Reliable Windows Heap Exploits

Matt Conover & Oded Horovitz

Agenda

- Introduction to heap exploits
- Windows heap internals
- Arbitrary memory overwrite explained
- Applications for arbitrary memory overwrite + exploitation demos
- Special notes for heap shellcodes
- XP SP2 heap protection
- Q & A

Introduction

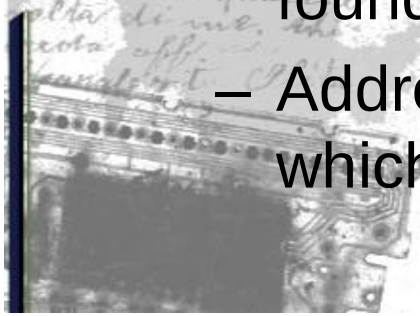
- Heap vulnerabilities become mainstream

  DCOM (seems to be the inflection point), Messenger, MSMQ, Script Engine

- Researchers try to address heap overflows:
  - David Litchfield – “Windows Heap Overflows”
  - LSD – “Microsoft windows RPC security vulnerabilities”
  - Dave Aitel – “Exploiting the MSRPC heap overflow I,II”
  - Halvar – “3rd Generation exploits”

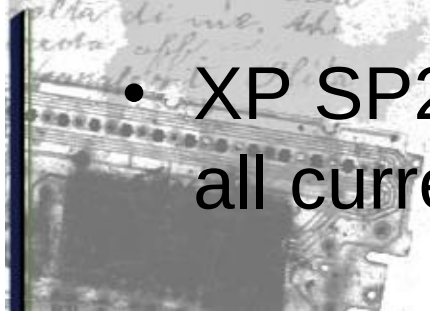
Introduction

- Even experts use some voodoo magic as main ingredient of exploits
  - Making a 4-byte overwrite (discussed more later) is a guess work
  - Failures are not well understood
- Available exploits are service pack dependent
  - Shellcode address is not known,
  - SEH address varies between service packs
  - During exception handling, pointer to buffer can be found on the stack (in exception record)
  - Address of instruction that access the stack is needed, which is SP dependent



Introduction: Our Findings

- Found several techniques at each stage of a heap overflow that greatly improve reliability
- Much greater understanding of the Windows heap internals and its processes.
- Determined why existing techniques are unreliable
- XP SP2 will greatly improve protection and stop all currently used techniques

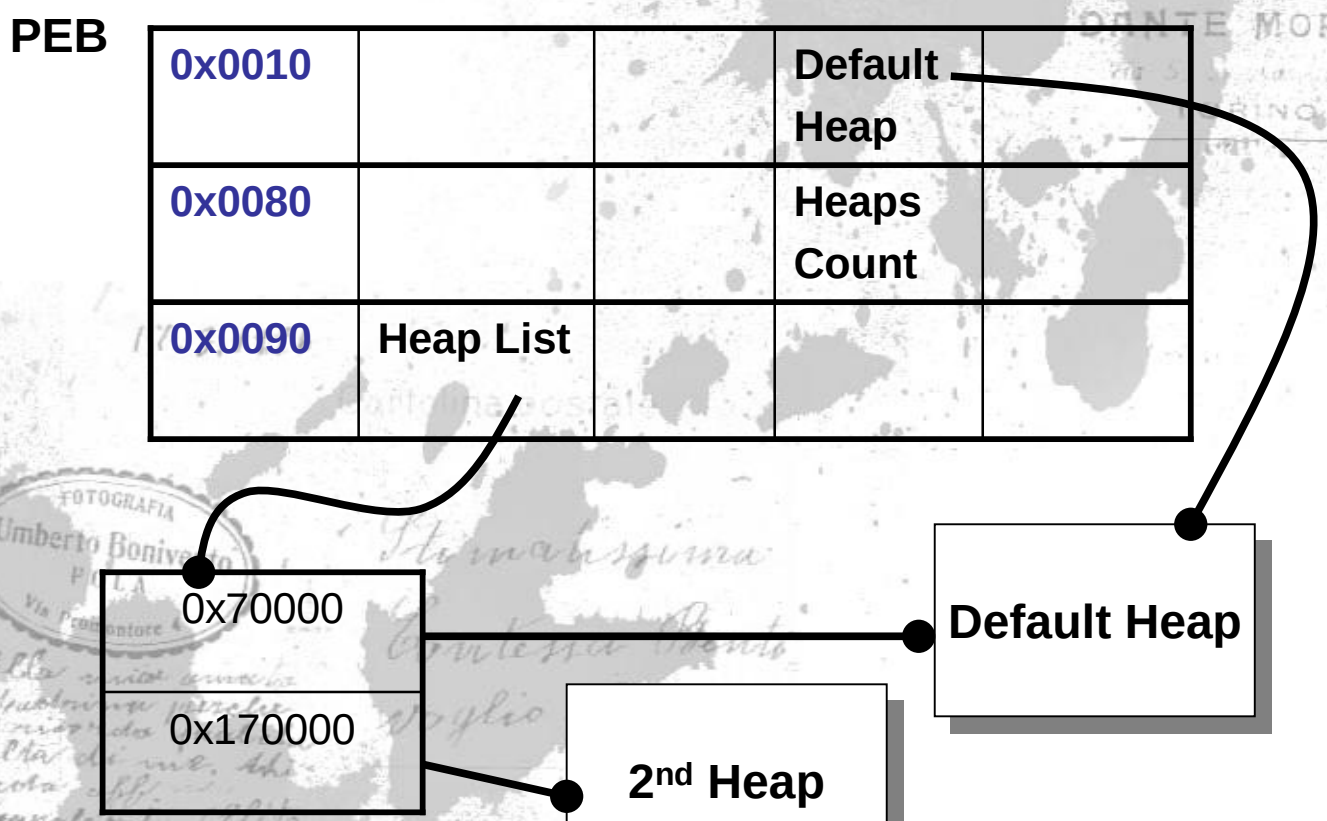


Windows Heap Internals

- What Is Covered
  - Heap internals that can aid in exploitations
    - Heap & process relations
    - The heap main data structures
    - The algorithms for allocate & free
- Not Covered
  - Heap internals that will bore you to death
  - Stuff that is not directly related to exploit reliability
  - Algorithms for “slow” allocation or heap debugging

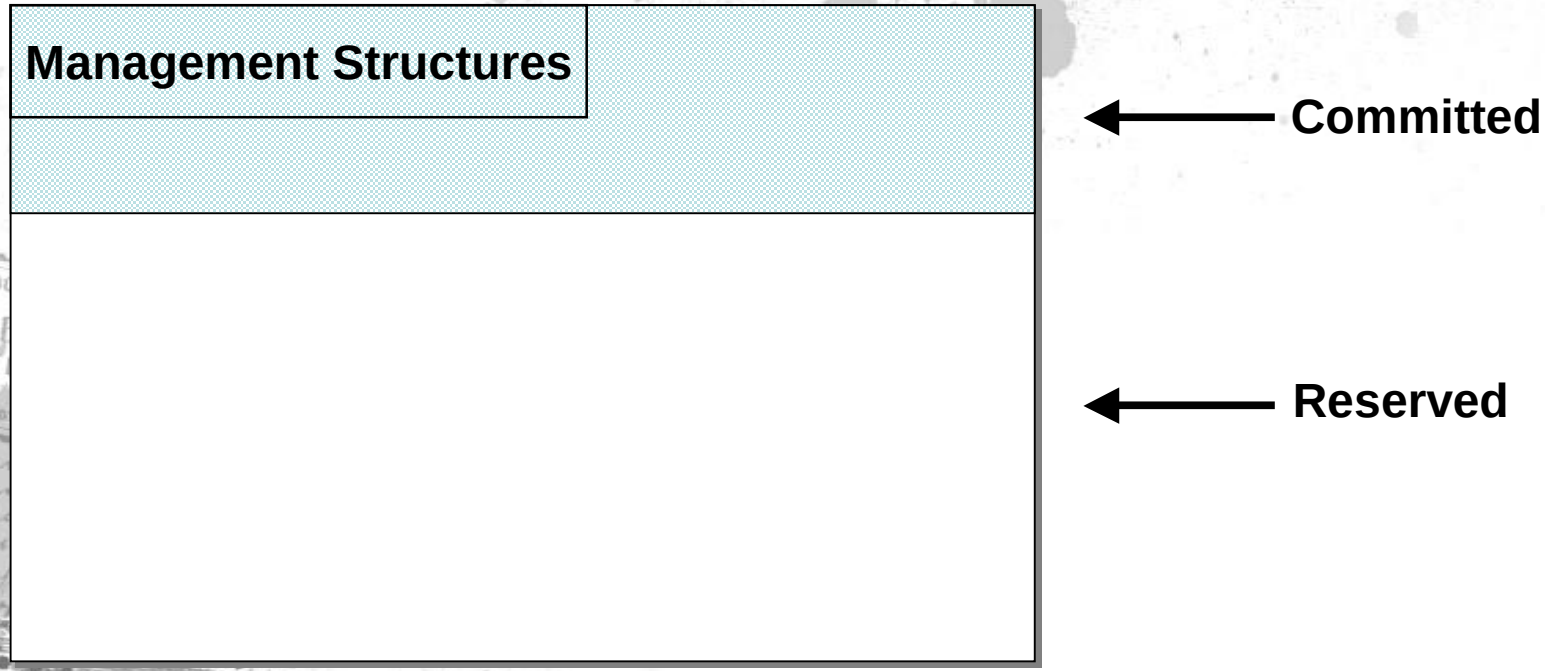
Windows Heap Internals

- Many heaps can coexist in one process



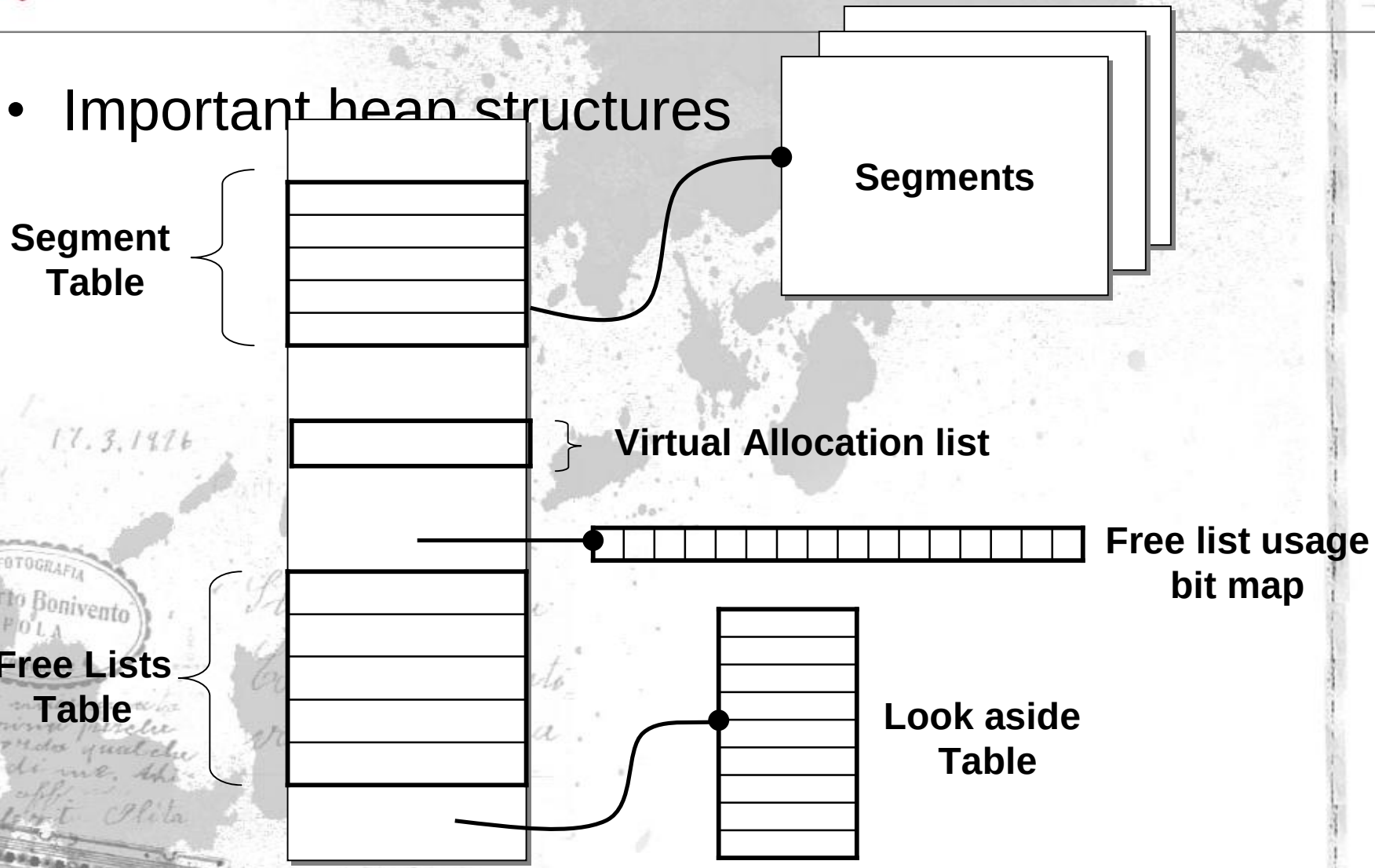
Windows Heap Internals

- Heap starts with one big segment
- Most segment memory is only reserved
- Heap management is allocated from the heap!



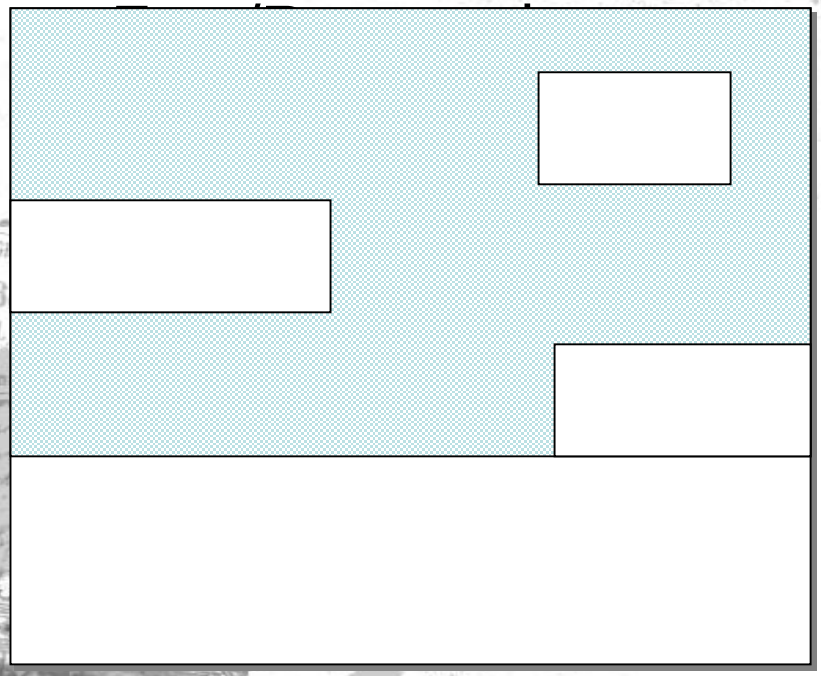
Windows Heap Internals

- Important heap structures



Windows Heap Internals

- Segment management
  - Segment limits (in pages)
  - List of uncommitted blocks

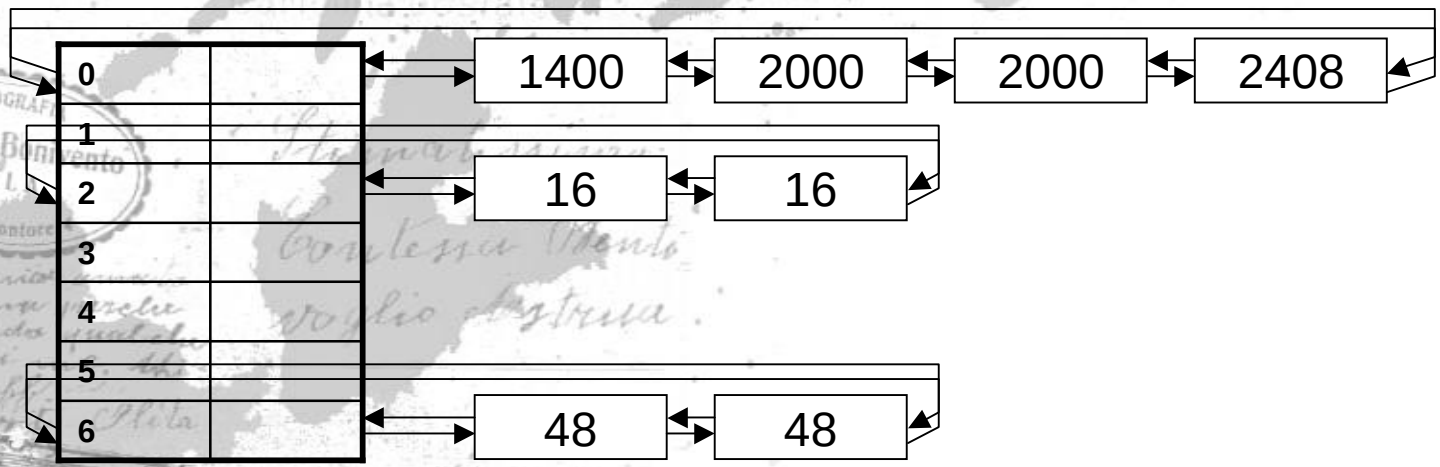


Reserved

Committed

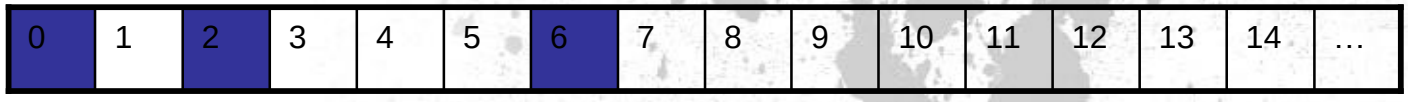
Windows Heap Internals

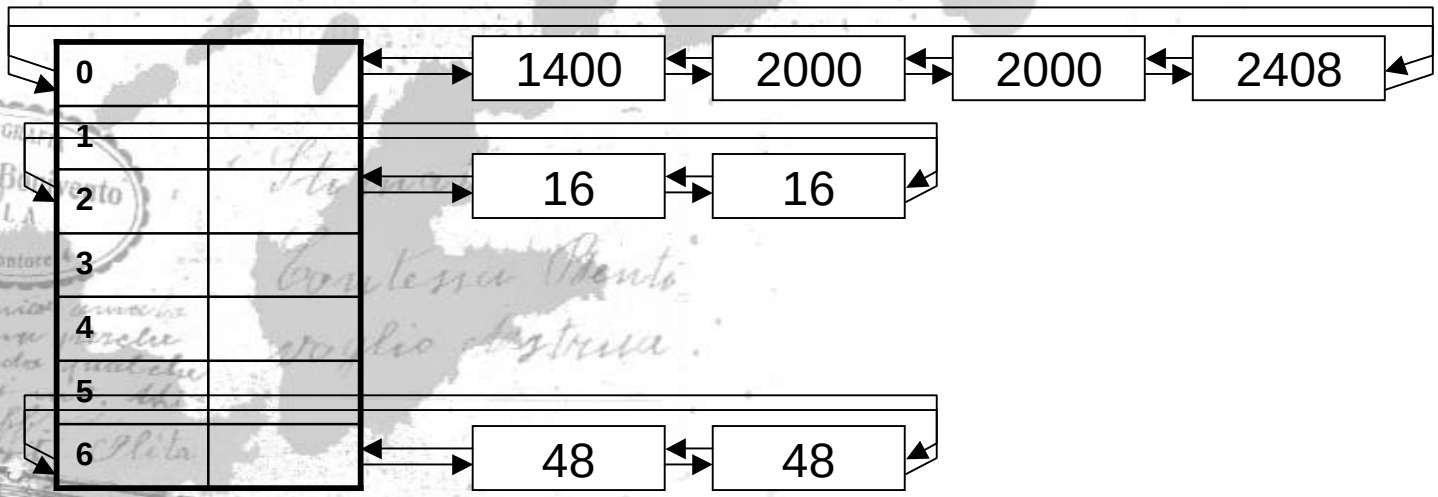
- Free List management
  - 128 doubly-linked list of free chunks
  - Chunk size is table row index * 8 bytes
  - Entry [0] is an exception, contains buffers of 1024 < size < “Virtual allocation threshold”, sorted from small to big



Windows Heap Internals

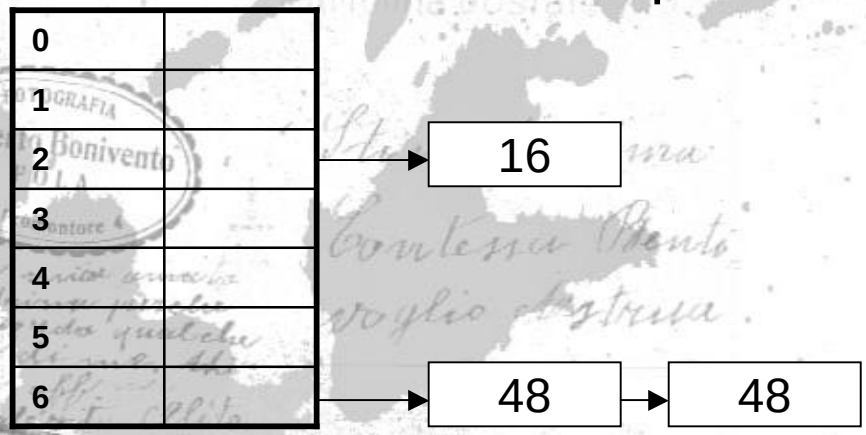
- Free List Usage Bitmap
  - Quick way to search free list table
  - 128 Bits == 4 longs (32 bits each)





Windows Heap Internals

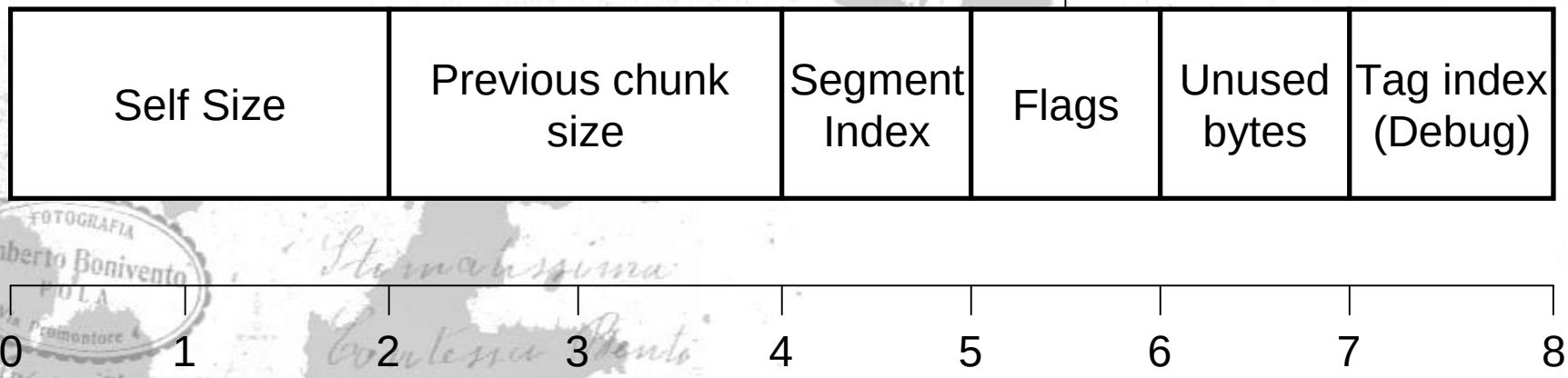
- Lookaside Table
  - Fastest route for free and alloc
  - Starts empty
  - 128 singly-linked lists of busy chunks (free but left busy)
  - Self balanced depth to optimize performance



Windows Heap Internals

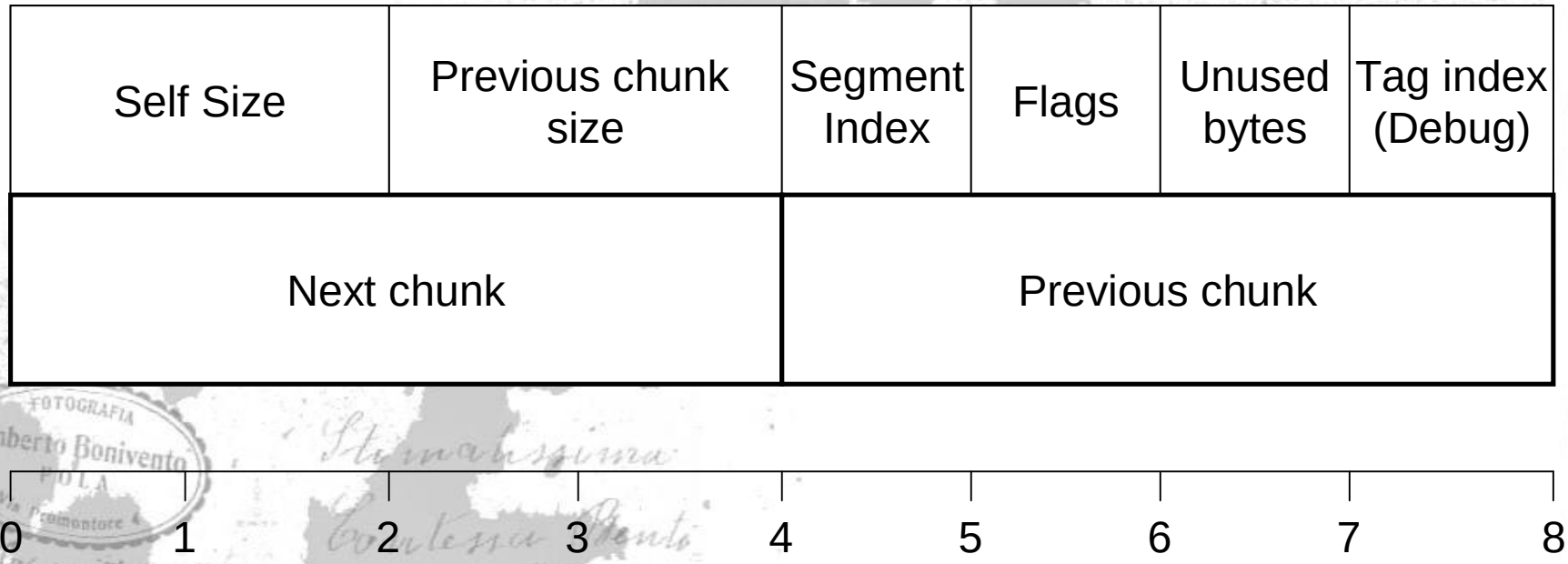
- Basic chunk structure – 8 Bytes

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

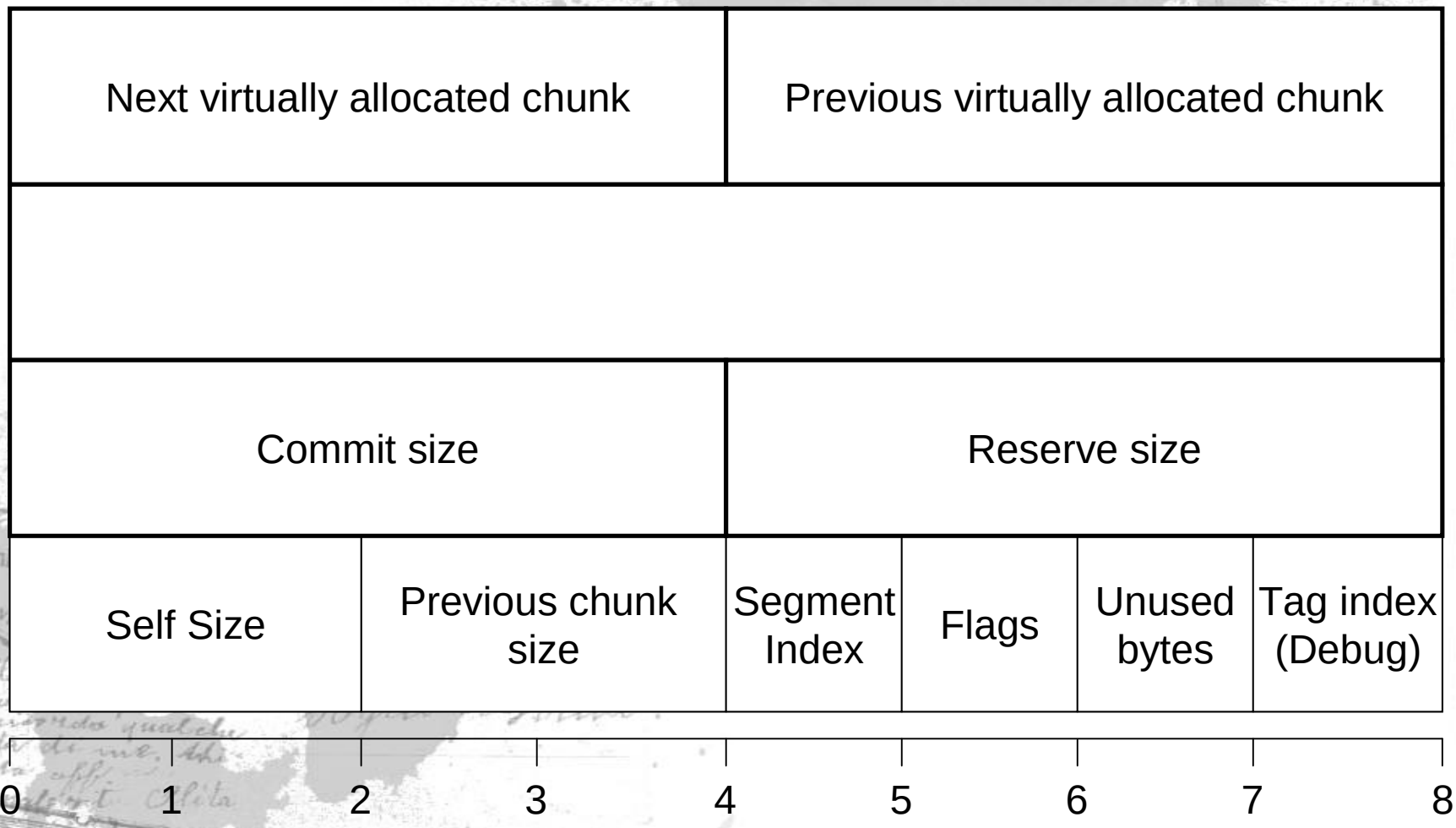


Windows Heap Internals

- Free chunk structure – 16 Bytes



Windows Heap Internals



Windows Heap Internals

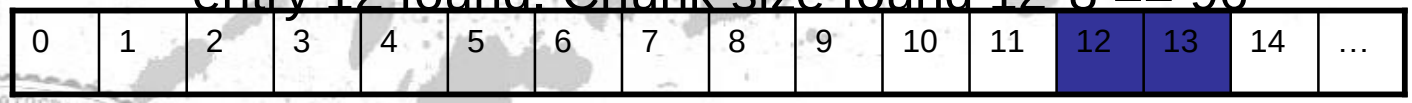
- Allocation algorithm (high level)
- Adjust size. Add 8, and 8 bytes aligned upward
- If size is smaller than virtual alloc threshold {
  - Lookaside
  - Free list
  - Cache
  - Free list [0]
  - If can't find memory, extend heap as needed
- }
- If size needed is >= than virtual alloc threshold
  - Allocate memory from the OS, add the chunk to list of virtually allocated buffer

Windows Heap Internals

- Allocate algorithm – Lookaside search
- Take buffer from Lookaside only if
  - There is a Lookaside table
  - Lookaside is not locked
  - Requested size is smaller than 1024 (to fit the table)
  - There is exact match for requested size (e.g., Lookaside is not empty)
- If lookaside is not empty, remove from Lookaside and return it to the user

Windows Heap Internals

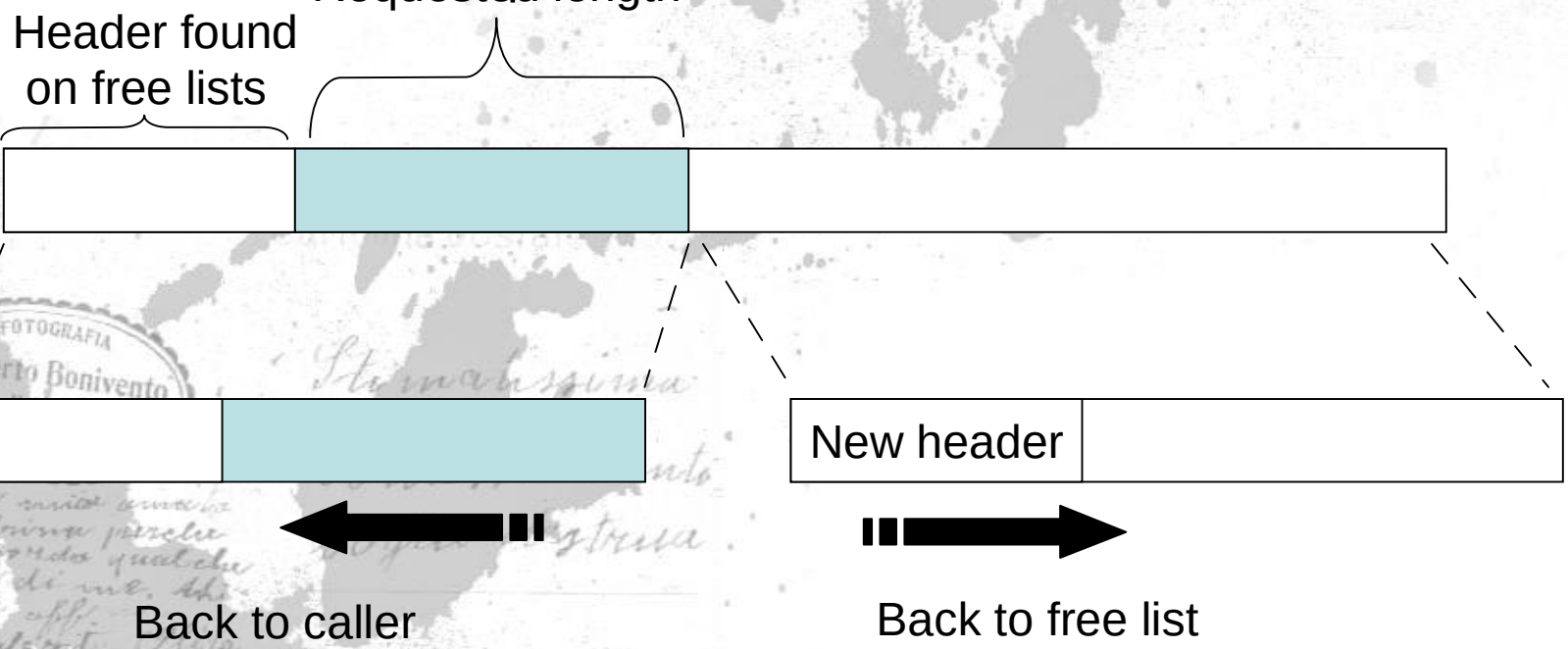
- Allocate algorithm – Free list search
- Search available free list bitmap to find big enough entry
- Example:
- user ask for 64 bytes
- start looking with entry 64/8 == 8
- entry 12 found. Chunk size found 12*8 == 96



- If no entry found in the bit array, return a block from the heap cache or FreeList[0])

Windows Heap Internals

- Allocate algorithm – Free list search
- When chunk is taken from free list, we check its size. If size is bigger than what we need by 16 or more bytes we will split the chunk and return it to the heap

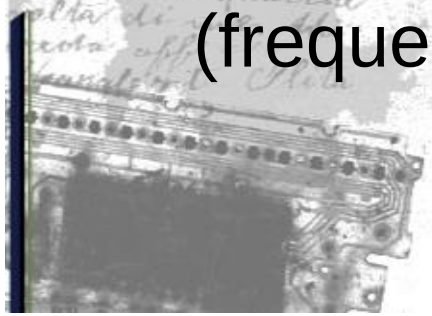


Windows Heap Internals

- Finding a chunk in FreeList[0]
- Used when the cache is not present or empty
- This is usually what happens for chunk sizes > 1K
- FreeList[0] is sorted from smallest to biggest
- Check if FreeList[0]->Blink to see if it is big enough (the biggest block)
- Then return the smallest free entry from free list[0] to fulfill the request, like this:
- While (!BigEnough(Entry->Size, NeededSize))
- Entry = Entry->Flink

Windows Heap Internals

- Heap Cache Internals
- Every time a chunk size > decommit threshold (4K) is freed while the total free bytes in heap are > 64K, it is decommitted and returned to the uncommitted range
- This gets very expensive. For this reason heap caching was added in Windows 2000 SP2
- By default, it is disabled and only created if the program is making short use of big chunks (frequent allocs/frees of chunks >= 4K)



Windows Heap Internals

- Heap Cache Internals
- It's basically like the free lists for Chunks < 1K
- It is a fixed size based on decommit threshold (896 entries)
- Each entry in the CacheTable is a doubly linked of chunks for that specific size (except the last entry in CacheTable)
- If the heap cache is present and not empty, it is used BEFORE FreeList[0].

Windows Heap Internals

- Heap Cache Internals
- CacheTableIndex = ChunkSize – 1K (0 is 1024, 1 is 1032, etc.)
- The last entry in the cache is the equivalent of the old FreeList[0] (sorted list of free chunks)
- So chunks > 8K are put into CacheTable[895]

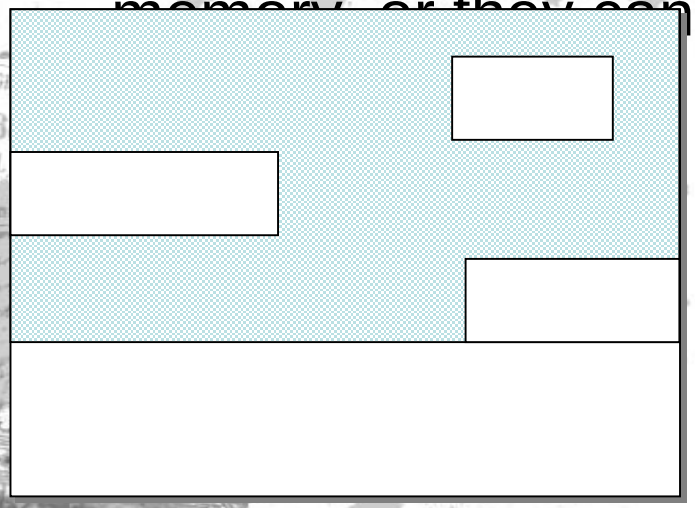
Windows Heap Internals

- Finding a chunk in the Cache
- If (Index != LastEntryInCacheTable and ChunkTable[Index] != NULL) return chunk
- Else If (Index != NumEntries-1) Iterate through ChunkTable[Index] and return the first chunk big enough.
- Else
  - Use CacheTable bitmap to find a bigger entry
  - (This bitmap works just like the free lists bitmap)
  - Return unused portion to free lists

- Heap Exploitation when cache is present…
- If exploiting a program with large chunks, set the Cache to NULL! This ensures FreeLists[0] is used
- Otherwise, Litchfield's heap cleanup trick (discussed BlackHat Windows 2004) which uses FreeList[0] will not work!

Windows Heap Internals

- Allocate algorithm – Heap extension
  - If no chunk can fulfill request and heap is growable, commit more memory from segments reserved memory
  - Reuse “holes” of uncommitted range if possible
  - If existing segments do not have enough reserved memory, or they can not be extended, create a new



Reserved

Committed

Windows Heap Internals

- Allocate algorithm – Virtual Allocate
- Used when ChunkSize > VirtualAlloc threshold (508K)
- Virtual allocate header is placed on the beginning of the buffer
- Buffer is added to busy list of virtually allocated buffers (this is what Halvar's VirtualAlloc overwrite is faking)

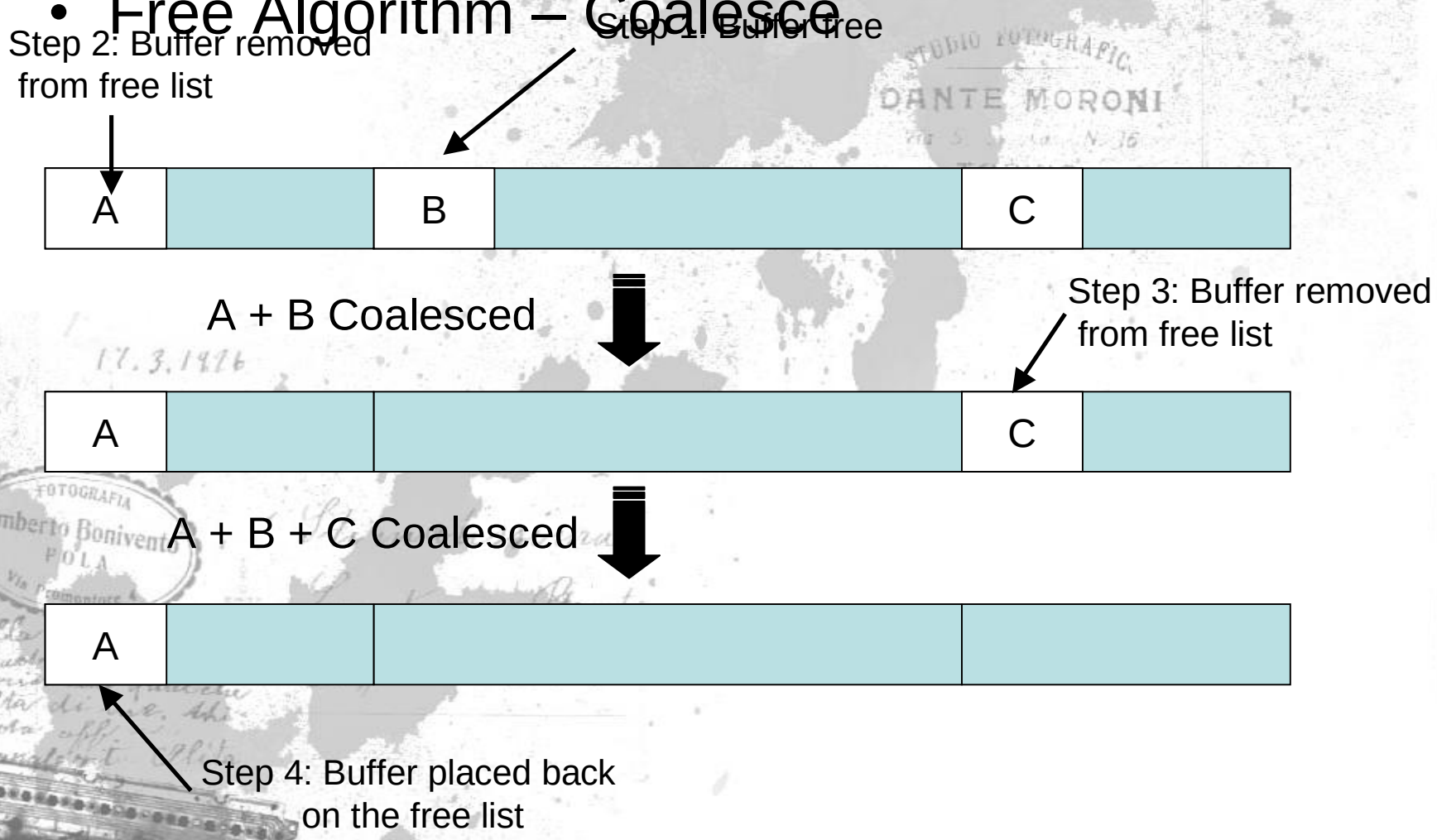
Windows Heap Internals

- Free Algorithm (high level)
- If buffer is free, address is not aligned, or segment index is bigger than “max segments (0x40)” return error;
- If buffer is not a virtually allocated chunk{
- Try to free to Lookaside
- Coalesce buffer & place on free list or cache
- }
- If virtually allocated buffer{
- Remove buffer from busy virtually allocated buffers
- Free buffer back to the OS
- }

Windows Heap Internals

- Free Algorithm – Free to Lookaside
- Free buffer to Lookaside only if
  - There is a Lookaside table
  - Lookaside is not locked
  - Requested size is smaller than 1024 (to fit the table)
  - Lookaside is not “full” yet
- If buffer can be placed on Lookaside, keep the buffer flags set to busy and return to caller.

- Free Algorithm – Coalesce



Windows Heap Internals

- Free Algorithm – Coalesce
- Where coalesce cannot happen
- Chunk to be freed is virtually allocated
- Highest bit in Chunk flags is set
- Chunk to be freed is first **à** no backward coalesce
- Chunk to be freed is last **à** no forward coalesce
- If the chunk to be coalesced with is busy
- The size of the coalesced chunk would be >= 508K

Windows Heap Internals

- Free Algorithm – Put Coalesced Chunk in FreeList
- If block size < 1024, then insert to proper free list entry
- If block size > De-commit threshold and total heap free size is over decommit total free threshold, then decommit buffer back to the OS. NOTE: if this step happens enough times, the heap cache is created
- If chunk is smaller than virtual allocate threshold, insert the block into free list [0]
- If chunk block is bigger than virtually allocate threshold, break the buffer to smaller chunks (each one as big as possible), and place them into the cache (if present) or free list [0].

Windows Heap Internals

- Summary – Questions?
- Main structures – Segments, Lookaside, Free lists, Cache, Free list [0], Virtual alloc list

- Free / alloc algorithm work order
  - Lookaside
  - Free list
  - Cache
  - Free list[0]

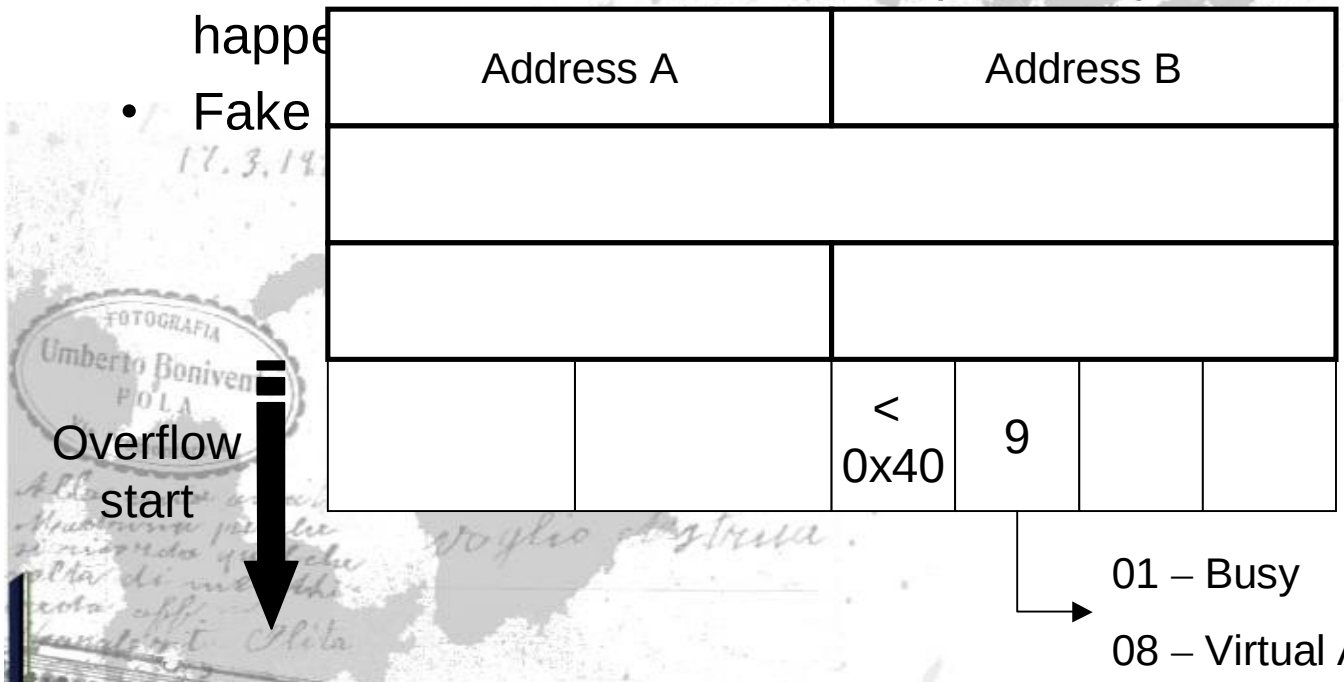
- Heap memory is totally recyclable
  - Big free buffers are divided on allocation
  - Small buffers are coalesced to create bigger buffers

Heap Exploitation: Basic Terms

- 4-byte Overwrite
  - Able to overwrite any arbitrary 32-bit address (WhereTo) with an arbitrary 32-bit value (WithWhat)
- 4-to-n-byte Overwrite
  - Using a 4-byte overwrite to indirectly cause an overwrite of an arbitrary-n bytes
- Double 4-byte Overwrite:
  - Two 4-byte Overwrites result from the same operation (e.g., a single free)
- AddressOfSelf Overwrite:
  - 4-byte overwrite where you set WhereTo, and WithWhat is already the address of a chunk you control

Arbitrary Memory Overwrite Explained

- VirtualAlloc 4-byte Overwrite (Halvar)
- Utilize the virtual allocation headers
- Arbitrary memory overwrite will happen when the buffer AFTER the source overflow chunk is freed (if already freed, this will never happe
- Fake



Arbitrary Memory Overwrite Explained

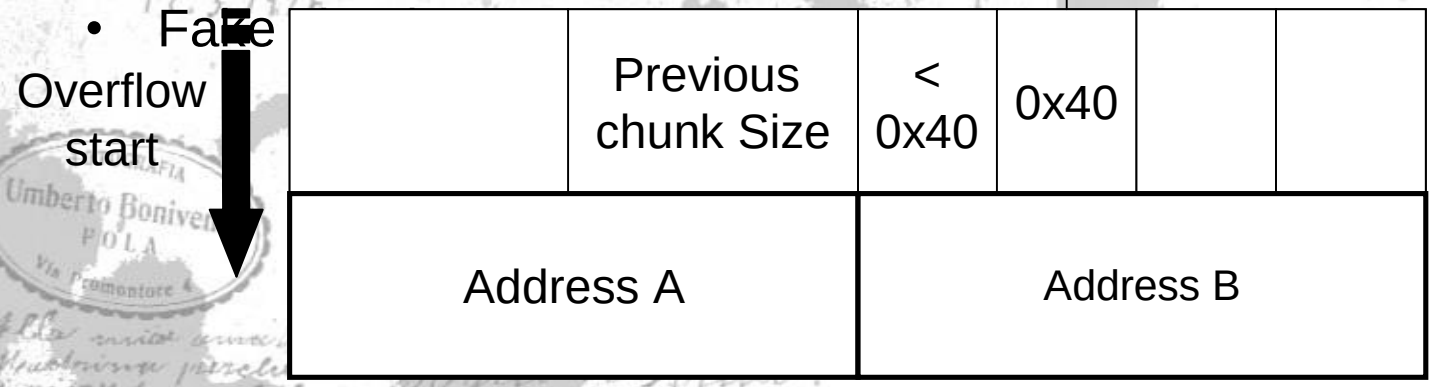
- VirtualAlloc 4-byte Overwrite
- Pros for this method
  - If next buffer is indeed busy arbitrary memory overwrite will happen and will keep heap state (almost) intact
- Cons of this method
  - If overflow involves string operations, you can't use this method to overwrite memory having a NUL byte
  - You need at least 24 bytes of data in overflowed buffer
  - If buffer was not busy, no arbitrary memory overwrite will happen, may cause heap corruption (explained in next slide)

Arbitrary Memory Overwrite Explained

- Side effects of faking a busy virtual allocated buffer
- In case the buffer was originally free it might be later used in an alloc, **the heap will ignore the fake busy flags** (this is important in other cases as well)
- If fake self-size value is not guessed correctly AND free list entry was not exactly the one the user asked for the buffer will get split. In that case the heap will create a new free chunk which overlap legitimate chunks
- Normal usage of the buffer by the application may corrupt random heap headers

Arbitrary Memory Overwrite Explained

- Coalesce-On-Free 4-byte Overwrite
- Utilize coalescing algorithms of the heap
- No one seems to be (knowingly) using this technique yet
- Arbitrary overwrite happens when either the overflowed buffer gets freed (good) or when the buffer AFTER the faked buffer gets freed (bad)
- Fake

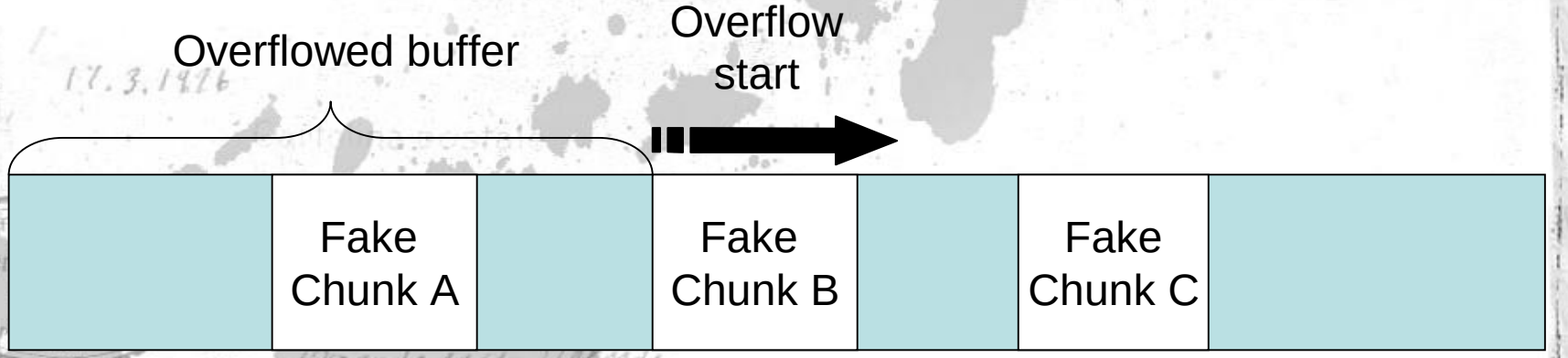


Arbitrary Memory Overwrite Explained

- Coalesce-On-Free 4-byte Overwrite
- Pros for this method
  - Arbitrary memory overwrite will always happen
  - If buffer was busy, RtlFreeHeap will not crash since it checks flags and return with error if heap is busy
  - One NUL byte is allowed in memory address
  - Can be used even when overflowed buffer size is 0
- Cons for this method
  - Unless self-size in fake header is guessed correctly, the coalesced buffer may overlap other chunks. This will most likely lead to heap corruption
  - Chunk after fake chunk may be freed first and will probably lead to heap corruption

Arbitrary Memory Overwrite Explained

- Coalesce-On-Free Double Overwrite
- Overflowed buffer overwrites a real chunk header with Fake Chunk B
- Arbitrary overwrite happens when the buffer next to the overflowed buffer gets freed (same as VirtualAlloc 4-byte Overwrite)



§Busy
§Previous size lead to Fake A
§Size lead to Fake B

Arbitrary Memory Overwrite Explained

- Coalesce-On-Free Double Overwrite
- Pros for this method
  - Provide 2 arbitrary memory overwrite in one overflow
  - One NULL byte is allowed in memory address
- Cons for this method
  - Assume next chunk is busy
  - Depends on overflowed buffer size
  - High likelihood that will corrupt application data (Fake C)
  - If next buffer was not originally busy, will cause same side effects as Halvar's method

Arbitrary Memory Overwrite Explained

- Up to now

Address A	Address B	Comments
Unhandled exception filter	Call [esi+xx] Or similar	High rate of success, but SP dependent
Vector Exception Handling	Stack location pointing to our buffer	High rate of success, but SP dependent
PEB Locks	Guessed address or application specific	Medium rate of success (because of guessed address), SP **Independent**

Can we improve on that?

Arbitrary Memory Overwrite

- Lookaside List Head Overwrite
- We have learned from heap internals that Lookaside is the first option to satisfy allocate request, as well as free request
- We also know that the Lookaside table starts empty
- By default Lookaside location is fixed relatively to the heap
- Therefore …

1. If we can send request that will cause alloc with size < 1024
2. The application will free it to the Lookaside
3. Since we know Lookaside location..
4. We now know a memory location that points to our buffer!

Arbitrary Memory Overwrite

- Lookaside List Head Overwrite
- To find Lookaside entry location we need two parameters
- Heap base – The heap base is usually the same across service packs. It is not always the same across platforms
- Allocation size – Since we select the size we can control this value

- Lookaside Table = Heap base + 0x688
- Index = Adjusted(allocation size) / 8
- Lookaside entry location =
- Lookaside Table + Index * Entry size (0x30)

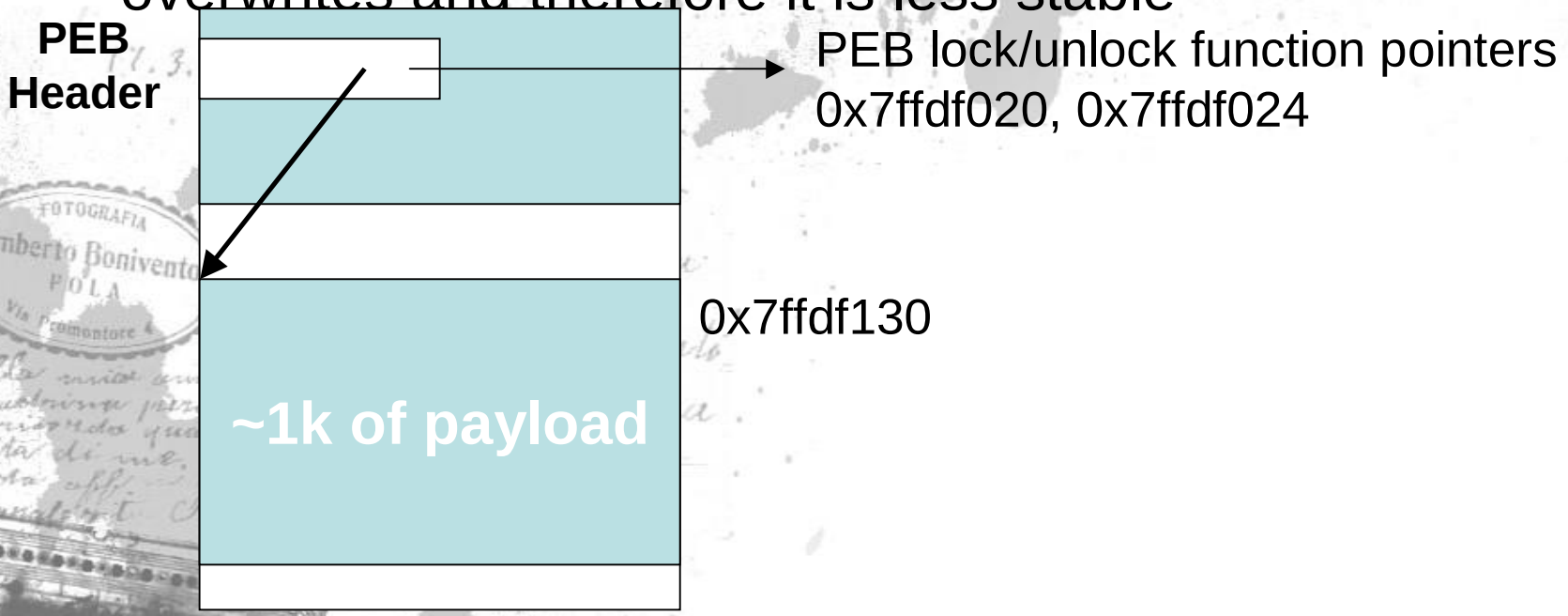
- Example: If Heap base is 0x70000, and allocated size is 922
- Index = Adjust(922) / 8 **è** 936 / 8 **è** 0x75
- Entry location = 0x70688+0x75*0x30 == 0x71c78

Arbitrary Memory Overwrite

- Lookaside Overwrite, 4-to-n-byte Overwrite (n <= ~1k)
- After populating the Lookaside entry we know the heap will return the same buffer if we request the same size again
- We will use arbitrary memory overwrite to change the value stored on the Lookaside entry
- Result: Next time we request the same buffer size, the heap will return the value we chose, allowing up to ~1k arbitrary memory overwrite!

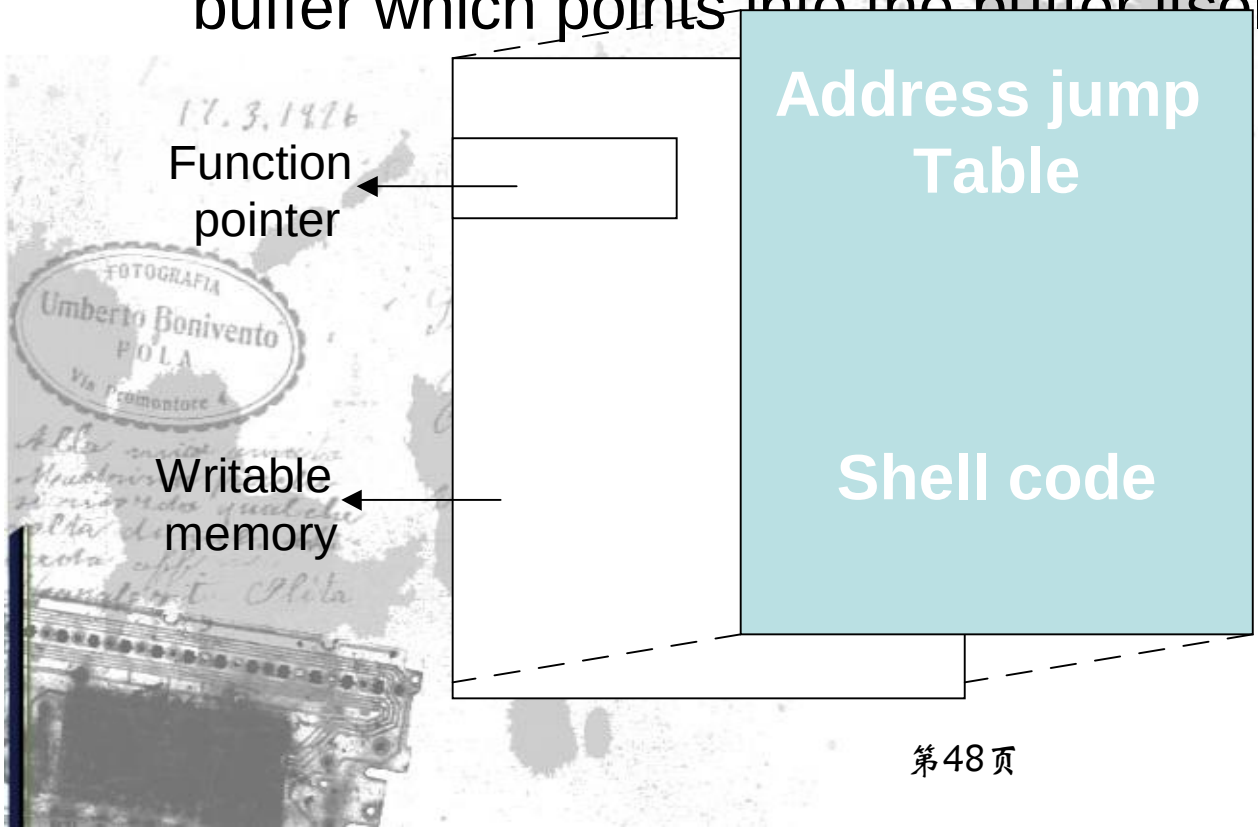
Arbitrary Memory Overwrite

- Uses of 4-to-n-byte Overwrite (Application A)
- First copy all our shell code to a known location
- Then redirect PEB lock function pointer to that location. This method requires two separate arbitrary memory overwrites and therefore it is less stable



Arbitrary Memory Overwrite

- Uses of 4-to-n-byte Overwrite (Application B)
- Choose a section of memory that has a function pointer in it and copy our ~1k buffer on top of it. Since we know the location we can create an “address table” inside our buffer which points into the buffer itself



Arbitrary Memory Overwrite

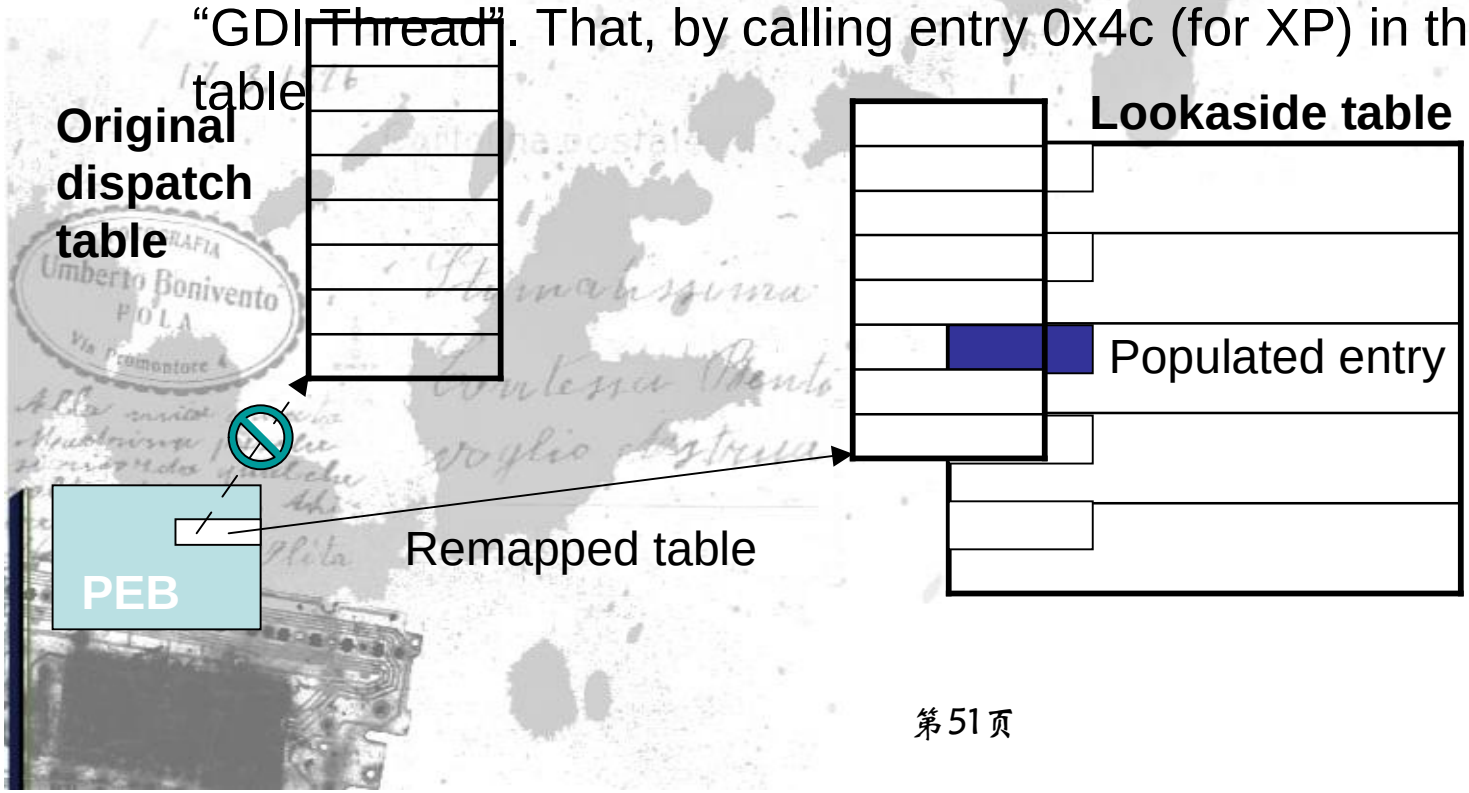
- Uses of 4-to-n-byte Overwrite (Application C)
- Find some writable string that the application uses as either path or command, overwrite it with malicious path or command
-
- David Litchfield gives an example of changing the string that is used by the “GetSystemDirectory” routine. Changing this path will allow loading of attacker DLL without code execution
- \\1.2.3.4\backdoors\
- c:\winnt\system32\

Arbitrary Memory Overwrite

- Remapping Dispatch Table
- Instead of changing the Lookaside entry to allow us to write ~1k to an arbitrary location we can just redirect some other pointer to this known location
- Dispatch table can be a perfect candidate. Since in dispatch table every item in the table is pointer to a function, if we can remap a dispatch table to overlap the Lookaside and predict which entry will be used in the dispatch table, we can populate the right entry that will conveniently point to our buffer
- Luckily we have such an example

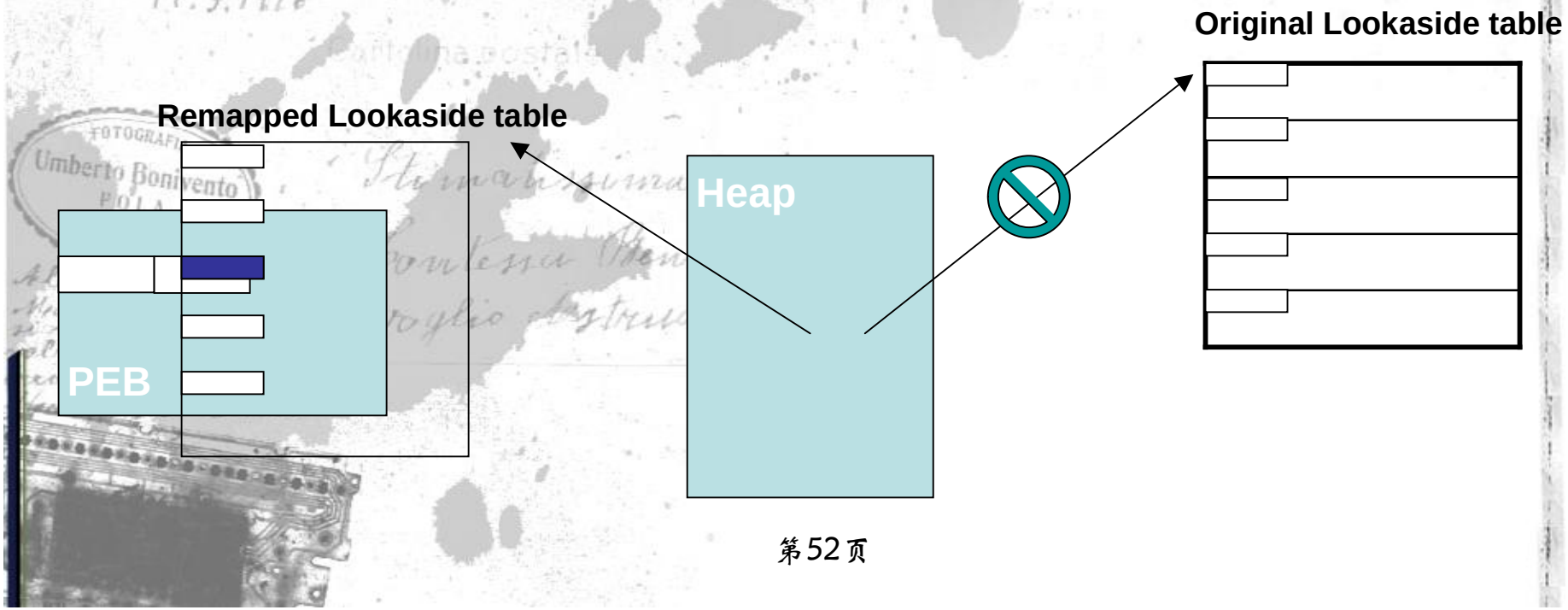
Arbitrary Memory Overwrite

- Remapping Dispatch Table (GUI Applications)
- The PEB contains a dispatch table for “callback” routines. This table is used in collaboration with the GDI component of the kernel
- Since the table is pointed to by the PEB the address is universal
- 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



Arbitrary Memory Overwrite

- Remapping Lookaside
- Although the Lookaside default location is 0x688 bytes from heap base, still the heap reference the Lookaside tables through a pointer
- We can change that pointer to overlap a function pointer
- Once we do it all we need is to allocate the right size, and the pointer will be automatically populated with the address of our buffer



Arbitrary Memory Overwrite

- Remapping Lookaside
- Limitation for Lookaside remapping
- Zero area will serve as good empty Lookaside space. If Lookaside is remapped over non zero area, we need to be careful since heap might return unknown values in alloc()
- Buffer will be freed into Lookaside only if Lookaside depth is smaller them max depth. (i.e. short value at offset 4 should be smaller than short value in offset 8)
- The address that is being overwritten by the heap as if it were the Lookaside entry is “pushed” on the Lookaside “stack”. Meaning, it will overwrite the first 4 bytes of your buffer. Therefore if these bytes make invalid command, it is not possible to use this method

Arbitrary Memory Overwrite

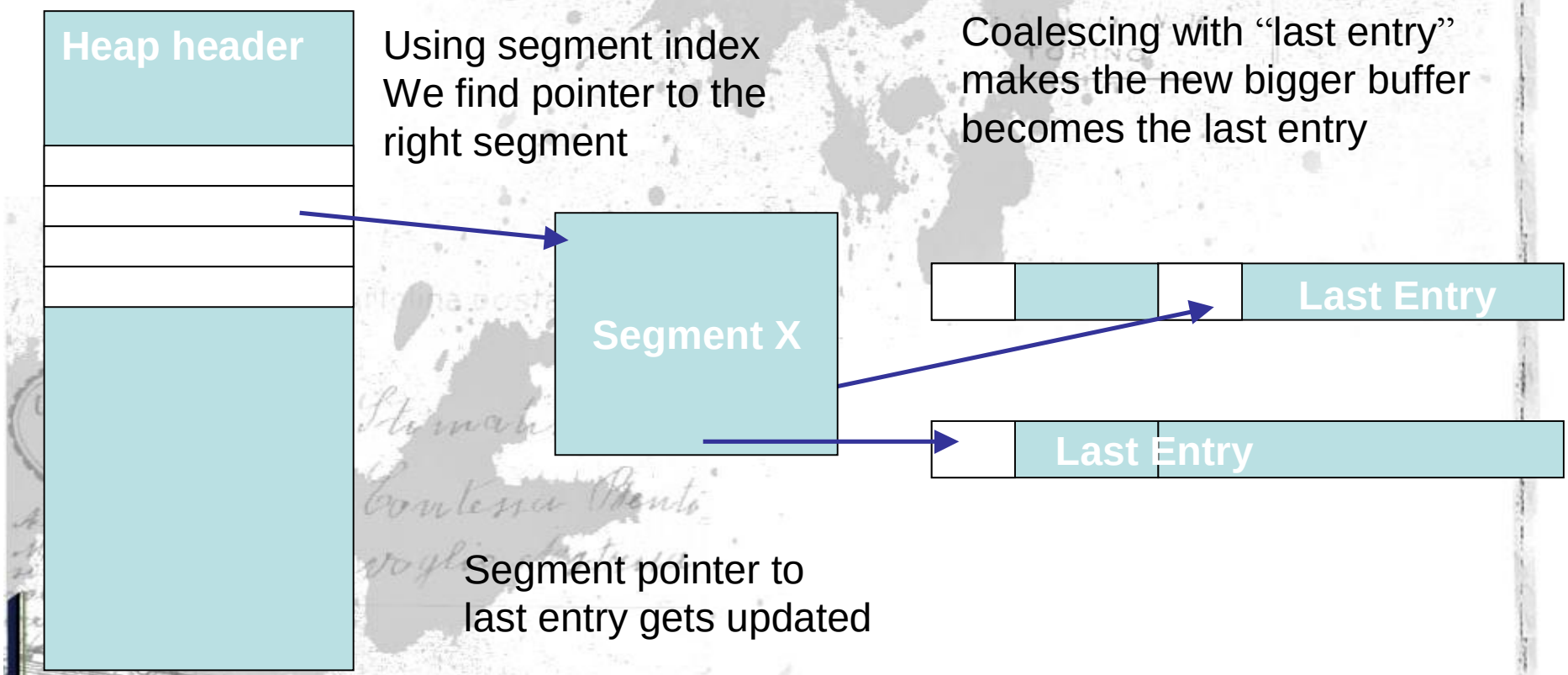
- Segment Overwrite (AddressOfSelf, Double Overwrite)
- Each segment in the heap keeps a pointer to the "Last entry" in the segment. Each time the segment is extended the last entry changes
- When a buffer is freed and coalesced it might coalesce with the last entry. When such a condition is met the segment updates its pointer to the last entry
- We can use this part of the algorithm to overwrite arbitrary memory with a pointer to our buffer

Arbitrary Memory Overwrite

- Segment Overwrite (AddressOfSelf, Double Overwrite)
- From the coalesce algorithm:
  1. If coalesced block has “Last entry” flag set
     1. Find segment using Segment index field of the chunk header
     2. Update segment’s last entry with new coalesced chunk address
- The operations above take place AFTER the arbitrary memory overwrite takes place as part of a coalesce of fake chunk
- Therefore, we can change the segment pointer in the heap structure and make the heap update arbitrary pointer with the address of our chunk

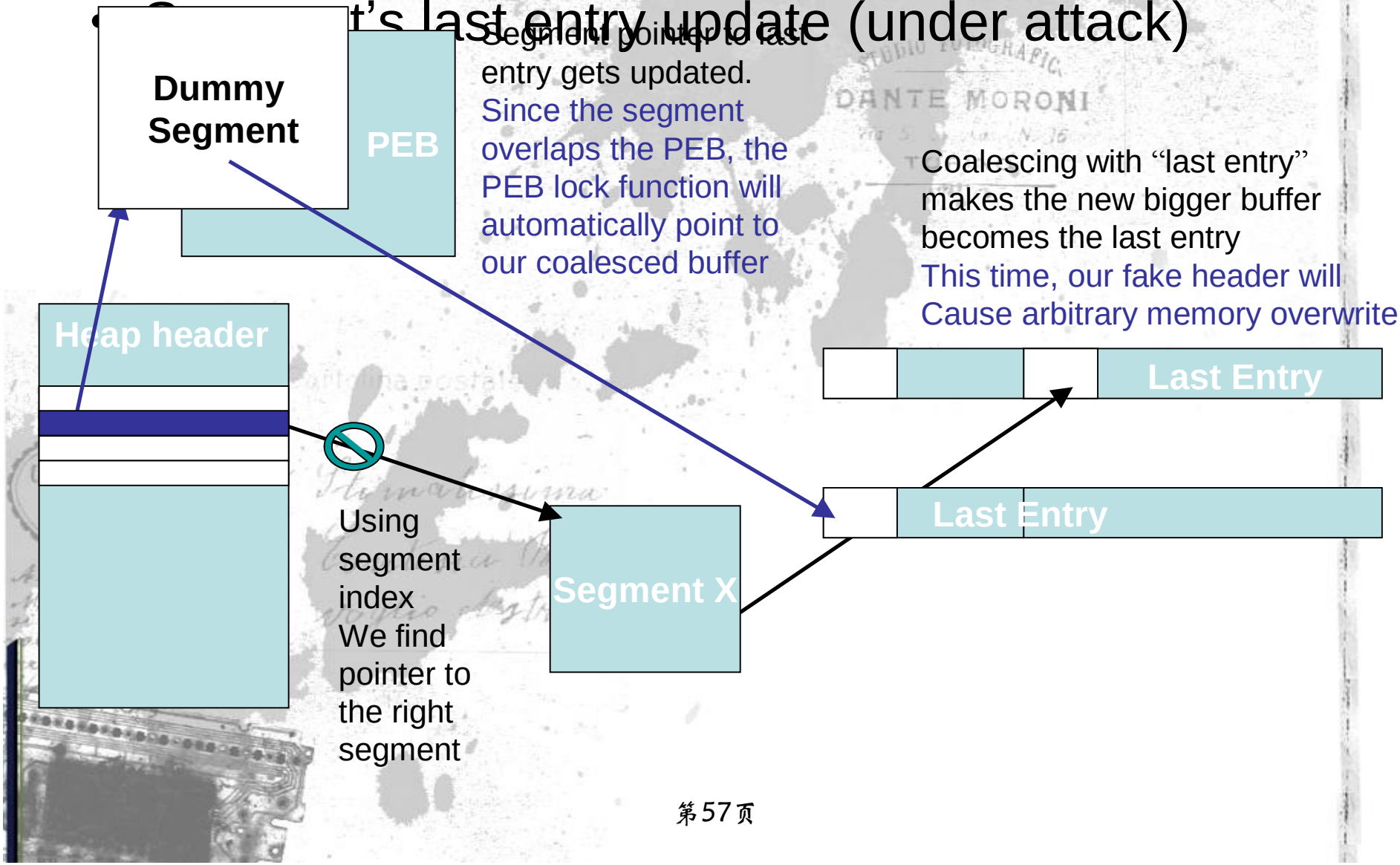
Arbitrary Memory Overwrite

- Segment's last entry update (normal operation)



Arbitrary Memory Overwrite

t's last entry update (under attack)



Arbitrary Memory Overwrite

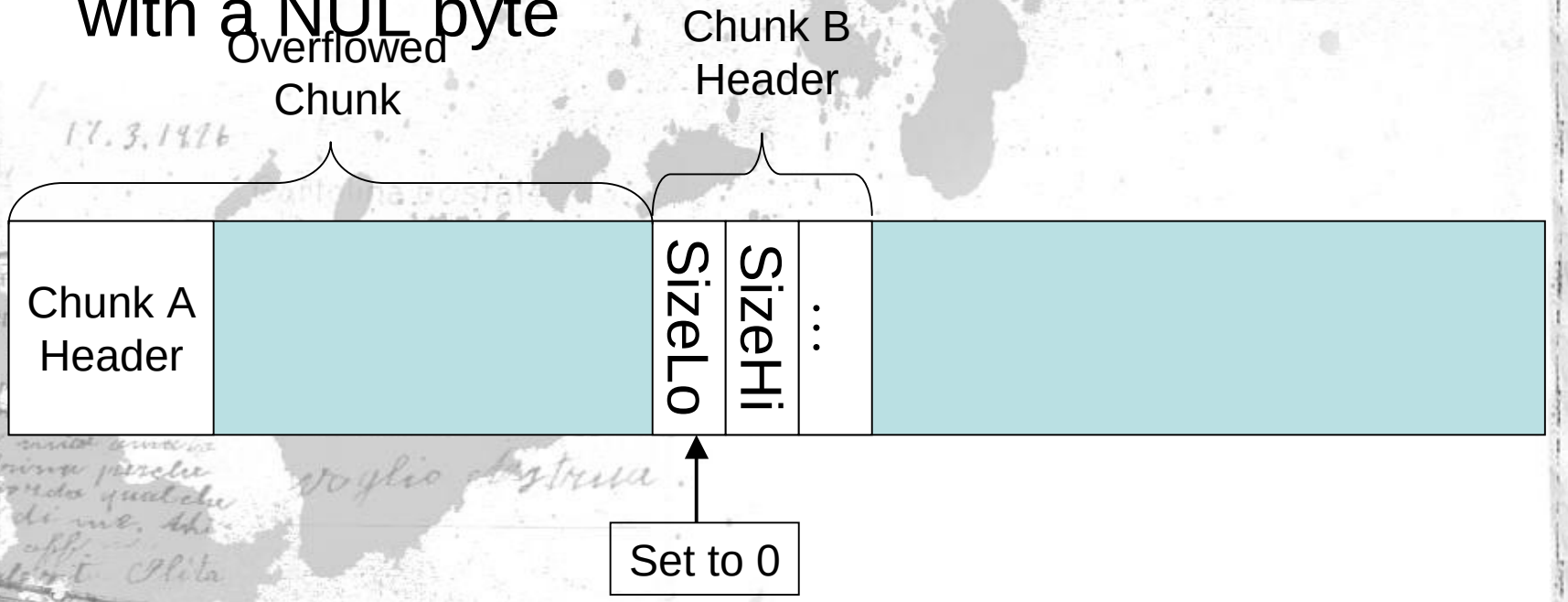
- Remapping Cache (AddressOfSelf)
- Cache at offset 0x170 in Heap
- Offset 0x2c of cache is an array of cached chunks > 1K
- Cache is usually NULL
- Similar results to Segment Overwrite for chunk sizes > 1K
- Less destructive than Segment Overwrite since it will not effect chunks < 1K

Arbitrary Memory Overwrite

- Remapping Cache (AddressOfSelf)
- Overwrite Cache pointer with SEH – (ChunkSize – 0x80) *4 – 0x2c
- ChunkSize is the size of the chunk you control (must be > 1K)
- When your chunk is freed, the pointer to it will be written into the SEH

Other Overflows?

- Off-By-Ones
- Off-by-ones for heap exploits means overwriting the lowest byte of the next block's chunk header with a NUL byte

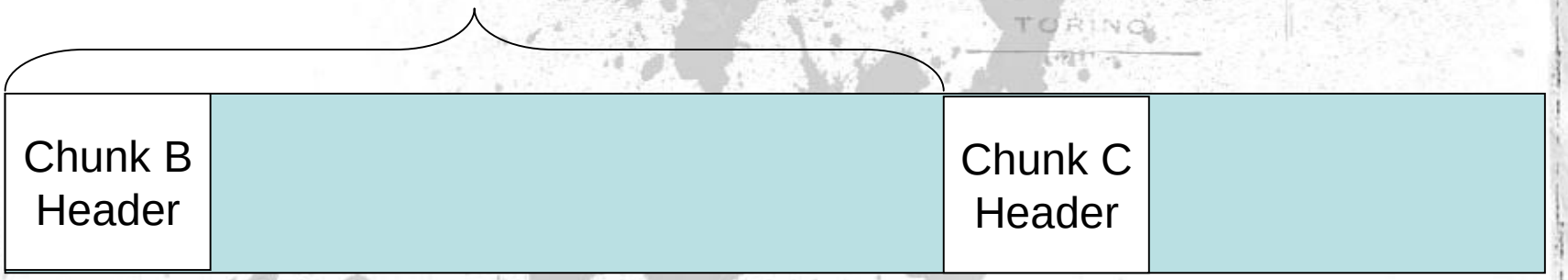


Off-By-Ones

- This will overwrite the lowest byte of the chunk size of the next chunk
- Only exploitable overwritten chunk was > 2K (because the high byte will be set)
- Otherwise, the chunk size will become 0 and this can't be exploited (for several different reasons)
- If ChunkSize > 2K this will shift down the start of the next chunk into the previous chunk

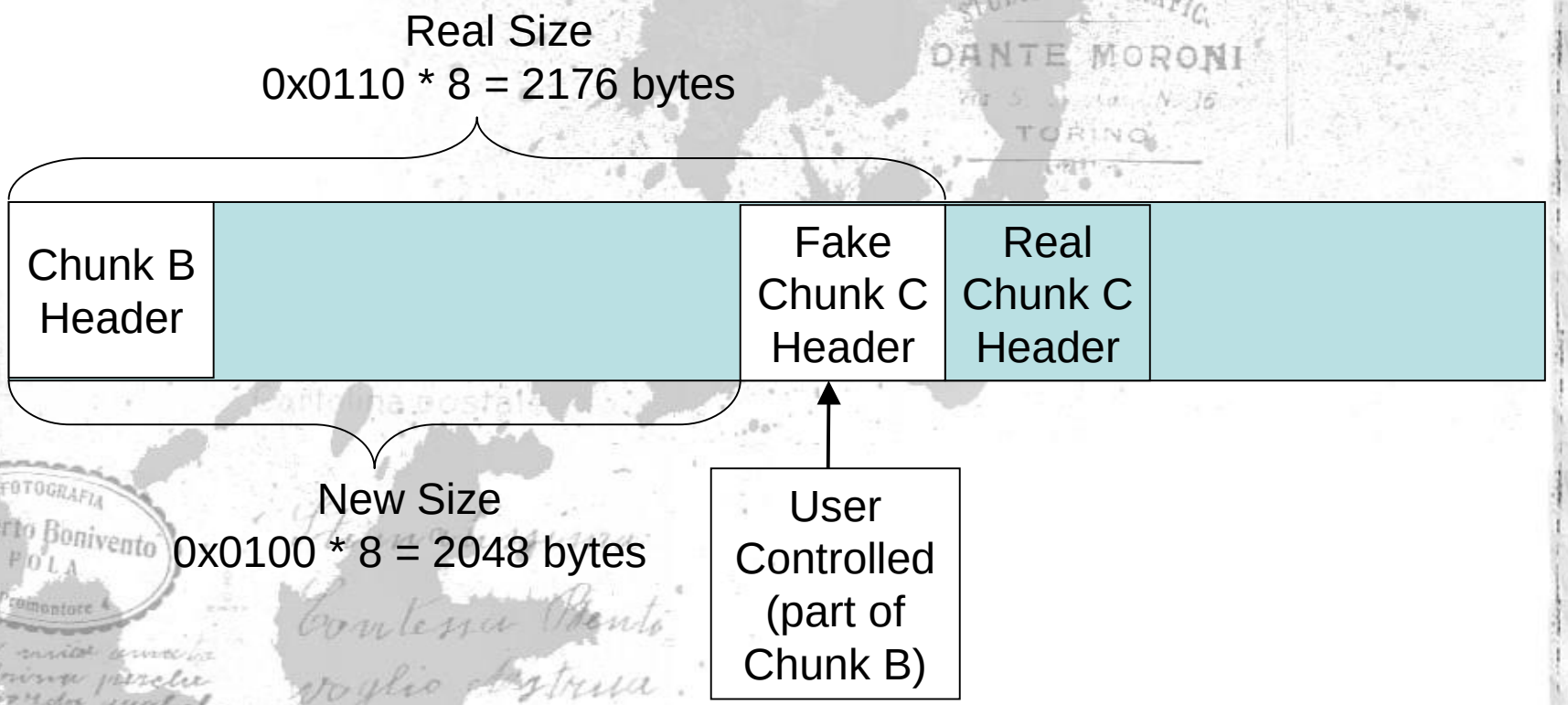
Off-By-Ones

- Before Off-By-One



Off-By-Ones

- After Off-By-One



Off-By-Ones

- This means you must have control of two sequential chunks A and B with chunk B > 2K bytes. One must:
- Cause off-by-one overflow in Chunk A
- This shifts down Chunk B's size
- Now fill in fake Chunk C header somewhere in Chunk B (where Chunk B thinks Chunk C starts)
- Fill in Chunk C header using the Overwrite on Coalesce technique
- When Chunk B is freed, 4-byte overwrite occurs

Other Overflows?

- Double Frees
- On Windows, only exploitable if:
- Chunk to be double freed is coalesced with previous chunk
- User can get the coalesced chunk before the double free

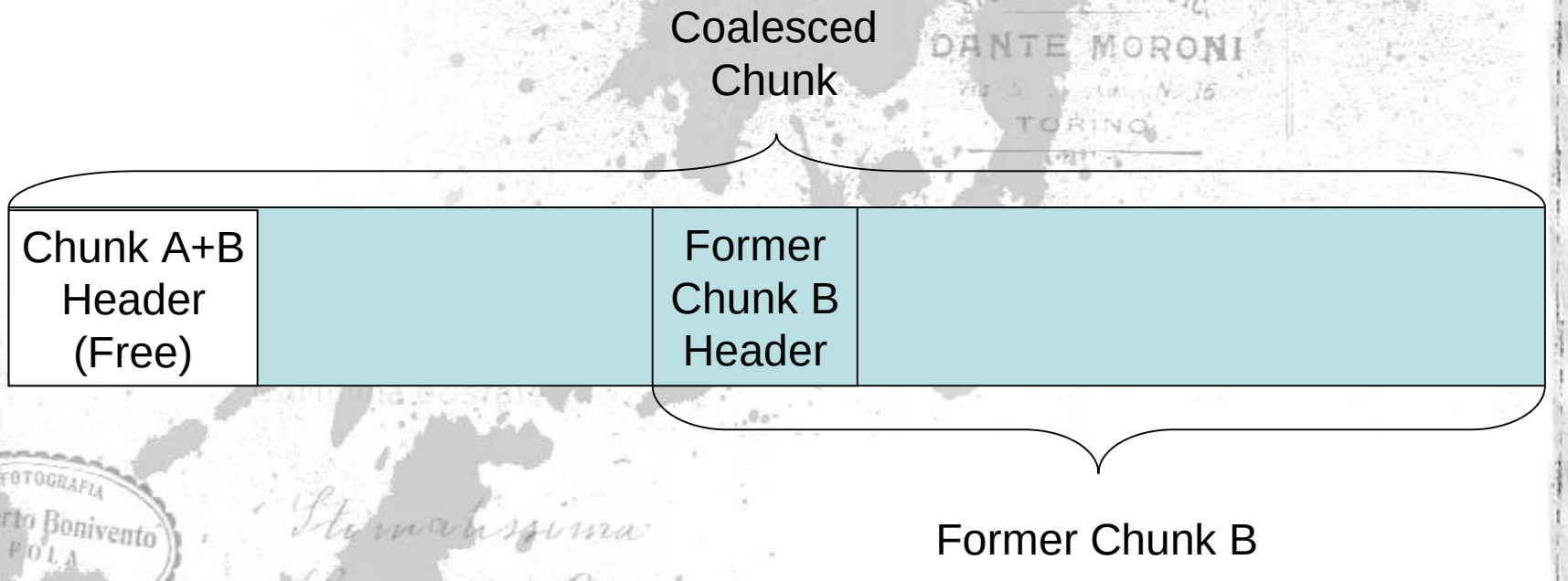
Double Free

- Chunk A and B (B is to be double freed)

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

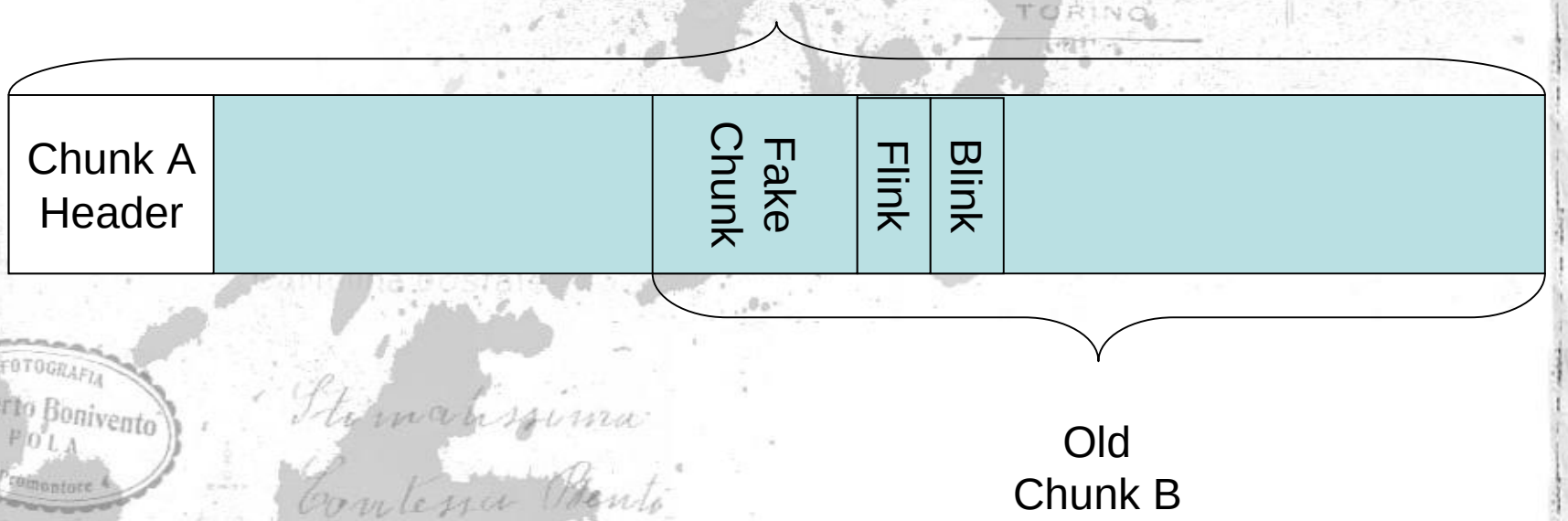
Double Free

- Chunk A and B (after B is freed)



Double Free

- User allocates Chunk A, sets up fake header, and waits for Chunk B to be freed again



Shellcode Notes

- Stabilizing execution environment
- To achieve arbitrary memory overwrite we have most likely corrupted the heap. In order to allow the shell code to execute successfully we need to fix the heap
- In addition to the corrupted heap we also overwritten the PEB lock routine we need to reset this pointer or else our shell code will be called again and again each time the lock routine is called
- Once the heap and lock routine are taken care of, we can execute our normal shell code

Shellcode Notes

- Fixing the corrupted heap
  - 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

XP Service Pack 2

- Major advancement in windows security
- Enforce better “out-of-the-box” security policy
- Reduce the amount of exposed interfaces. For example:
  - Firewall is on by default
  - RPC does not run anymore over UDP by default
- Improved web browsing and e-mail security
- For the first time windows code attempts to create obstacles for exploits development (MS Talk “Isolation & Resiliency”)

XP Service Pack 2

- Heap specific security improvement
- XP Service pack 2 includes multiple changes to address method of heap exploitation

- PEB randomization (note: still no heap randomization!)
- Security cookies on chunks header
- Safe unlink from doubly linked list

XP Service Pack 2

- PEB Randomization
- Until XP SP2 the PEB was always at the end of the user mode address space. Typically that address was 0x7ffdf000. (This address could have changed in case of the 3GB configuration)

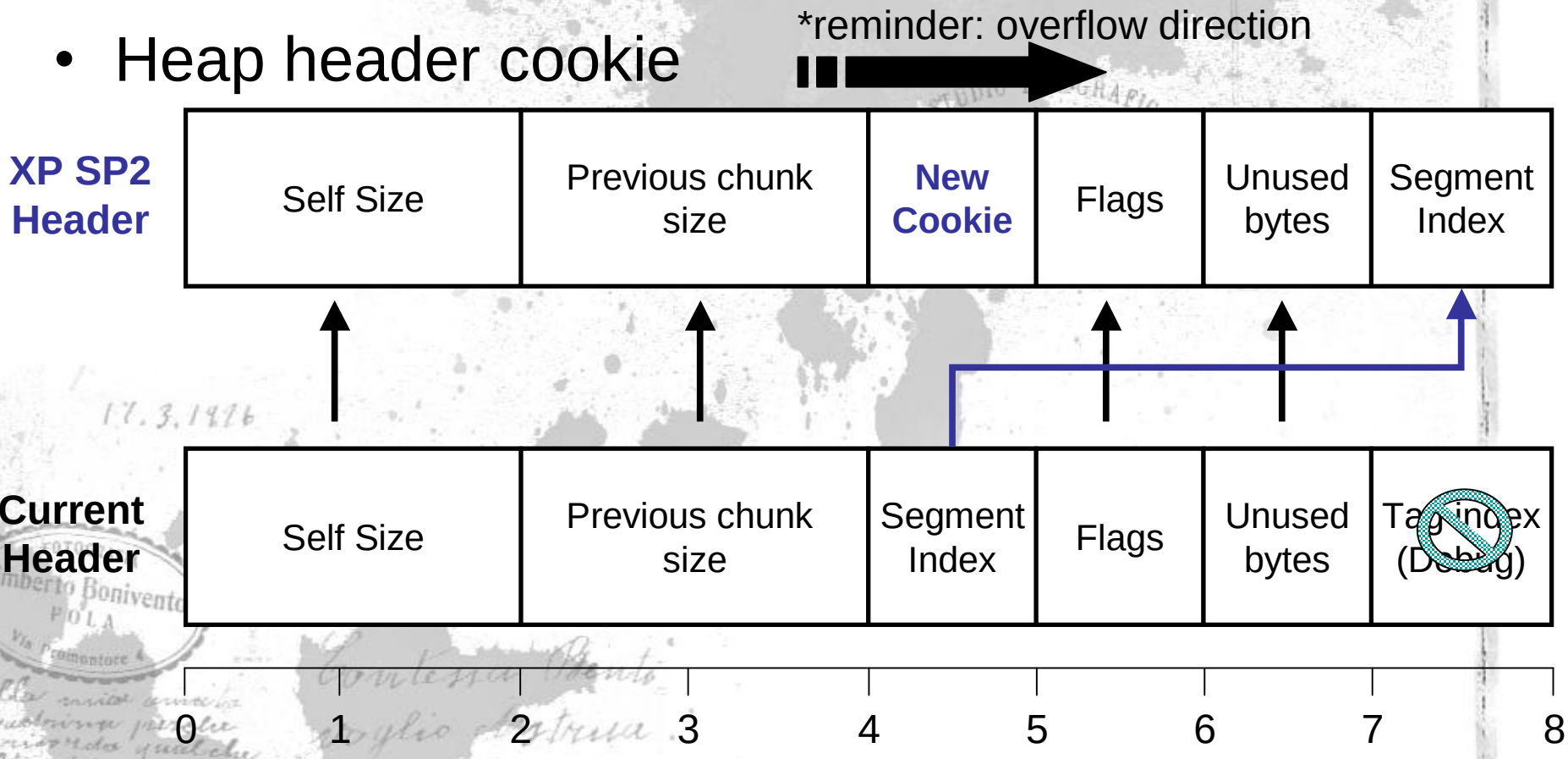
- Starting from XP SP2 the PEB location is no longer constant

- Early testing with the XP SP2 release candidate 1 showed us that the PEB stays close to the old address but may shift by a few pages.

- Sample new locations: 0x7ffdd000, 0x7ffd8000 etc..

XP Service Pack 2

- Heap header cookie



XP Service Pack 2

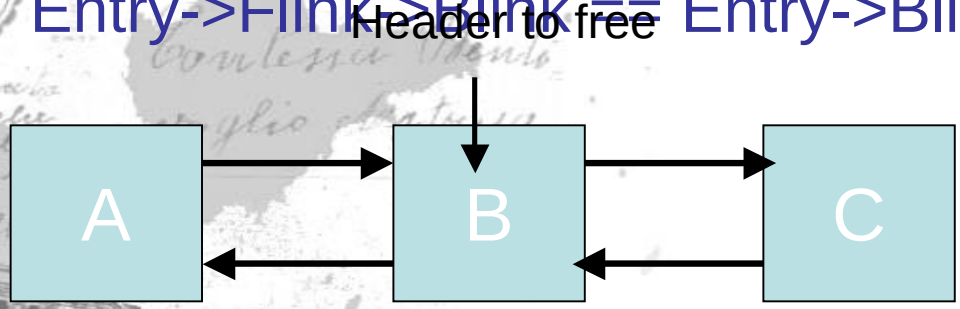
- Heap header cookie calculation
- The cookie of the heap will be calculated as follows
- Cookie = (&Heap_Header / 8) XOR Heap->Cookie

- The address of the heap will determine the cookie. In other words, in order to know the value of the cookie, you need to know the address of the header you overflow! It is clear that we cannot easily guess that. Otherwise there would be no use for all the methods we have presented here.

- On the other hand, the cookie is only one byte, there are only 256 possible values

XP Service Pack 2

- Safe unlinking
- The unlink operation is designed to take an item out of a doubly link list
- In the example below, B should be taken out the list. C should now point back to A, and A should point forward to C.
- XP SP2 heap will make sure that at the time of unlinking the following statement is true
- Entry->Flink->Blink == Entry->Blink->Flink == Entry



XP Service Pack 2: Summary

- It seems the arbitrary 4-byte overwrite will not be possible anymore
- These changes will not prevent attacks that utilize overwrite specific structures on the heap. This is what heap exploits until the 4-byte overwrite techniques were discovered.
- Much more research must be done on the XP SP2's changes. New exploitation techniques will likely evolve in the following months.

Other Protection Mechanisms

- If using XP SP2 is not an option, the next best thing is to randomize the heap base.
- Use similar technique XP SP2 does with PEB for heap base
- Changing the SizeOfHeapReserve or SizeOfHeapCommit in the NT_HEADERS section of the PE will change the heap base. This will add a layer of protection against worms
- Still bruteforcing is possible
- Hopefully XP SP2 changes will be retroactively added elsewhere

Summary: Overwrite Types

- 4-byte Overwrite
  - Able to overwrite any arbitrary 32-bit address (WhereTo) with an arbitrary 32-bit value (WithWhat)
- 4-to-n-byte Overwrite
  - Using a 4-byte overwrite to indirectly cause an overwrite of an arbitrary n bytes
- Double 4-byte Overwrite:
  - Two 4-byte Overwrite result from the same operation
- AddressOfSelf Overwrite:
  - 4-byte overwrite where you control WhereTo, and WithWhat is the address of a chunk you control

Summary: Overwrite Sources

- Coalesce-On-Free Overwrite:
  - A 4-byte Overwrite that occurs when the overflowed chunk (the source of the overflow) gets freed
- Coalesce-On-Free Double Overwrite:
  - A 4-byte Overwrite that occurs when the chunk after the overflowed chunk (the on with a fake header) gets freed
- VirtualAlloc Overwrite:
  - A 4-byte Overwrite that occurs while freeing a virtually allocated block

Summary: Overwrite Attacks

- ListHead Overwrite:
  - 4-byte Overwrite, WhereTo is a Lookaside or FreeList list head that leads to a 4-to-n-byte Overwrite
- Segment Double Overwrite:
  - Double 4-byte AddressOfSelf overwrite
- Remapping the Lookaside:
  - 4-byte Overwrite, WhereTo is the Cache pointer in the heap structure
- Remapping the Cache:
  - 4-byte Overwrite, WhereTo is the Cache pointer in the heap structure