆内部机制
  - 对于进程的关系
  - 堆的主要数据结构
  - 分配与释放算法

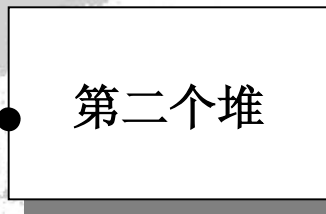
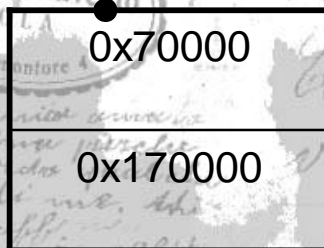
- 不包括的内容

- 烦你到死的堆内部机制
- 和溢出利用没有直接关系的部分
- “慢速”分配或者堆调试的算法

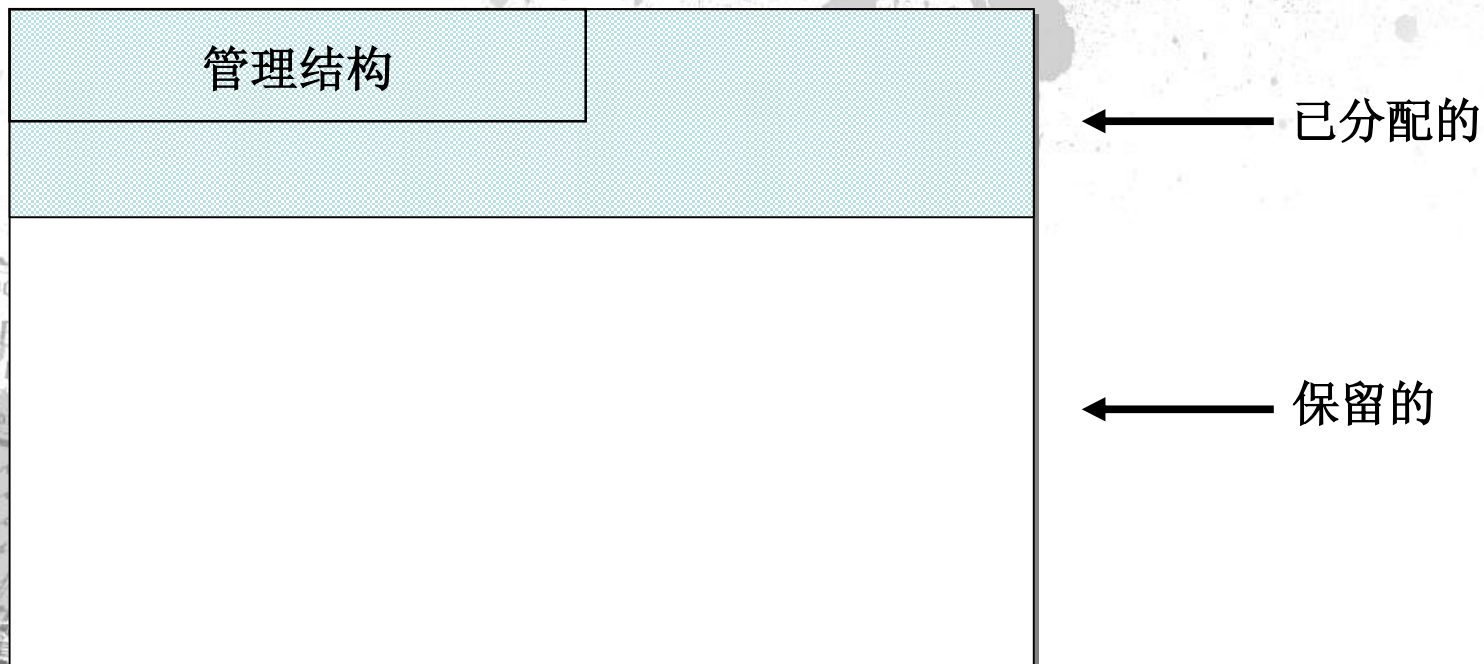
- 多个堆可以共存在一个进程内

PEB

0x0010			默认堆
0x0080			堆计数
0x0090	堆链表		



- 堆起始于一个很大的段
- 大部分的段内存被保留使用
- 堆管理结构也是从堆中分配出来的！



- 重要的堆结构

Segment Table

Segments

Virtual Allocation list

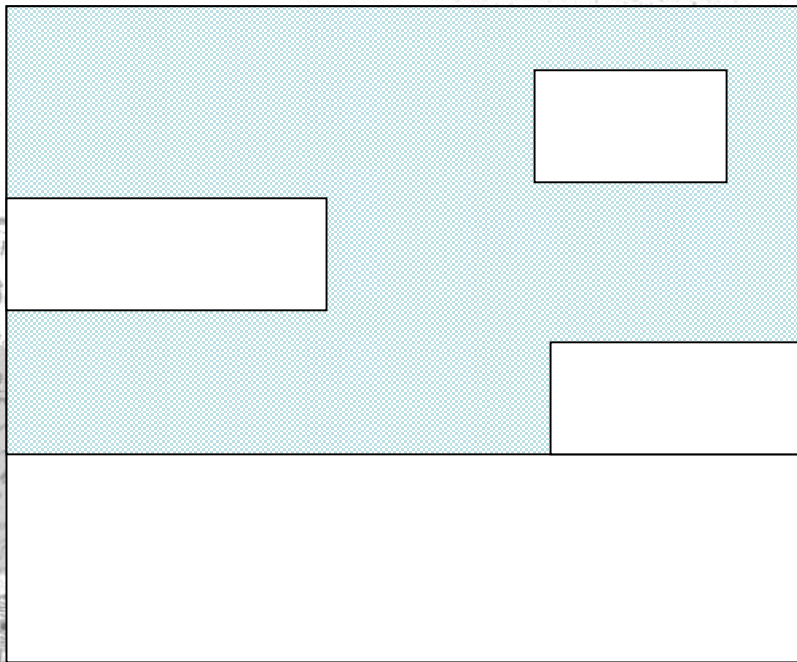
Free list usage bit map

Free Lists Table

Look aside Table



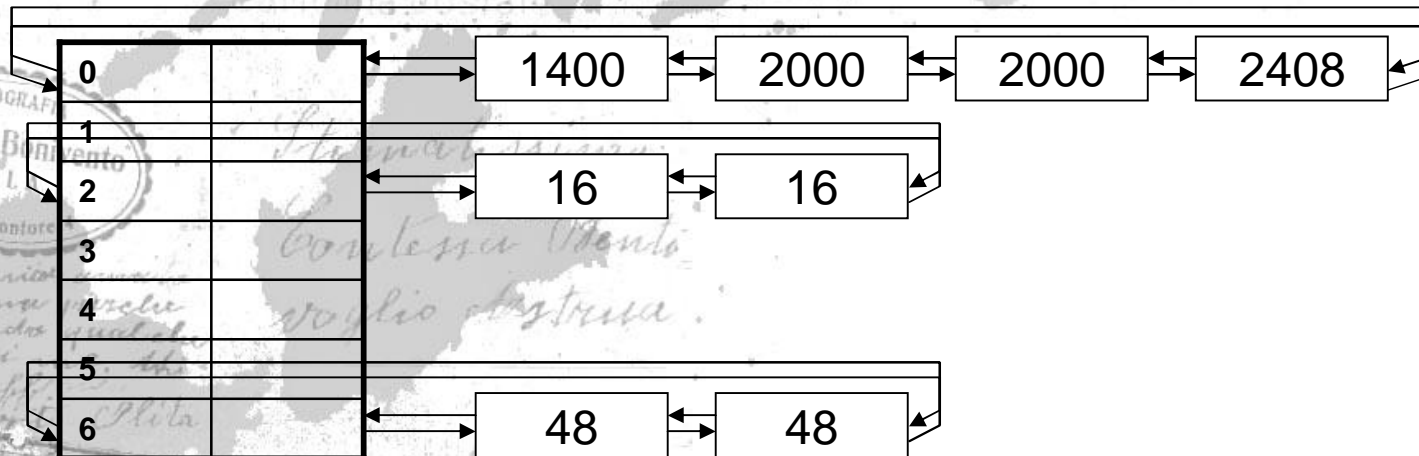
- 段管理
  - 段界限 (页内)
  - 未指定块的链表
  - 空闲/保留页计数



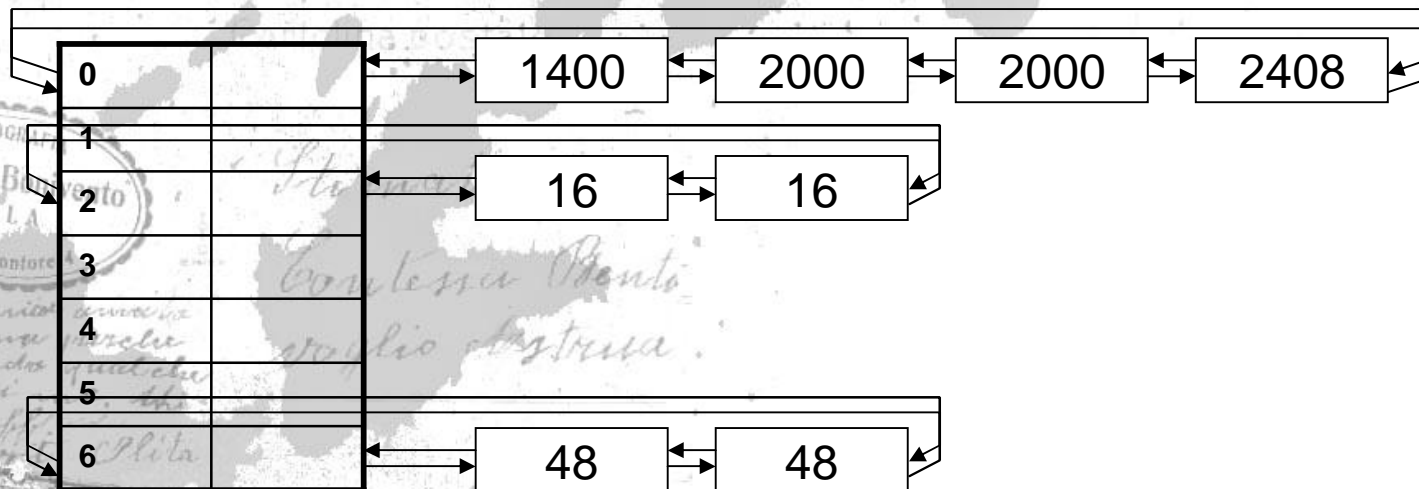
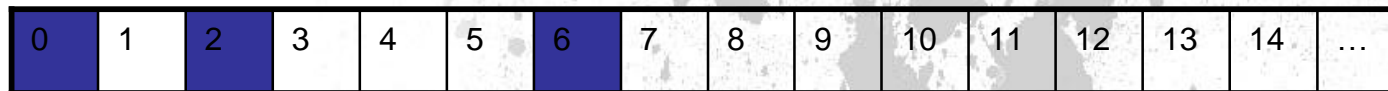
保留的

已分配的

- Free List 管理
  - 128 空闲块的双向链表
  - 块大小为表的行号 \* 8 字节
  - 条目 [0] 是一个例外，他包含了一系列从小到大顺序排列的buffers，每个的字节数在1024到虚拟分配极限之间



- 空闲链表使用情况位图
  - 搜寻空闲链表的快捷方法
  - 128 位 == 4 字节长 (每个32位)

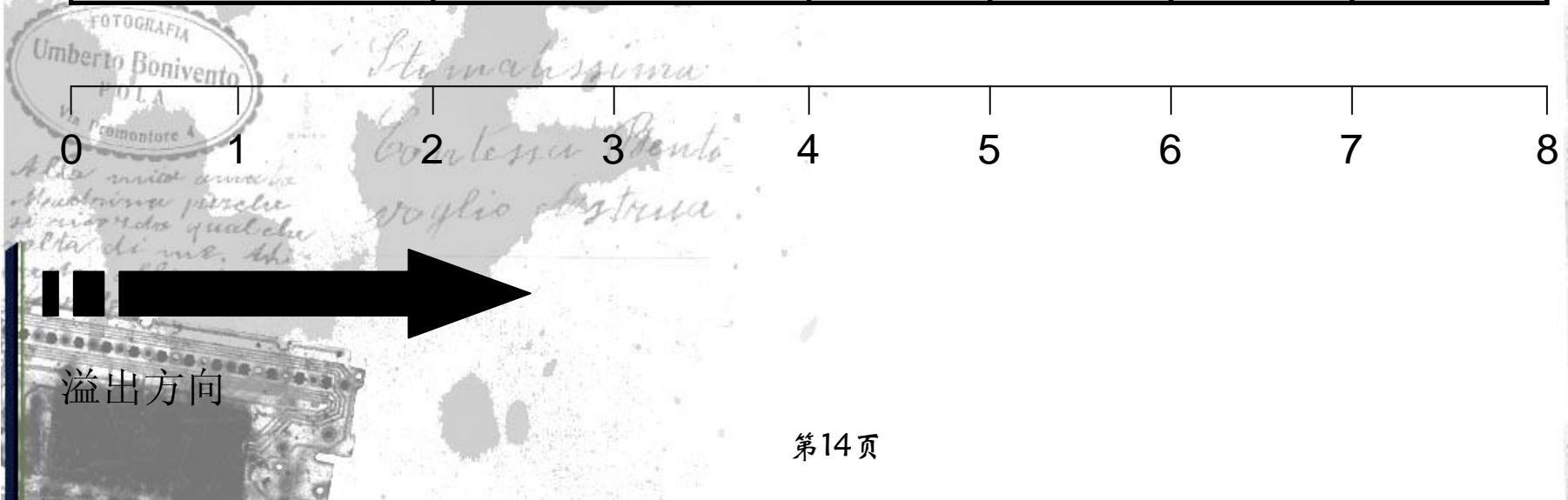


- Lookaside表
  - 释放与分配的最快的路子
  - 开始为空
  - 128 单向忙块的链表 (已经空闲，但是被临时占用)
  - 自动平衡深度来优化性能



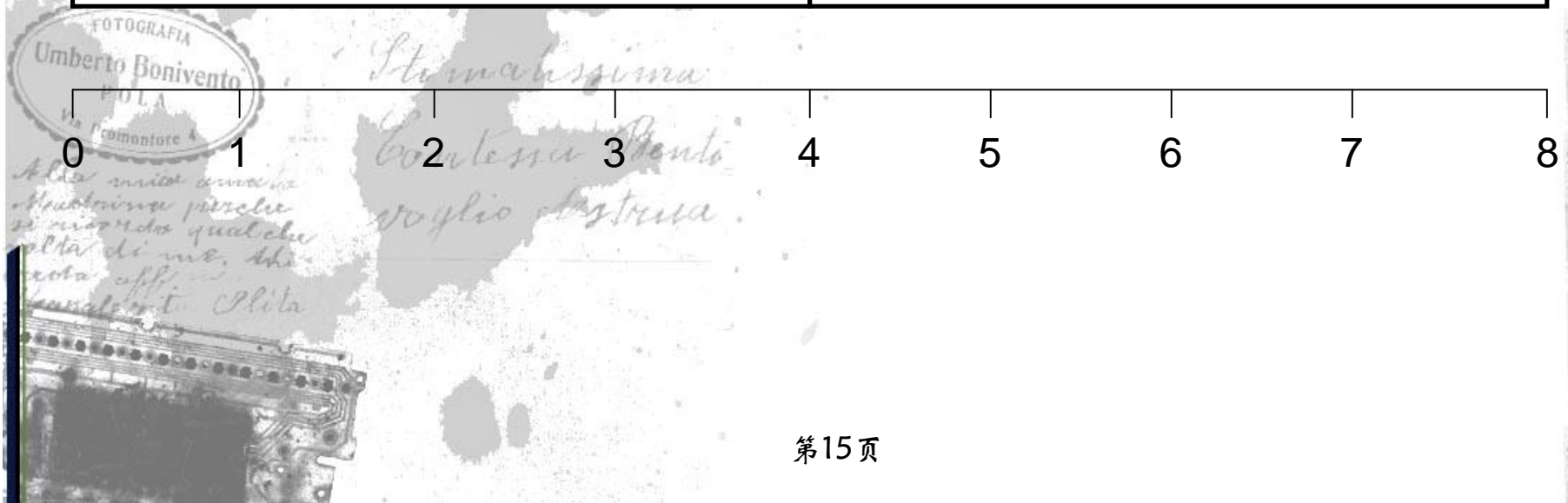
- 块的基本结构 – 8 字节

- 01 – Busy
- 02 – Extra present
- 04 – Fill pattern
- 08 – Virtual Alloc
- 10 – Last entry
- 20 – FFU1
- 40 – FFU2
- 80 – No coalesce



- 空闲块的结构 – 16 字节

自身长度	前一个块的长度	段索引	Flags	未使用的字节	标签索引 (调试用)
下一个块			前一个块		

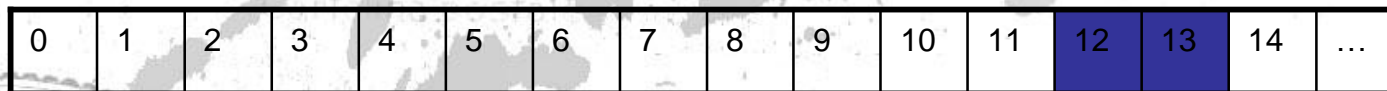




- 分配算法 (高层)
- 调整长度. 加 8, 向上与8字节对齐
- 如果长度比虚拟分配极限要小 {
  - Lookaside
  - 空闲链表
  - Cache
  - 空闲链表[0]
- 如果找不到可用内存, 就需要扩展堆
- }
- 如果长度  $\geq$  虚拟分配极限
  - 从操作系统中分配内存, 并把块加入到虚拟分配块 (buffer) 链表中去

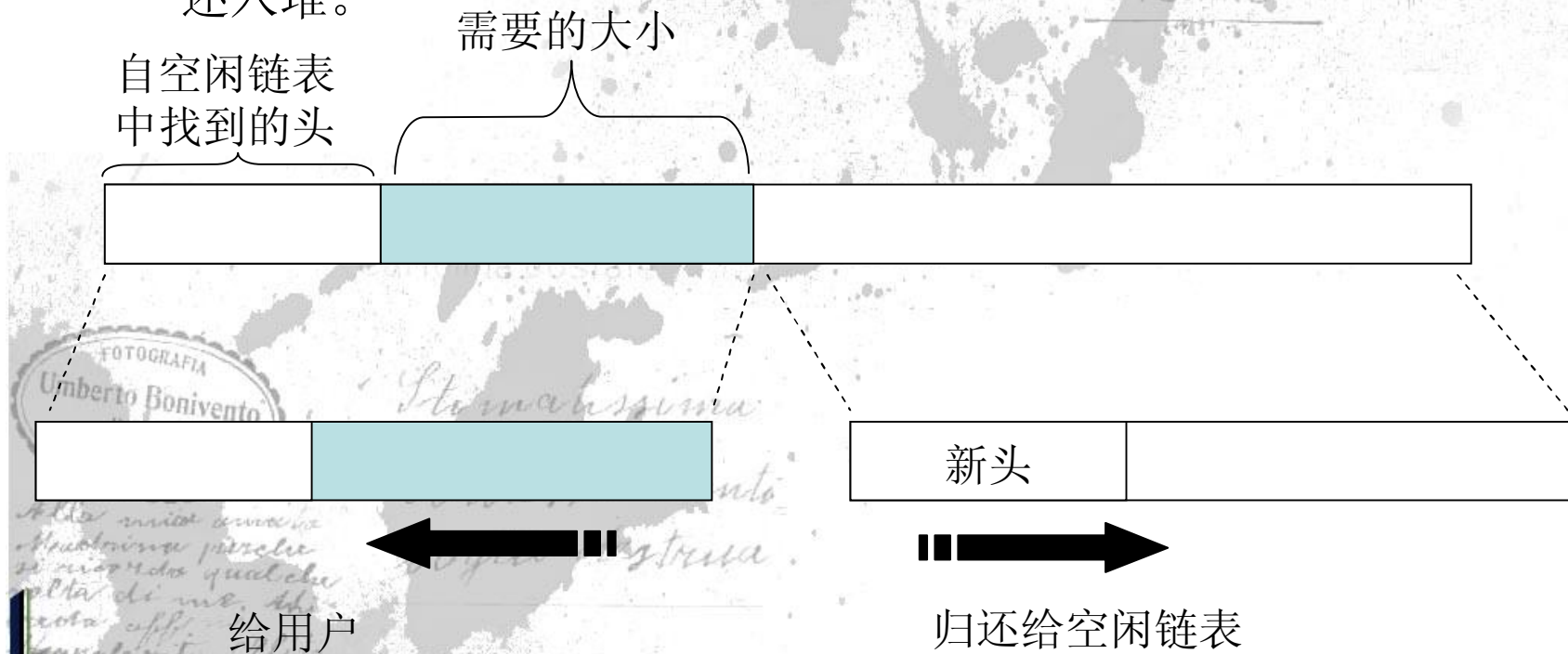
- 分配算法 – Lookaside 搜索
- 从Lookaside中获得buffer, 当且仅当:
  - 存在 Lookaside 表
  - Lookaside 未被锁定
  - 需要的长度小于 1024 (to fit the table)
  - 刚好满足所请求的大小 (例如, Lookaside 不为空)
- 如果 lookaside 不为空, 就从 Lookaside 中移除并返回给用户。

- 分配算法 – 搜寻空闲链表
- 搜寻可用的空闲链表位图来找到大小合适的项
- 例子:
  - 用户申请需要64字节
  - 从入口  $64/8 == 8$  开始寻找
  - 在入口号为12的地方找到，块大小为  $12*8 == 96$  字节



- 如果在位数组中没有找到任何项，那么就从堆 Cache 或者返回 FreeList[0] 中返回。

- 分配算法 – 搜寻空闲链表
- 如果块是从空闲链表中获得的，那么要检查它的大小。如果块的大小比我们需要的大16或者更多的字节，则将块分开，（多余的部分）归还入堆。



- 在 FreeList[0] 中寻找块
- 用于cache不存在或者是为空的时候
- 通常发生在块大小 > 1K 的时候
- FreeList[0] 是由小到大顺序排列的
- 检查 FreeList[0]->Blink 看其是否够大 (最大的一块)
- 返回的是满足需求的最小的块, 如下:
  - While (!BigEnough(Entry->Size, NeededSize))
  - Entry = Entry->Flink

- 堆 Cache 的内部机制
- 块大小 $>4k$ 的释放极限时，要累积到总的需要释放字节 $>64k$ 时候才会被一并释放，并且归还到未分配的空间中
- 这样做的代价太大。因此，在windows 2000 SP2 中增加了堆cache。
- 默认情况下它是禁用的，只有当程序短暂地使用大块（频繁地分配/释放 $\geq 4k$ 的块）才会被创建

- 堆 Cache 的内部机制
- 基本上类似于块大小 $<1k$ 的空闲链表
- 它有一个基于释放极限的固定大小 (896 项)
- Cache表中的每一项都是有着特定大小的双向链表块 (除了 Cache表中的最后一个项)
- 如果提供了堆cache而且非空, 那么它将在 FreeList[0]前起作用.

- 堆 Cache 的内部机制
- Cache表索引号 = 块大小 - 1K (0 就是 1024, 1 就是 1032, 等等)
- Cache中的最后一项等价于旧的FreeList[0] (顺序的空闲块链表)
- 故>8k的块都被放到了 CacheTable[895] 中

- 在 Cache 中寻找可用块
- If (Index != LastEntryInCacheTable and ChunkTable[Index] != NULL) return chunk
- Else If (Index != NumEntries-1)
- 遍历 ChunkTable[Index]并且返回第一个合适的块。
- Else
- 利用cache表位图来寻找一个更大的项(这里的位图工作方式和前面的空闲链表位图是一样的) 返回没有使用的部分到空闲链表

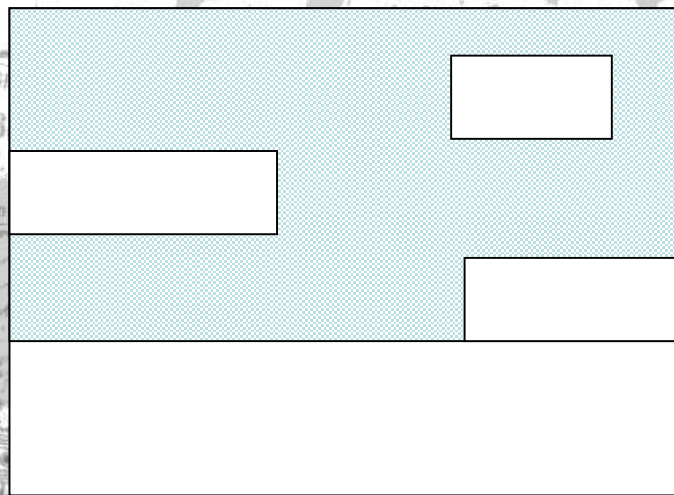
- Cache存在情况下的堆利用...
- 对于使用大块的程序，要将cache置为NULL！这样才能确保 FreeLists[0] 被使用
- 另外, Litchfield 使用FreeList[0]的堆清除技巧 (BlackHat Windows 2004中讨论过)不再有效!

- 分配算法 – 堆扩展

如果没有块满足要求，同时堆（大小）是可增长的，则从段保留内存中分配更多的内存。

尽可能地重用内存里未分配范围中的“洞”。

如果现有的段没有足够的保留内存，或者不能被扩充，那么就创建一个新的段。



保留的

分配的

- 分配算法 – 虚拟分配
- 当块大小 > 虚拟分配极限(508K)的时候使用
- 虚拟分配的头部被放置在buffer的开始
- Buffer被添加到虚拟分配buffer的忙碌链表中去  
(这也就是Halvar's VirtualAlloc 覆盖写伪造的地方)

- 释放算法

• 连接 buffer 并且放置到空闲链表或者cache中

• 将buffer释放还给操作系统

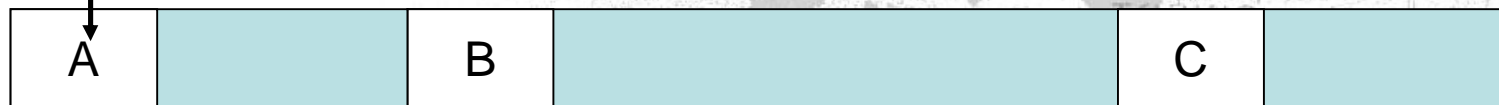
• }

- 释放算法 – 释放到 Lookaside
- Buffer被释放到 Lookaside 当且仅当：
  - 存在Lookaside 表
  - Lookaside 表未被锁定
  - 请求长度小于1024字节 (满足表要求)
  - Lookaside 没有满
- 如果buffer能够被放置到Lookaside中去, 则保持buffer的标志位为忙, 并且返回给调用者

## • 释放算法 - 连接

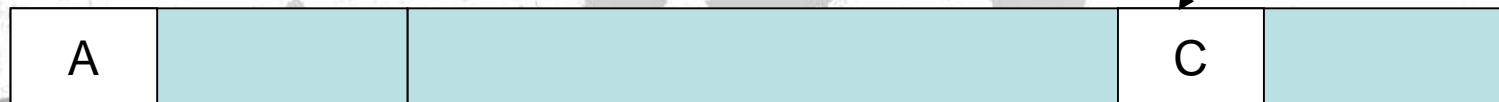
步骤2: 将Buffer从空闲链表中移出

步骤1: 释放Buffer



A + B 连接

步骤3: 将Buffer从空闲链表中移出



A + B + C 连接



步骤4: 将Buffer放回空闲链表中

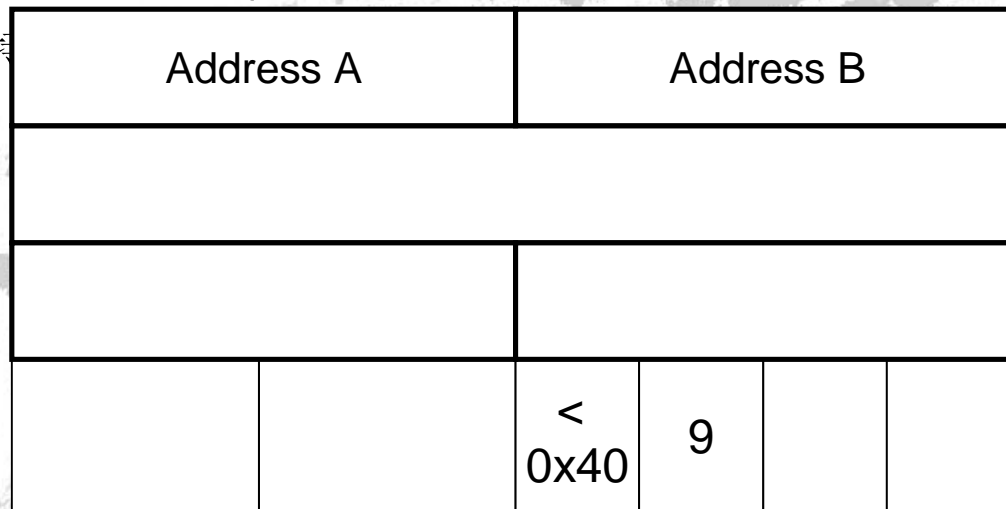
- 释放算法 – 连接
- 什么情况下连接情况不会发生
- 被释放的块是虚拟分配的
- 块标志字的最高位被置位
- 被释放的块是第一块  $\rightarrow$  不会向后连接
- 被释放的块是最末块  $\rightarrow$  不会向前连接
- 连接到的块当前状态为忙
- 连接后的块长度将  $\geq 508K$

- 释放算法 – 将连接后的块放入空闲链表
- 如果块长度  $< 1024$ , 则插入到适当的空闲链表项中
- 如果块长度  $>$  释放极限并且总的堆空闲长度超过了一次性释放的下限, 则把buffer释放给操作系统. 注意: 如果这一步骤多次发生, 那么堆cache将会被创建
- 如果块长度小于虚拟分配的极限, 则插入到空闲链表的[0]项中去
- 如果块长度大于虚拟分配的极限, 则将其拆分为小块 (每一块尽可能的大), 并把它们插入到 cache (如果存在的话) 或者空闲链表的 [0]项去.

- 概要 – 问题?
- 主要结构 – 段, Lookaside, 空闲链表, Cache, 空闲链表的 [0]项, 虚拟分配链表
- 释放 / 分配算法的工作顺序
  - Lookaside
  - 空闲链表
  - Cache
  - 空闲链表第[0]项
- 堆内存是完全可回收的
  - 大块的buffers在分配时被分开
  - 小块的buffers通过连接形成更大的buffers

- 4字节覆盖：
  - 可以用任意32位值覆盖32位的任意地址的值
- 4到n字节覆盖：
  - 用4字节覆盖间接的覆盖任意n字节
- 双4字节覆盖：
  - 一次操作覆盖两个4字节 (如free操作)
- AddressOfSelf Overwrite:
  - 4-byte overwrite where you set WhereTo, and WithWhat is already the address of a chunk you control

- VirtualAlloc 4字节覆盖 (Halvar)
- 利用虚拟内存分配地址
- 当被溢出堆块被释放后, 任意地址值就会被覆盖 (如果已经释放就不会产生覆盖操作)
- 构造特



Overflow  
start

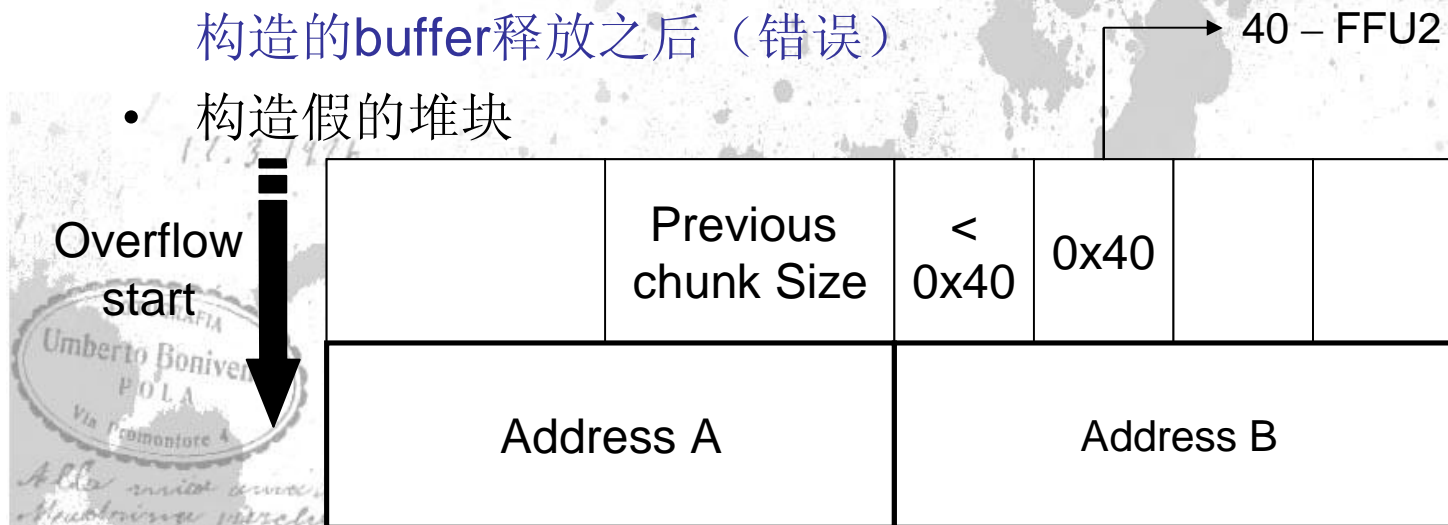
01 – Busy

08 – Virtual Alloc

- VirtualAlloc 4字节覆盖
- 该技术的优势
  - 如果下一个buffer的状态为busy, 覆盖任意地址后基本上不会破坏堆的结构
- 该技术的缺点
  - 如果溢出有字符串操作, 你不能覆盖包含有NUL字节的内存
  - 在被溢出的内存里至少有24字节的数据
  - 如果buffer的状态不是busy, 将不能覆盖任意内存地址, 并且可能破坏堆结构 (下一页详细讨论)

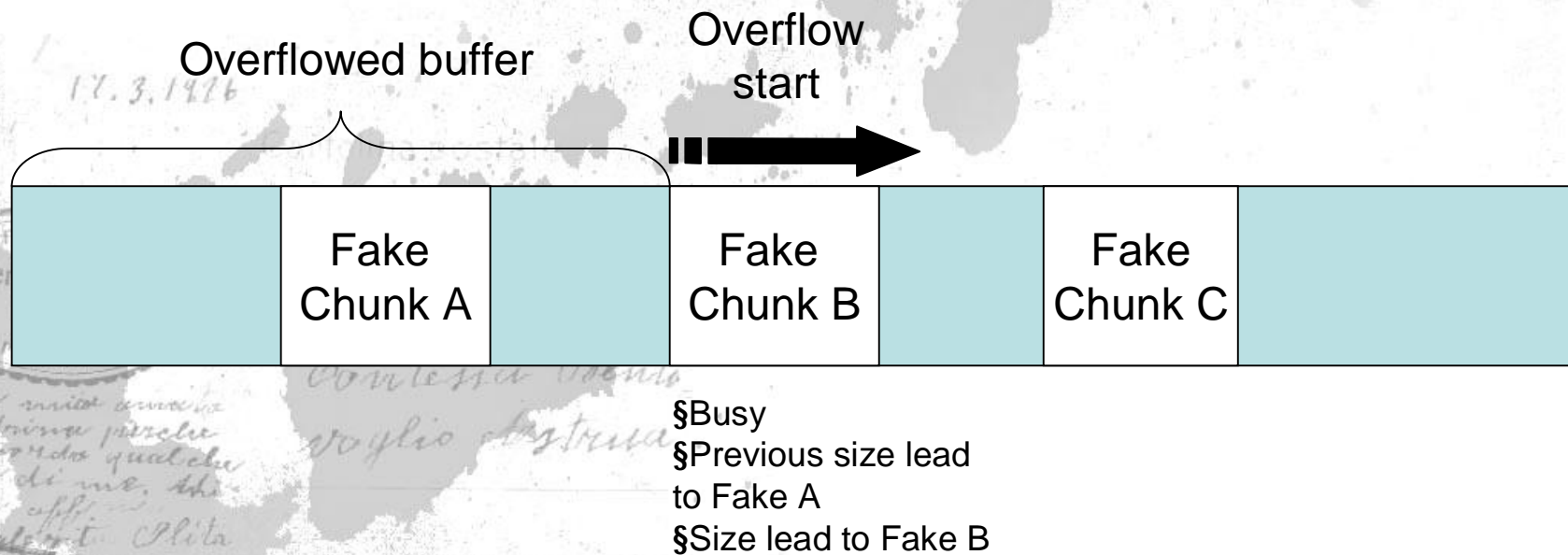
- 构造一个**busy**状态的虚拟内存所带来的其它影响
- 如果堆块被释放后，可能会被系统再次分配，这时堆就会忽略**busy**标志位 (这一点在其它方面同样重要)
- 如果堆块本身的大小没有被正确设置并且**free**链表上的入口地址指定错误的堆就会被破坏。这种情况下系统就会创建一个新的堆块覆盖了正常的堆块
- 应用程序对堆块的正常使用会破坏随机的堆块头

- Coalesce-On-Free 4-byte Overwrite
- 利用堆的合并特性
- 当前没有人（已知的）使用该技术
- 内存覆盖要么发生在被溢出buffer被释放时（正确）要么发生在我们构造的buffer释放之后（错误）
- 构造假的堆块



- Coalesce-On-Free 4-byte Overwrite
- 该方法的优势
  - 内存覆盖比较可靠
  - 如果堆块状态被设置为busy,因为RtlFreeHeap会检查堆块的标志位如果状态为busy就会返回error而不会导致崩溃
  - 在内存地址中可以包含一个NULL字节
  - 甚至溢出buffer大小为0的情况下仍然可以使用该技术
- 该方法的劣势
  - 合并操作可能会覆盖其他堆块,除非堆块的大小猜测的非常准确,这可能会导致堆被破坏
  - 构造堆块的下一个堆块可能首先被释放可能造成堆被破坏

- Coalesce-On-Free Double Overwrite
- 溢出用构造的堆块B覆盖了一个真实的堆块
- 当与被溢出堆块相邻的堆块释放时会产生一个任意地址覆盖(与VirtualAlloc 4字节覆盖技术相同)



- Coalesce-On-Free Double Overwrite
- 该方法优势
  - 可以在一次溢出中覆盖两个任意地址
  - 在内存地址中可以包含一个NULL字节
- 该方法劣势
  - 假定下一个堆块是busy
  - 取决于被溢出buffer的大小
  - 很有可能覆盖应用程序的数据(Fake C)
  - 如果下一个堆块原始状态不是busy,可能导致与Halvar方法相同的其它影响

- 目前为止

Address A	Address B	Comments
Unhandled exception filter	Call [esi+xx] 或类似	成功率高但是与SP版本相关
Vector Exception Handling	堆栈地址指向我们的buffer	成功率高但是与SP版本相关
PEB Locks	猜测地址或者应用特定地址	中等成功率但与SP版本无关 需要猜测地址

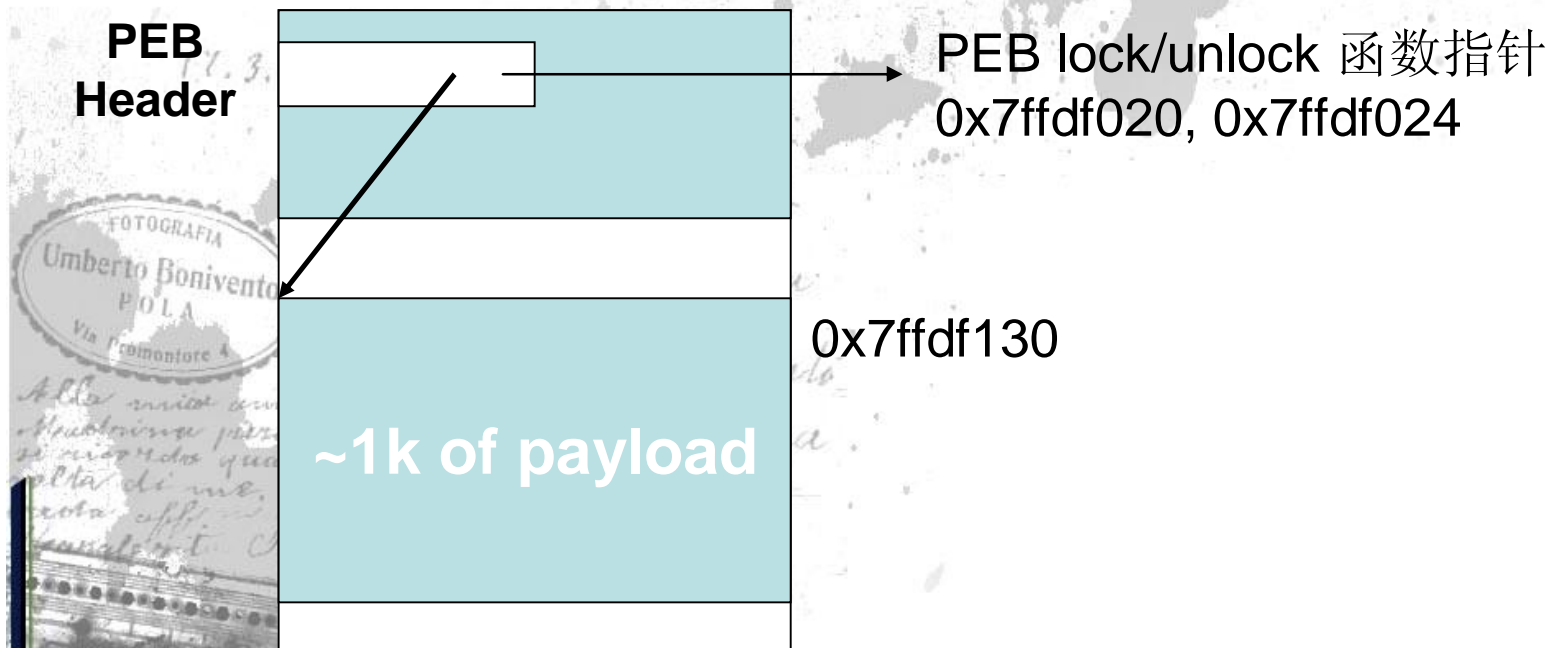
我们可以改进吗?

- Lookaside List Head Overwrite
- 从前面的堆内部结构分析可以知道要进行分配，释放的操作首先使用Lookaside表
- Lookaside表初始化为空
- 默认情况下Lookaside表位置相对于堆是固定的
- 因此 ...
  1. 如果我们申请分配内存就会得到小于1024字节大小的堆块
  2. 应用程序会把它释放给Lookaside表
  3. 因为我们知道Lookaside表的位置..
  4. 我们现在知道了一个指向我们buffer的内存地址!

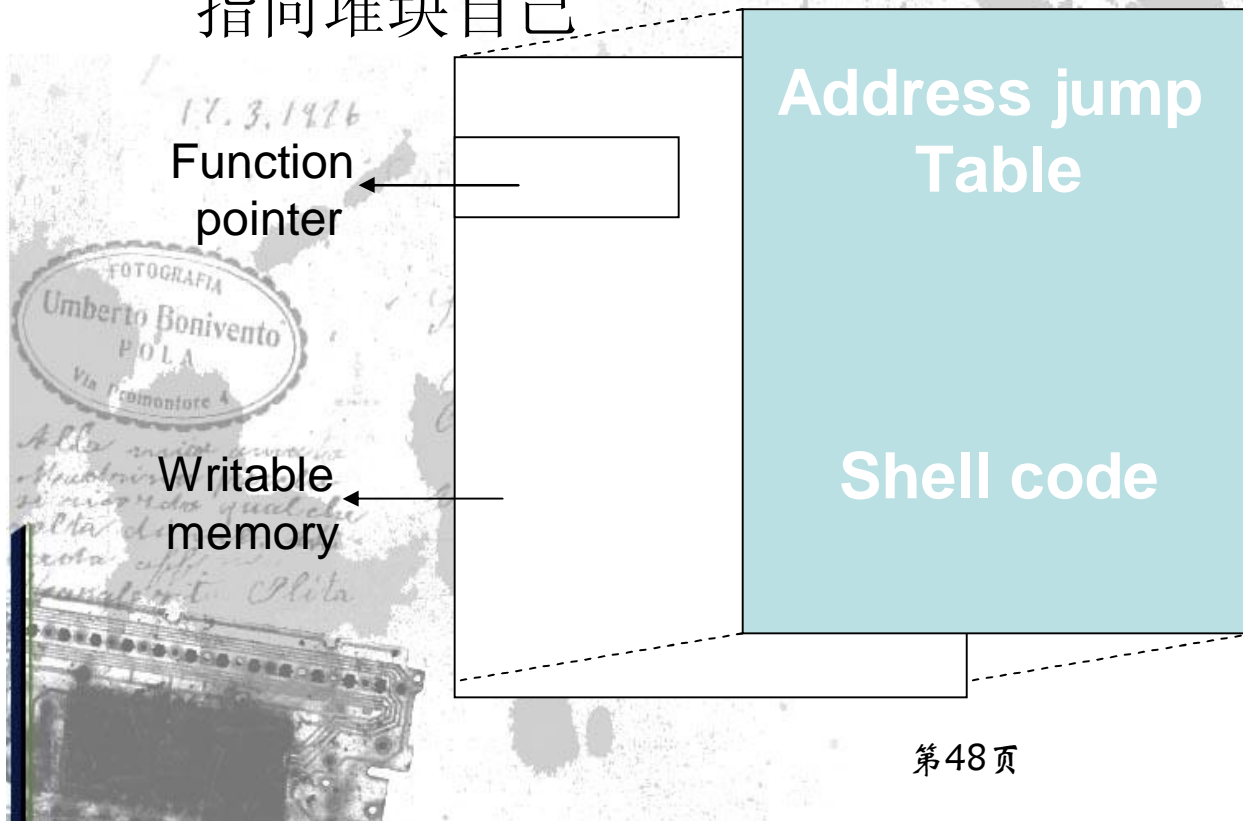
- Lookaside List Head Overwrite
  - 我们需要两个参数来找到Lookaside的入口地址
  - 堆的基地址 – 堆的基地址与SP不相关，但与平台相关.
  - 分配大小 – 因为是我们选择的大小所以我们可以控制这个大小值
- 
- Lookaside Table = Heap base + 0x688
  - Index = Adjusted(allocation size) / 8
  - Lookaside entry location =  
• Lookaside Table + Index \* Entry size (0x30)
- 
- 例子: 如果堆的基地址是0x70000, 分配大小为922
  - Index = Adjust(922) / 8 è 936 / 8 è 0x75
  - 入口地址 = 0x70688+0x75\*0x30 == 0x71c78

- Lookaside Overwrite, 4-to-n-byte Overwrite ( $n \leq \sim 1k$ )
- 在设置好Lookaside表项后，如果我们申请同样大小的堆块时系统就会返回相同大小的堆块
- 我们使用任意内存覆盖技术修改Lookaside上存储的值。
- 结果: 当下一次我们申请相同大小的堆块时，堆就会返回我们在修改的地址的值，这样我们就有了最大到1k的任意内存覆盖技术!

- Uses of 4-to-n-byte Overwrite (Application A)
- 首先把我们所有的shellcode拷贝到一个已知的地址
- 然后把PEB lock函数重定向到这个地址. 这种方法需要两次内存覆盖所有可能有些不稳定



- Uses of 4-to-n-byte Overwrite (Application B)
- 选择一段有函数指针的内存区域然后把我们可以控制的最大到1k的堆块覆盖过去. 因为我们可以准确的得到该地址, 我们可以在这个堆块里面构造一个“address table” 并且指向堆块自己



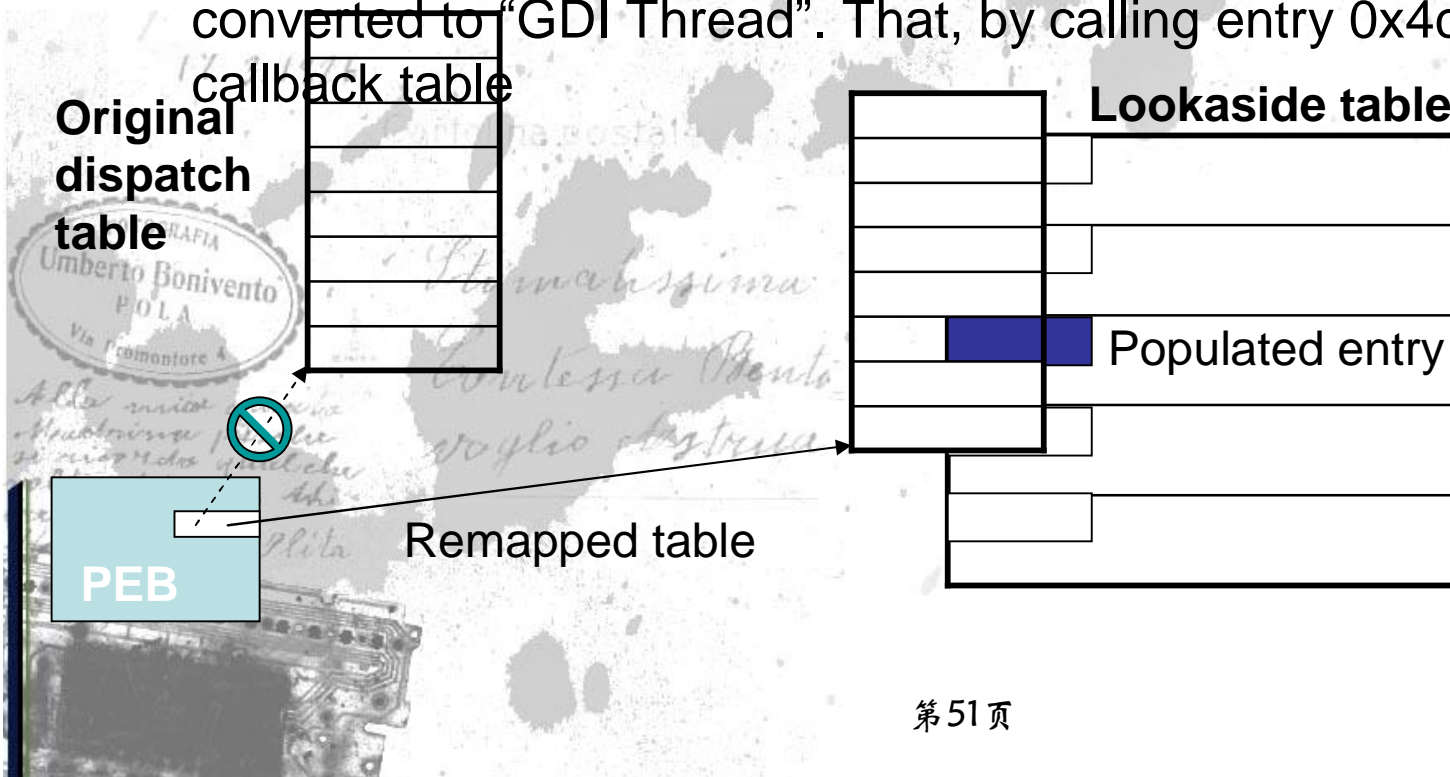
- Uses of 4-to-n-byte Overwrite (Application C)
- 可以寻找应用程序中可写的一些字符串，可以是路径或者是命令，用我们指定的字符串替换这些地址或者命令
- 
- David Litchfield 使用了一种了方法可以作为例子：修改“GetSystemDirectory”例程所使用的字符串，修改路径可以加载攻击者所提供DLL而不用直接执行代码



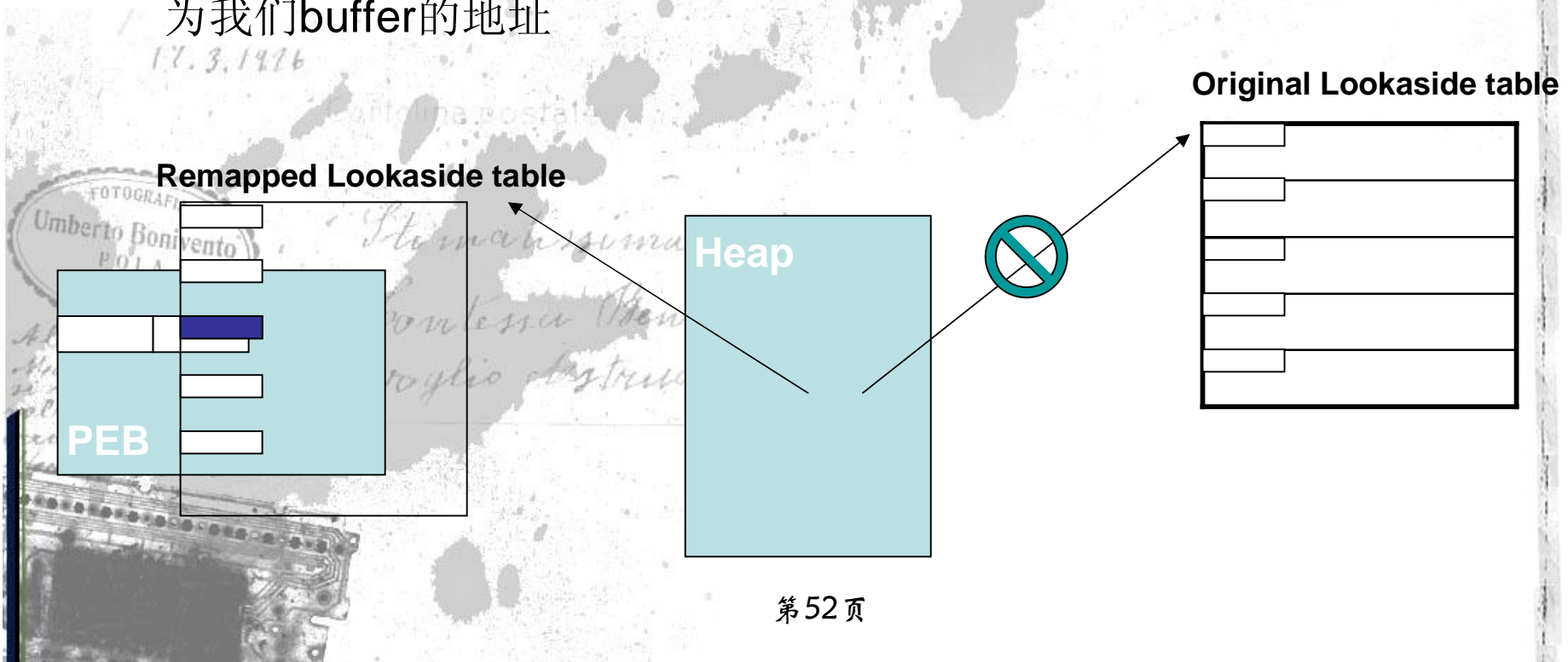
- c:\winnt\system32\  
\\1.2.3.4\backdoors\

- 重映射 Dispatch表
- 如果不用修改Lookaside 项入口地址覆盖最大到 1k任意内存的方法我们还可以重定向其它指针到这个已知的地址
- Dispatch表可以作为一个很好替代方法. 因为 Dispatch表中的每个指针指向一个函数, 如果我们可以在重映射dispatch表以覆盖 Lookaside表并指出使用dispatch 表中的那个一个项, 我们就可以构造出正确的表项指向我们的buffer
- 幸运的是我们有一个例子

- 重映射 Dispatch表(GUI 应用程序)
- PEB中包含了一个dispatch表用作“callback”列程. 该表用作与内核中的GDI组件协同工作
- 因为指向该表的指针保存在PEB中所以该地址是通用地址
- 当一个线程When a thread does the first GDI operation it is being converted to “GDI Thread”. That, by calling entry 0x4c (for XP) in the callback table



- 重映射Lookaside表
- 虽然Lookaside表的默认地址是对基地址加上0x688,但是堆仍是使用一个指针索引Lookaside表
- 我们可以修改这个指针并覆盖一个函数指针
- 当我们完成以上工作后只需要分配适当的大小,指针就会自动被赋值为我们buffer的地址

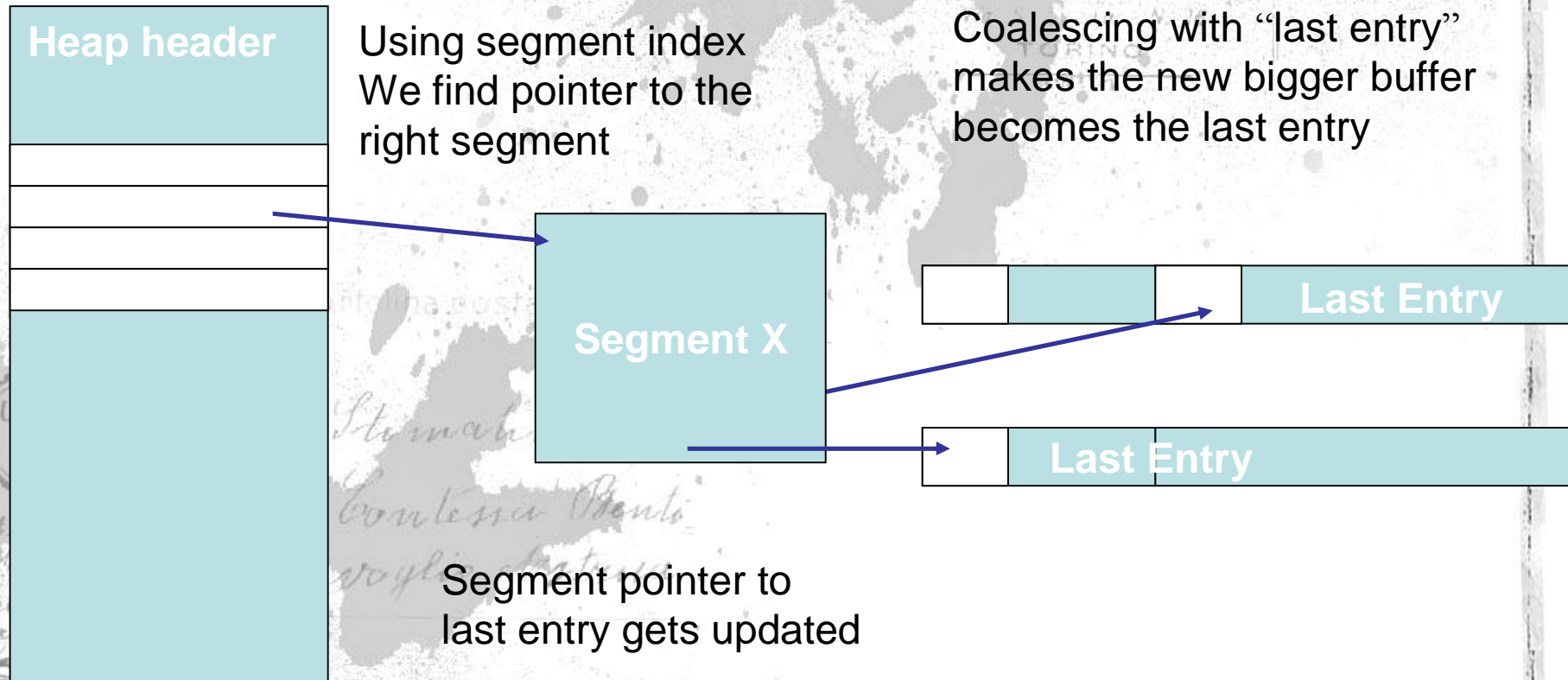


- 重映射Lookaside表
- 重映射Lookaside表的局限性
- 0可以作为Lookaside表的空值，如果Lookaside表被映射到非零区域，我们要注意注意的是alloc()可能会返回任意值。
- 只有当Lookaside的深度小于最大深度时，堆块才会被释放到Lookaside表(i.e. short value at offset 4 should be smaller than short value in offset 8)
- 被堆覆盖的地址就好像Lookaside表项被压栈到Lookaside这个栈里面,这就意味着会覆盖你的buffer前面4个字节，因此如果这4个字节导致非法指令，就不能使用该方法

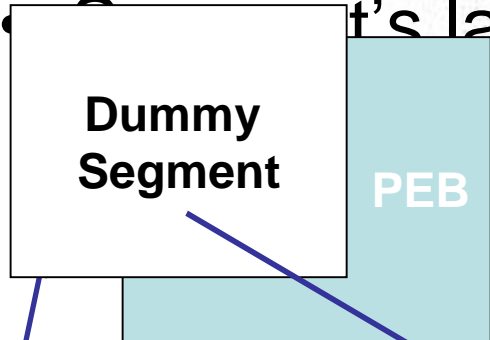
- 段覆盖(AddressOfSelf, 两次覆盖)
- 堆中的每一个段都保存了一个指向“上一个堆块”的指针。每一次段扩展“上一个堆块”的地址都会被改变。
- 当一个堆块被释放时它可能会与上一个堆块进行融合。在这种条件下段会更新它的指针指向为上一个堆块
- 我们可以使用这种方法用指向我们buffer的指针来覆盖任意内存地址

- Segment Overwrite (AddressOfSelf, Double Overwrite)
- 融合算法:
  1. 如果被融合的堆块“Last entry”标志位被置位
    1. 通过堆块头中的段索引找到段
    2. 用新的被融合的堆块地址更新段中上一个堆块的地址
- 融合我们构造的堆块时覆盖任意内存地址后会进行以上操作
- 因此，我们可以改写堆结构中的段指针然后当堆更新时就会把我们堆块的地址覆盖任意指针

- Segment's last entry update (normal operation)

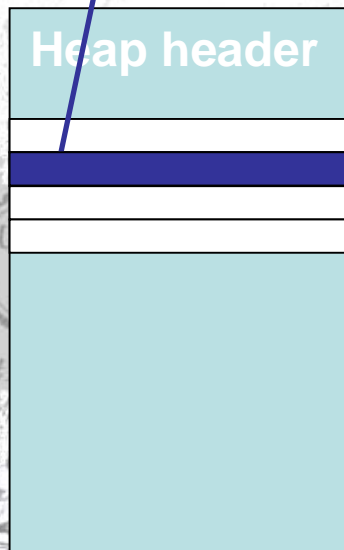


### Segment's last entry update (under attack)



Segment pointer to last entry gets updated. Since the segment overlaps the PEB, the PEB lock function will automatically point to our coalesced buffer

Coalescing with "last entry" makes the new bigger buffer becomes the last entry  
This time, our fake header will Cause arbitrary memory overwrite



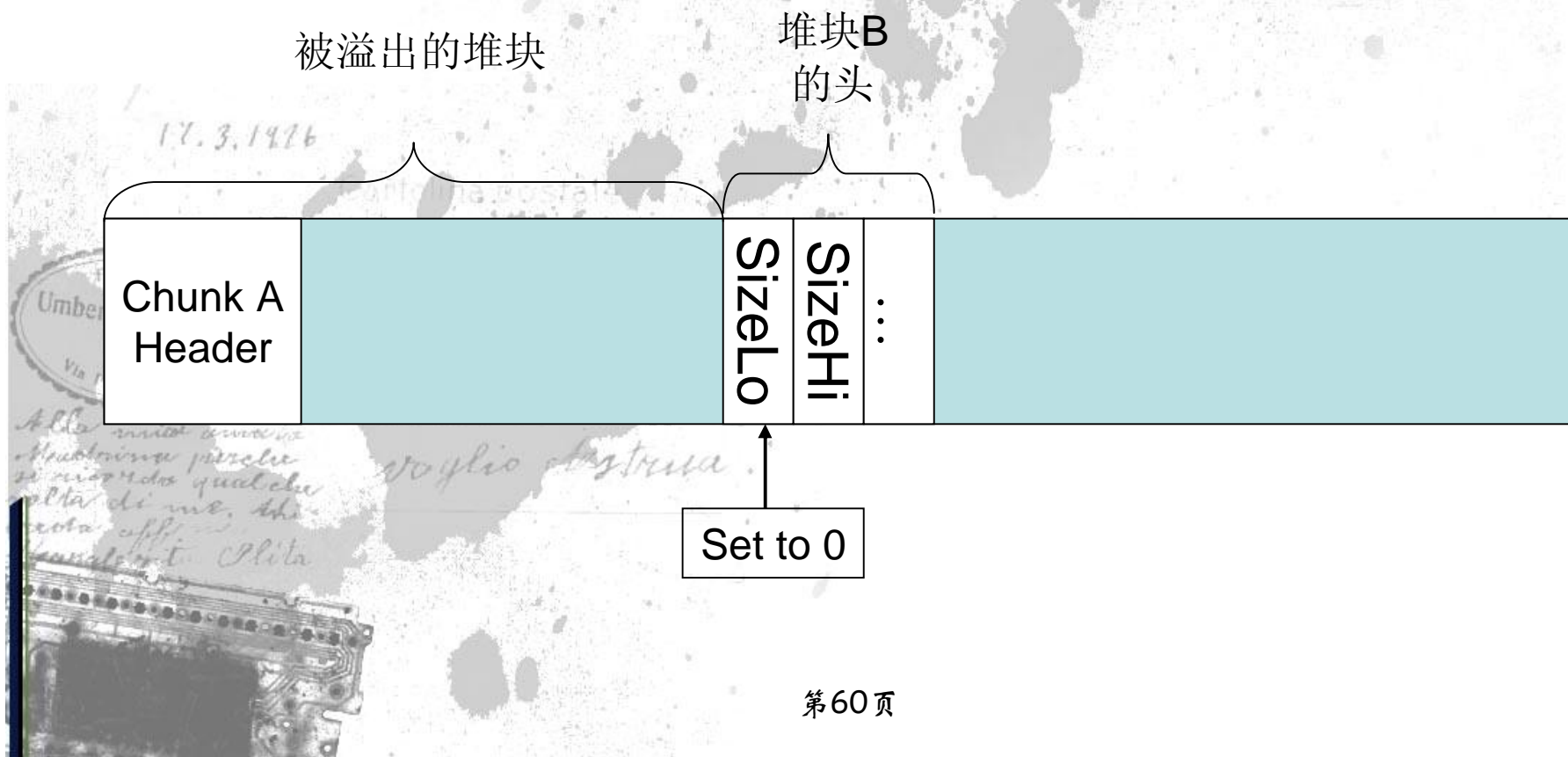
Using segment index We find pointer to the right segment



- Remapping Cache (AddressOfSelf)
- Cache位于堆偏移量0x170上
- cache偏移量0x2c为大于1k的缓存堆块的数组
- Cache通常为NULL
- 当堆块大于1k时对于段覆盖的结果一样
- 比段覆盖更小的破坏性因为不会影响小于1k的堆块

- Remapping Cache (AddressOfSelf)
- 用  $SEH - (ChunkSize - 0x80) * 4 - 0x2c$  覆盖 Cache 指针
- **ChunkSize** 为你控制的堆块的大小（必须大约 1k）
- 当这个堆块被释放时，指针将会被写入 **SEH**

- 单字节溢出
- 单字节溢出的含义为覆盖了下一个堆块头的的最字节为NULL



- 覆盖相邻堆块的堆块大小的最后一个字节
- 只有当堆块大于2k时才能被利用
- 否则，堆块的大小会变为0就不能被利用了(有很多原因)
- 如果ChunkSize > 2K就会把下一个堆块的起始地址提前到上一个堆块中

- Before Off-By-One

真实大小

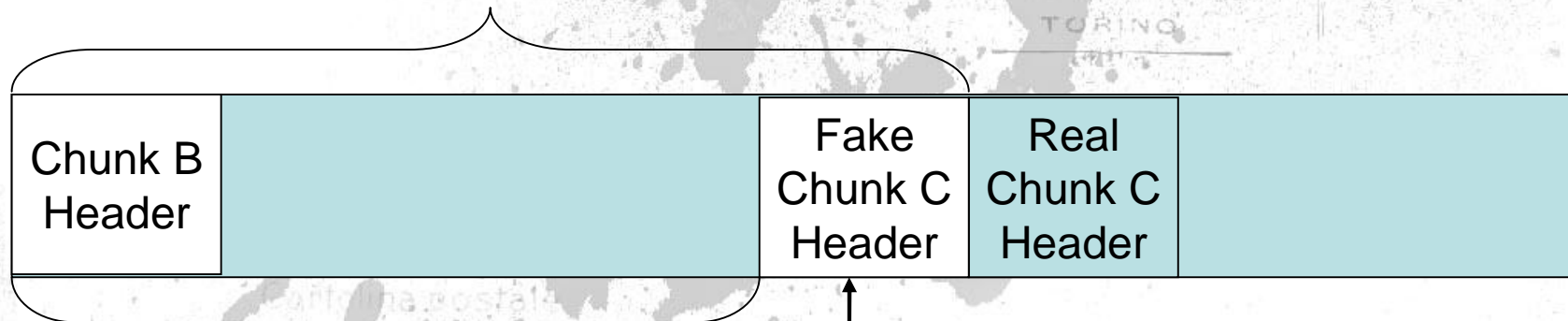
$$0x0110 * 8 = 2176 \text{ bytes}$$

Chunk B Header		Chunk C Header	
-------------------	--	-------------------	--

- After Off-By-One

真实大小

$$0x0110 * 8 = 2176 \text{ bytes}$$



New Size

$$0x0100 * 8 = 2048 \text{ bytes}$$

用户可以控制的  
(Chunk B的一部分)

- 这就意味着你必须控制两个连续的堆块A和B, B的大小大于2k字节。 One must:
- 单字节溢出发生在堆块A中
- 堆块B的大小会被减小
- 把伪造的堆块C头放到堆块B中(放到堆块C的起始位置)
- 使用融合技术修改堆块C的头
- 当堆块B被释放时, 就会产生4字节覆盖

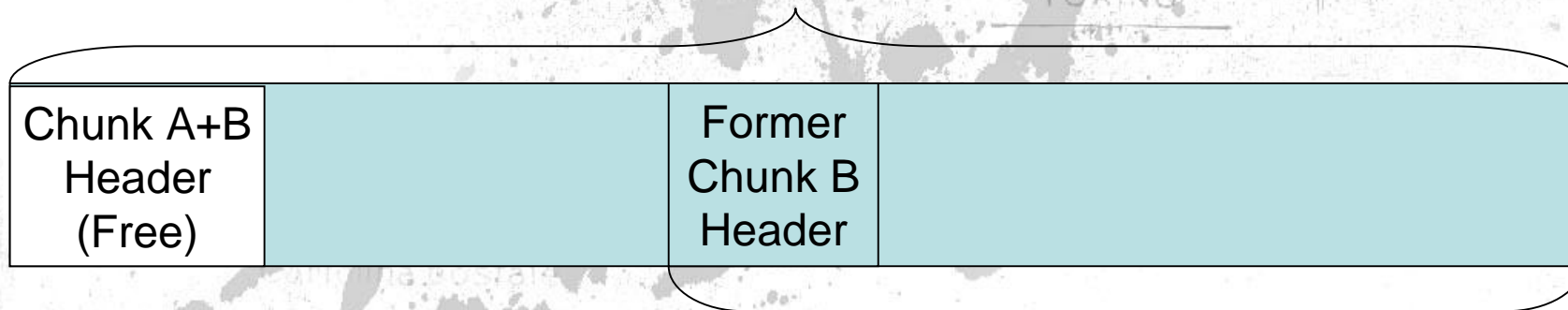


- 堆块A和B (B被两次释放)

Chunk A Header (Free)		Chunk B Header (Busy)	
-----------------------------	--	-----------------------------	--

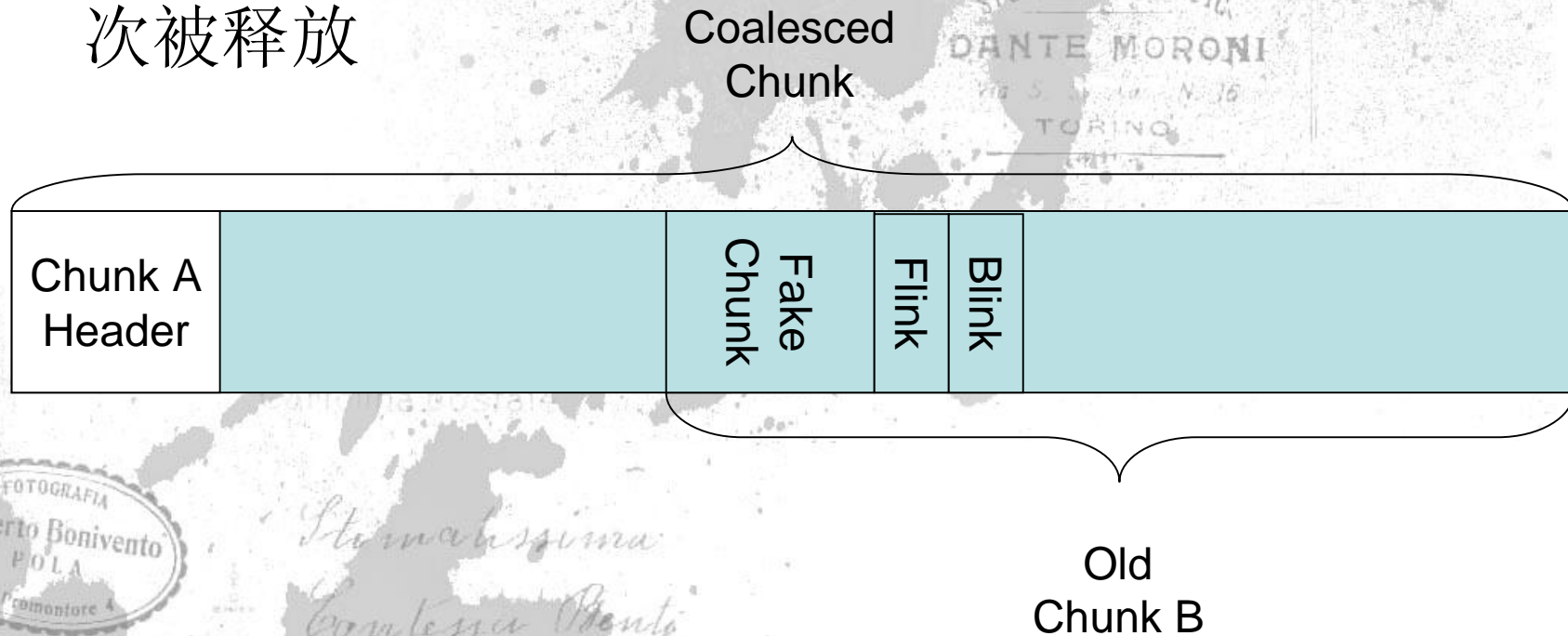
- 堆块A和B (B被释放后)

融合的堆块



以前的堆块 B

- 用户分配堆块A，设置好伪造的头，等待堆块B再次被释放



- Stabilizing execution environment
- 为了覆盖任意内存地址，堆很有可能被破坏。为了使shellcode正确执行，我们必须修复堆。
- 除了对可能被破坏，我们还覆盖了PEB的lock例程。所以我们必须重置这个指针要不然我们的shellcode 会被一次又一次的执行。
- 当堆和lock例程的问题被修复后，我们就可以运行我们正常的shellcode

- 修复被破坏的堆
  - 把Cache指针设置为NULL Set Cache pointer to NULL so FreeList[0] is used
  - Clearing the heap “Free lists” (Litchfield’s method). This approach will allow us to keep the heap in place and hopefully get rid of the problematic chunks by clearing any reference to them
  - Replace the heap with a new heap. If the vulnerable heap is the process default heap, update the default heap field in the PEB. In addition replace the RtlFreeHeap function with “ret” instruction.  
Note: Some problem may still exist since some modules might still point to the old heap header.
  - Intercept calls to RtlAllocateHeap as well as RtlFreeHeap. Redirect allocate calls with old heap header to alternative heap header, just return when RtlFreeHeap is called

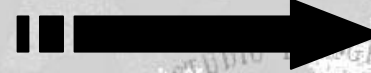
- Windows安全的主要进步
- 增强了“out-of-the-box”安全策略
- 减少了暴露接口的数量。比如：
  - 防火墙缺省打开
  - RPC 不再是缺省走UDP
- 提高了web浏览和e-mail安全
- Windows代码第一次尝试对溢出利用开发设置障碍 (微软所说的“隔离”(Isolation)和“弹回”(Resiliency))

- 堆(heap)进行了特别的安全改进
- XP sp2 在地址上包含了多个变化
- PEB 随机化 (注意: 不是堆随机化!)
- 在堆块头设置安全cookies
- 从双向链表安全摘除(Safe unlink )

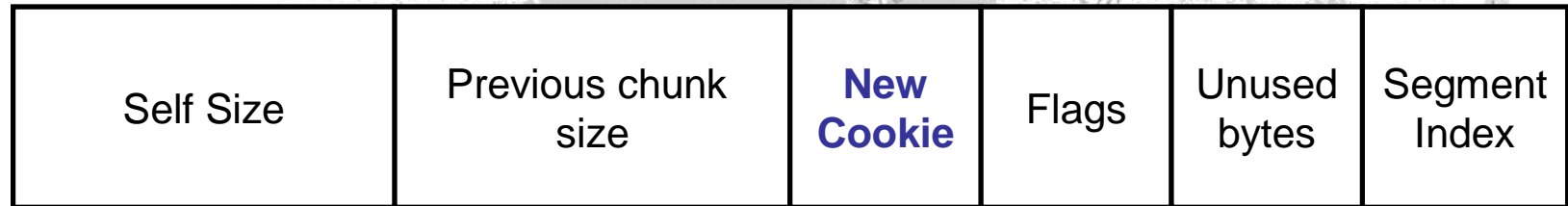
- PEB 随机化
- 在XP SP2 前PEB总是在用户模式地址空间的最后。典型的地址是 0x7ffdf000(这个地址在3GB配置的时候可以改变)。
- 从 XP SP2开始 PEB 的位置不在固定。
- 在早期测试 XP SP2 rc1的时候 PEB 所在的位置接近老的地址，但是可能偏移了几页。
- 比如: 0x7ffd000, 0x7ffd8000 等。

- 堆头cookie

溢出方向



XP SP2  
头



普通头



0

1

2

3

4

5

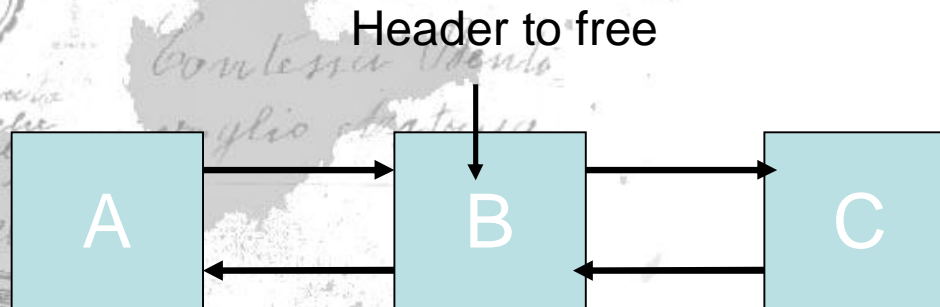
6

7

8

- 堆头 cookie 的计算
- 堆头 cookie 使用下面公式计算：
- $\text{Cookie} = (\&\text{Heap\_Header} / 8) \text{ XOR Heap} \rightarrow \text{Cookie}$
- 堆的地址将决定 cookie. 换句话说，为了知道 cookie 的值，你需要知道溢出的堆头的地址。显然，我们不能轻易的猜测到这个值，否则我们已经在哪儿介绍过的所有方法将没有用。
- 另外，cookie 只有一个字节，所以只有 256 个可能值

- 安全摘除
- 摘除操作是为了把目标从双向链表中去掉
- 在下面的例子中，**B**将从链表中摘除，**C**现在将向后指向**A**，而**A**将向前指向**C**。
- XP SP2 堆将确保在摘除操作的时候下面的状态是 true
- $\text{Entry} \rightarrow \text{Flink} \rightarrow \text{Blink} == \text{Entry} \rightarrow \text{Blink} \rightarrow \text{Flink} == \text{Entry}$



- 现在看起来改写任意4个字节将不再可能。
- 这些变化不能阻止利用覆盖特定的堆结构来进行攻击。在4字节覆盖技术被发明以前，都在使用这种堆攻击利用技术。
- 必须在XP SP2的变化上进行更多的研究。新的利用技术将可能在接下来的几个月里面发展出来。

- 如果别无选择的使用XP SP2,接下来最好的事情是使堆基址(heap base)随机化。
- 在XP SP2中使用类似于随机化PEB的方法来随机化堆基址(heap base)。
- 改变PE中的NT\_HEADERS 节中的 SizeOfHeapReserve 或者 SizeOfHeapCommit 将改变堆基址( heap base)。这将增加一层抵抗蠕虫的保护。
- 暴力猜测仍然可能
- XP SP2的这些改变很有可能被用在其他产品上

- 4-字节覆盖
  - 能够用任意的32位值覆盖任意的32位地址
- 4-到-n-字节 覆盖
  - 用4字节覆盖从而间接导致n字节覆盖
- 两次4-字节 覆盖：
  - 两次4字节覆盖导致的结果一样
- 地址自身的覆盖：
  - 要被覆盖的4字节地址是你能控制的，用来覆盖的是你能控制的块的地址。

- 利用Free覆盖：
  - 一个4字节覆盖发生在溢出块（溢出的源头）被释放的时候
- 利用Free的两次覆盖：
  - 一个4字节覆盖发生在溢出块后面的块(构造的假块)被释放的时候
- VirtualAlloc 覆盖：
  - 一个4字节覆盖发生在释放一块实际分配的块（a virtually allocated block）的时候

- 链表头 覆盖:
  - 4字节覆盖, 被覆盖的是Lookaside或者FreeList 链表头, 这将导致 4到-n-字节 覆盖
- 段 两次覆盖:
  - 两次 4-字节地址自身的覆盖
- 重新映射Lookaside:
  - 4字节覆盖, 被覆盖的是堆结构的Cache指针
- 重新映射 Cache:
  - 4字节覆盖, 被覆盖的是堆结构的Cache指针