MODERN VULNERABILITY EXPLOITATION: THE HEAP OVERFLOW

Heap

- Heap
  - Sometimes called the free store
  - Dynamically allocated area of memory
  - Stores global variables
  - Stores large variables
  - Stores dynamic variables
  - Controlled by the heap manager
  - Heap implementations vary from system to system
    - Heap exploits are heap implementation specific
    - So, heap exploits are system specific

Dynamic Memory Allocation

- malloc()
  - Allocate a chunk of memory for use
- realloc()
  - Resizes an allocated chunk of memory
- free()
  - Return a chunk of allocated memory to the system

Dynamic Memory Allocation

- Properly Used Memory Allocation

```
#include <stdio.h>
#include <stdlib.h>

void main(int argc, char** argv) {
    char *buf;

    buf = (char *) malloc(strlen(argv[1]) + 1);
    strcpy(buf, argv[1]);
    printf("%s\n", buf);
    free(buf);
}
```

```
C:\Documents and Settings\Jojo\Desktop\Advanced Reverse Engineering\sample_code\
heap>not_vuln.exe hello
hello

C:\Documents and Settings\Jojo\Desktop\Advanced Reverse Engineering\sample_code\
heap>not_vuln.exe hello123
hello123

C:\Documents and Settings\Jojo\Desktop\Advanced Reverse Engineering\sample_code\
heap>not_vuln.exe 1
1
```

Example Heap Vulnerabilities

- Shellcoder's Handbook (Second Edition)

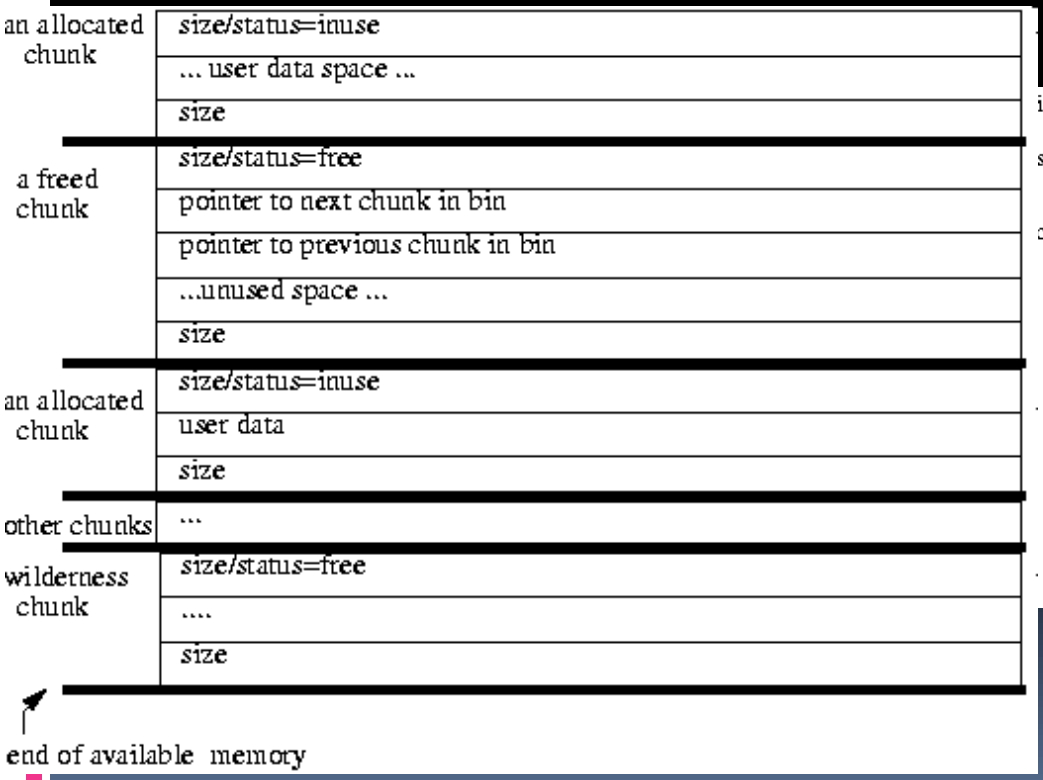
```
// Samba
memcpy(array[user_supplied_int], user_supplied_buffer, user_supplied_int2);

// Microsoft IIS
buf = malloc(user_supplied_int+1);
memcpy(buf, user_buf, user_supplied_int);

// Microsoft IIS off-by-a-few
buf = malloc(strlen(user_buf + 5));
strcpy(buf, user_buf);

// Solaris Login
buf = (char **) malloc(BUF_SIZE);
while (user_buf[i] != 0) {
    buf[i] = malloc(strlen(user_buf[i]) + 1);
    i++;
}

// Solaris Xsun
buf = malloc(1024);
strcpy(buf, user_supplied);
```

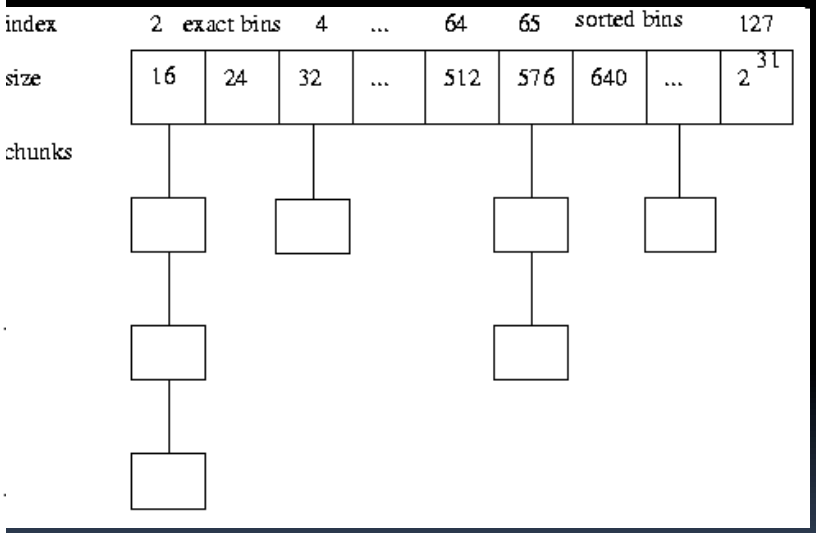
Linux malloc

- Doug Lea's malloc()
  - Called dlmalloc
  - Unallocated memory is grouped into "bins"
    - A bin is a linked list to all blocks of similar sizes
- Wolfram Gloger's malloc()
  - Called ptmalloc
  - Based on dlmalloc
- glibc's malloc()
  - Modified ptmalloc2 since glibc v2.3

dlmalloc

- Doug Lea's malloc()





http://g.oswego.edu/dl/html/malloc.html

ptmalloc

- Wolfram Gloger's malloc()
  - Includes fastbins

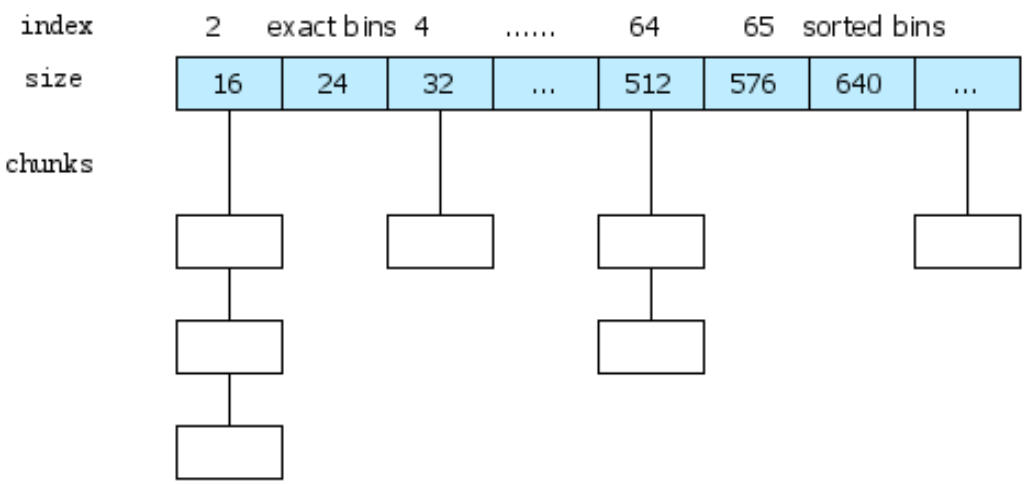


图 4: bins 结构示意图

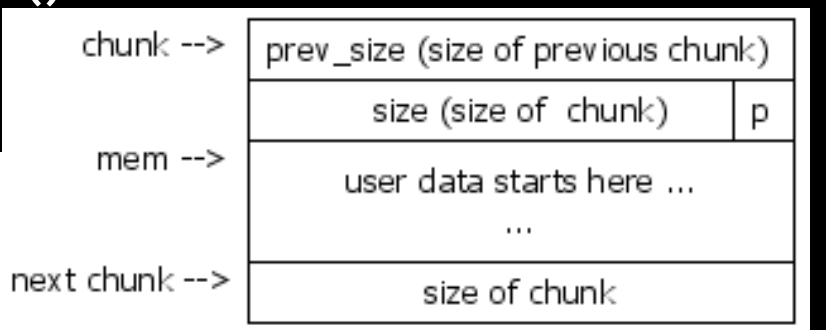


图 2: 使用中的chuck

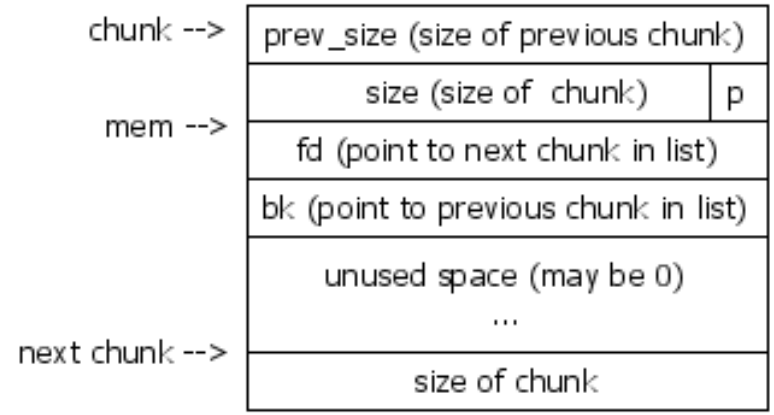


图 3: 空闲的thunk

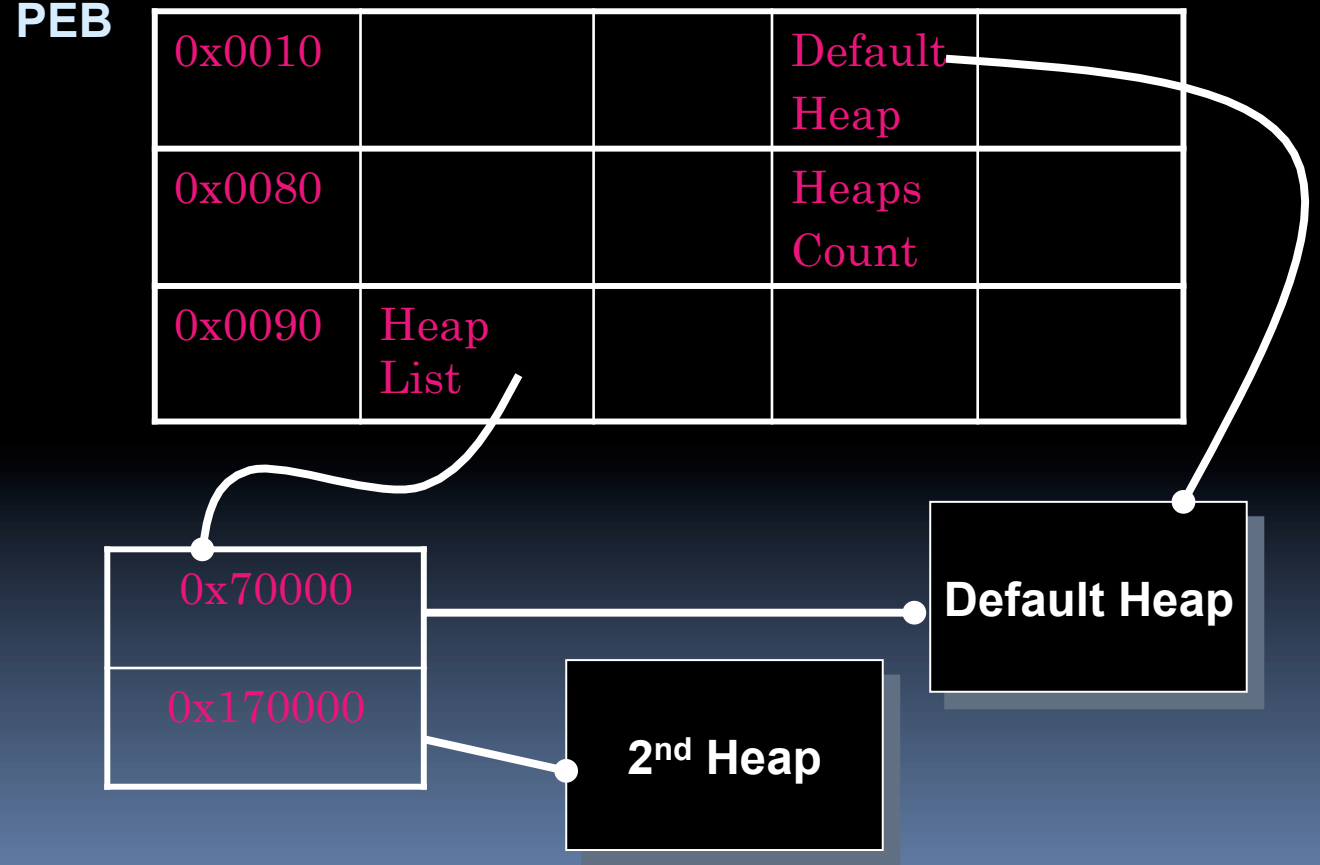
http://blog.csdn.net/phenics/archive/2006/06/07/777053.aspx

Windows Private Heaps

- Private Heaps
  - Every process has a default heap
  - We can create additional private heaps
  - HeapCreate()
  - HeapDestroy()
  - HeapAlloc()
  - HeapReAlloc()
  - HeapFree()

Windows Heaps

- Process Environment Block (PEB)



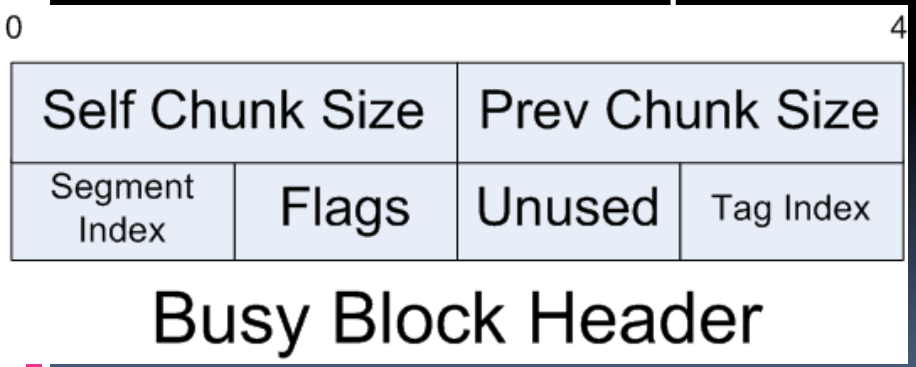
http://cybertech.net/~sh0ksh0k/projects/winheap/XPSP2%20Heap%20Exploitation.ppt

Windows Heap

- Windows
  - Heap is allocated into 8-byte chunks
    - Called "allocation units" or "indexes"
  - A set of allocated chunks is called a "block"
    - 18 bytes are needed, how large is our block?
  - Block headers

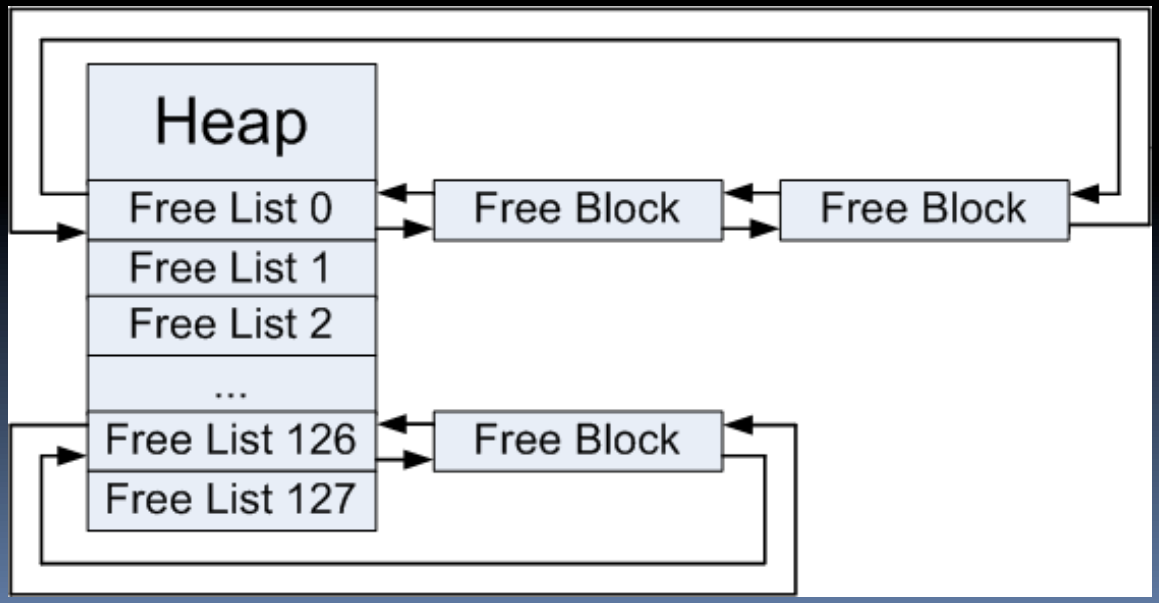
Windows Heap

- Windows Heap Block Headers
  - Size: (size of block + header) / 8
  - Segment index: memory segment for block
  - Unused: amount of free (additional) bytes
  - Flink/Blink: pointer to next/previous free block



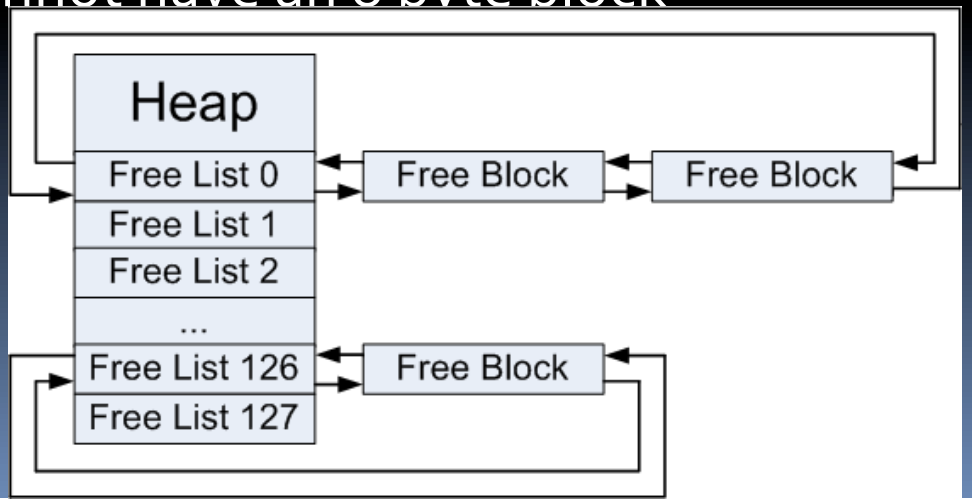
Windows Heap

- Windows Free List
  - Free blocks are recorded in an array of 128 doubly-linked lists
    - Called the free list



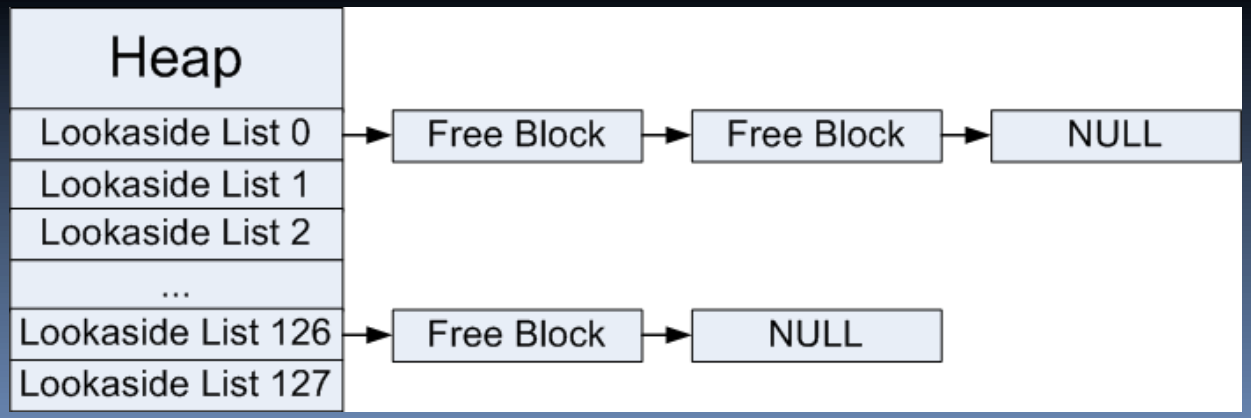
Windows Heap

- Windows Free List
  - Index of 0 holds blocks larger than 127 chunks
    - And less than 512K
    - Sorted from smallest to largest
  - Index of 1 is unused
    - Cannot have an 8 byte block



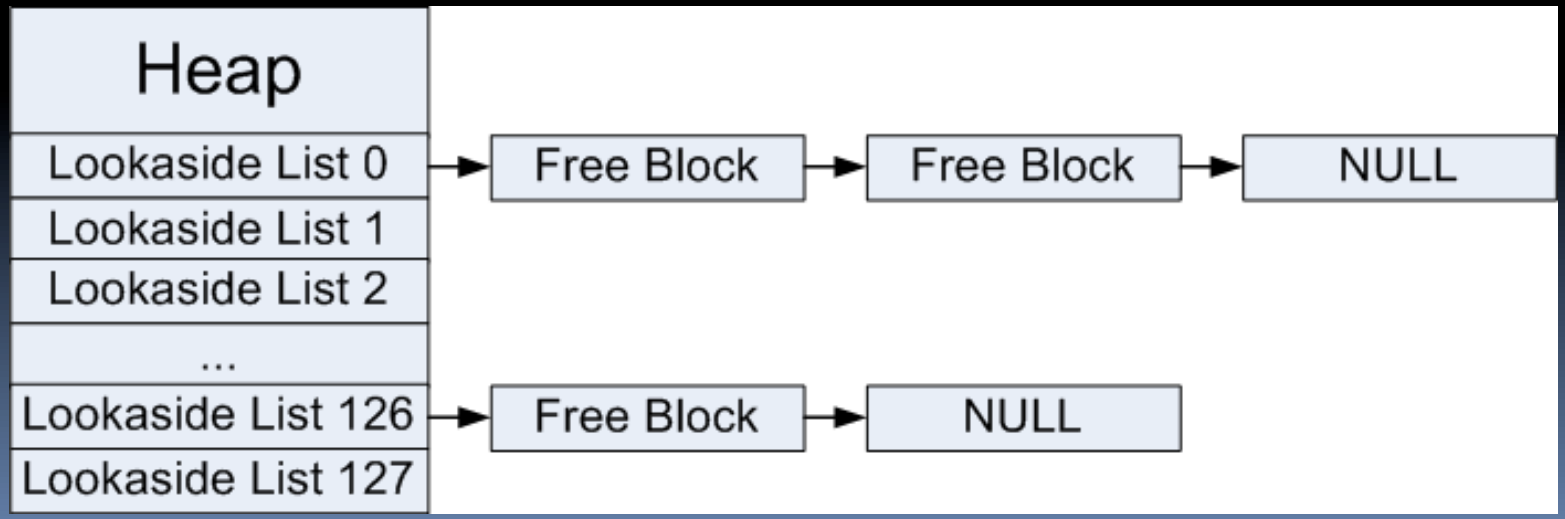
Windows Heap

- Windows Lookaside List
  - Freed blocks (marked as busy) are recorded in an array of 128 singly-linked lists
    - Called the lookaside list, used for fast allocates
  - Blocks are added to the lookaside list upon free()
    - Lookaside list is initially empty



Windows Heap

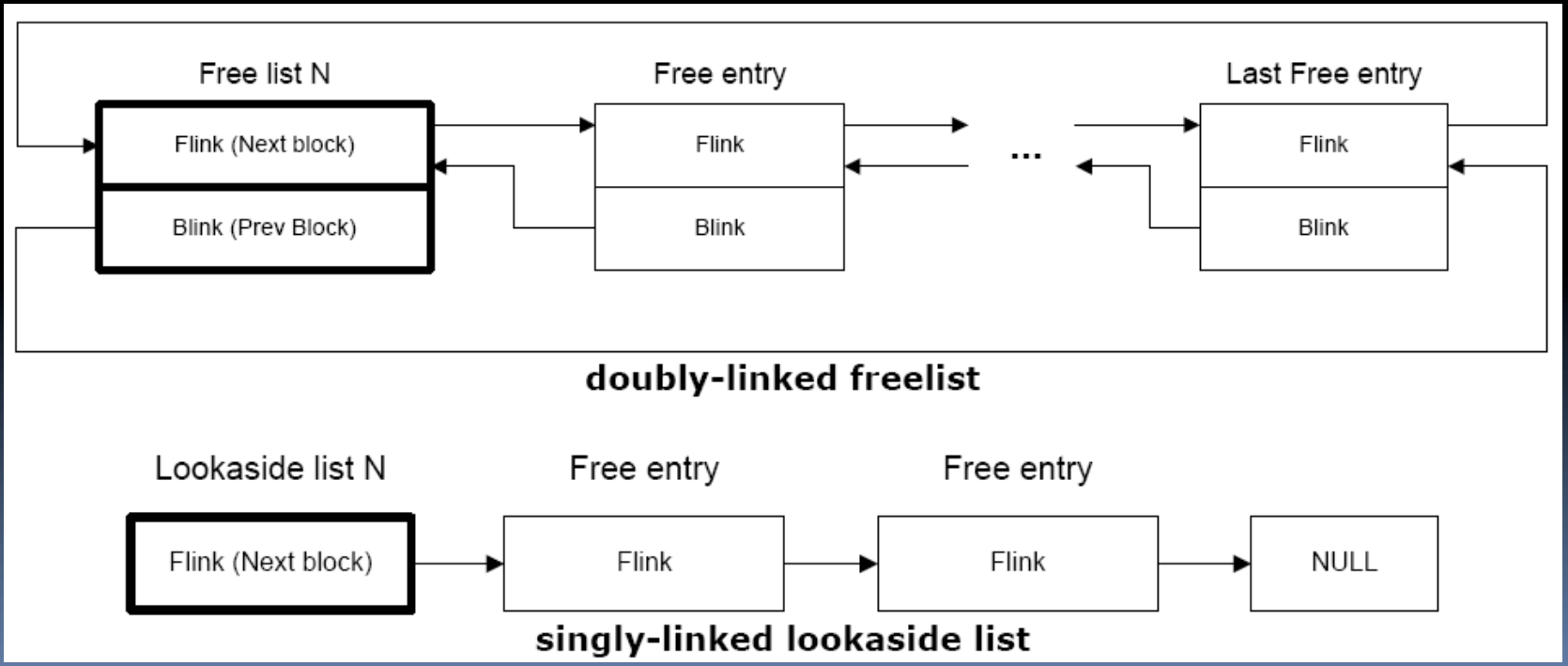
- Windows Lookaside List
  - Blocks are freed to the lookaside list first
    - Each lookaside list item can only hold 4 blocks
    - Blocks are restored to the free list if the lookaside is full



Windows Heap

- Windows Free Block Heap Management
  - From:

http://www.ptsecurity.com/download/defeating-xpsp2-heap-protection.pdf



Windows Heap

- Allocation Algorithm
  - If size >= 512K, virtual memory is used (not on heap)
  - If < 1K, first check the Lookaside lists. If there are no free entries on the Lookaside, check the matching free list
  - If >= 1K or no matching entry was found, use the heap cache (not discussed in this presentation).
  - If >= 1K and no free entry in the heap cache, use FreeLists[0] (the variable sized free list)
  - If a free entry can't be found, extend heap as needed

http://cybertech.net/~sh0ksh0k/projects/winheap/XPSP2%20Heap%20Exploitation.ppt

Windows Heap

- Free Algorithm
  - If the chunk < 512K, it is returned to a lookaside or free list
  - If the chunk < 1K, put it on the lookaside (can only hold 4 entries)
  - If the chunk < 1K and the lookaside is full, put it on the free list
  - If the chunk > 1K put it on heap cache (if present) or FreeLists[0]

http://cybertech.net/~sh0ksh0k/projects/winheap/XPSP2%20Heap%20Exploitation.ppt

Windows Heap

- Coalescing
  - Say, two adjacent memory blocks are freed
  - Windows tries to combines these memory blocks
    - Takes time
    - Reduces fragmentation
  - Combining freed memory blocks in this manner is called “coalescing”
  - Only blocks going into the free list coalesce

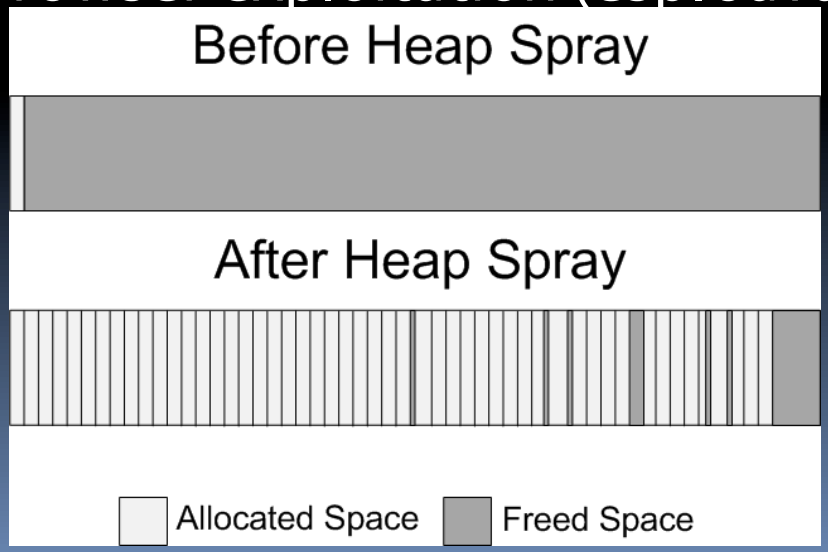
Windows Heap

- Free to Lookaside Algorithm
  - Free buffer to Lookaside list only if:
    - The lookaside is available (e.g., present and unlocked)
    - Requested size is < 1K (to fit the table)
    - Lookaside is not "full" yet (no more than 3 entries already)
  - To add an entry to the Lookaside:
    - Insert into appropriate singly-linked list
    - Keep the buffer flags set to busy (to prevent coalescing)

http://cybertech.net/~sh0ksh0k/projects/winheap/XPSP2%20Heap%20Exploitation.ppt

Heap Spraying

- Heap Spraying
  - Technique developed by SkyLined
  - Attempts to "spray" information on the heap
    - Makes position of allocated object predictable
  - Popular for browser exploitation (esp. JavaScript)



Heap Spraying

- Heap Spraying
  - Load many NOP/shellcode pairs to target heap
  - Typically, top of heap is allocated out first
  - So, a jump into this memory space has a good chance of landing in a NOP/shellcode pair

```
var nop = unescape("%u9090%u9090");

// Create a 1MB string of NOP instructions followed by shellcode:
//
// malloc header   string length   NOP slide   shellcode   NULL terminator
// 32 bytes        4 bytes         x bytes     y bytes     2 bytes

while (nop.length <= 0x100000/2) nop += nop;

nop = nop.substring(0, 0x100000/2 - 32/2 - 4/2 - shellcode.length - 2/2);

var x = new Array();

// Fill 200MB of memory with copies of the NOP slide and shellcode
for (var i = 0; i < 200; i++) {
    x[i] = nop + shellcode;
}
```

Heap Spraying

- Advanced Heap Spraying
  - Say, we can overwrite a C++ object on the heap
  - We can point it into the heap
  - Perform our heap spray
  - Wait for/invoke a virtual function call
    - Ex: Virtual function at [vtable + 8] is called

```
mov ecx, dword ptr [eax]     ; get the vtable address
push eax                     ; pass C++ this pointer as the first argument
call dword ptr [ecx+08h]     ; call the function at offset 0x8 in the vtable
```

```
object pointer   -->   fake object   -->   fake vtable      -->     fake virtual function

addr: xxxx             addr: yyyy          addr: 0x0C0C0C0C         addr: 0x0C0C0C0C
data: yyyy             data: 0x0C0C0C0C    data: +0 0x0C0C0C0C      data: nop slide
                                                 +4 0x0C0C0C0C            shellcode
                                                 +8 0x0C0C0C0C
```

Heap Feng Shui

- Heap Feng Shui
  - Techniques that manipulate the heap layout
    - Often dependent on size of blocks
  - Plunger technique
    - Allocate and free 6 large blocks
    - Clears the heap cache
  - Defragmenting the heap
    - Allocate blocks that are the size of our exploit
    - All available "holes" are filled and new blocks are allocated from the end of the heap
  - …Many others:

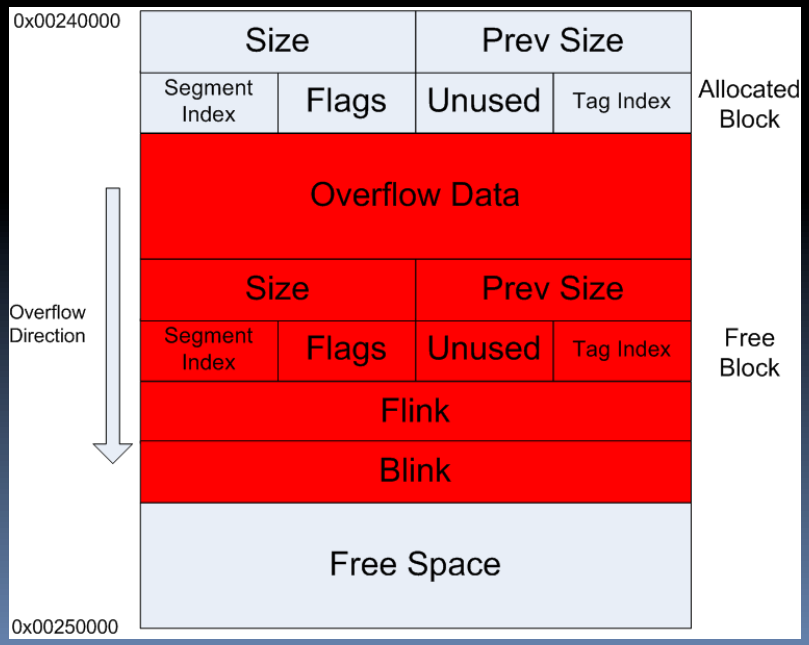
http://www.blackhat.com/presentations/bh-europe-07/Sotirov/Presentation/bh-eu-07-sotirov-apr19.pdf

Targets for a Heap Overflow

- Control Pointers
  - Global offset table (GOT)
  - Global function pointers
  - Virtual function pointers (vtable)
  - PEB function pointer
  - Thread environment block (TEB)
  - Unhandled exception filter (UEF)
  - Vectored exception handling (VEH)
  - Destructors (.DTORS)
  - atexit handlers
  - Stack values
  - Function pointers in general
- Global/Dynamically Allocated Data
  - Variables on the heap

Exploiting Heap Metadata

- Coalesce-on-Free 4-byte Overwrite
  - Say, we have an allocated block (with an overflow) followed by a free block in memory
  - We can overwrite Flink and Blink



Exploiting Heap Metadata

- Coalesce-on-Free 4-byte Overwrite
  - Say, we have an allocated block (with an overflow) followed by a free block in memory
  - We can overwrite Flink and Blink
  - If our overflow block is coalesced, this code is executed:

```
mov dword ptr [ecx],eax
mov dword ptr [eax+4],ecx

EAX - Flink
ECX - Blink
```

  - Meaning?
    - Arbitrary 32 bit overwrite (UEF is a common target)
    - Great method for systems < XPSP2

Exploiting Heap Metadata

- Lookaside List Head Overwrite
  - 4-to-n-byte overwrite
  - Overwrite a lookaside list head
  - Allocate that head
  - Allocated chunk points to value of overwrite
  - We can overwrite whatever we want
    - It's like having access to raw malloc calls
    - Common situation for heap exploits

Exploiting Heap Metadata

- Lookaside List Head Overwrite (How-to)
  - Use the Coalesce-on-Free Overwrite, with these values:
    - FakeChunk.Blink = &Lookaside[ChunkSize] where ChunkSize is a pretty infrequently allocated size
    - FakeChunk.Flink = what we want a pointer to
  - To calculate the FakeChunk.Blink value:
    - LookasideTable = HeapBase + 0x688
    - Index = (ChunkSize/8)+1
    - FakeChunk.Blink = LookasideTable + Index * EntrySize (0x30)
  - Set FakeChunk.Flags = 0x20, FakeChunk.Index = 1-63, FakeChunk.PreviousSize = 1, FakeChunk.Size = 1

http://cybertech.net/~sh0ksh0k/projects/winheap/XPSP2%20Heap%20Exploitation.ppt

Exploiting the UEF

- Unhandled Exception Filter (UEF)
  - "Last ditch effort" exception handler
  - Our goal is to install our own UEF

Exploiting the UEF

- Unhandled Exception Filter (UEF)
  - Location is OS and SP dependent
  - Find the location by disassembling SetUnhandledExceptionFilter()
  - NGSSoftware example:

```
77E7E5A1    mov ecx, dword ptr [esp+4]
77E7E5A5    mov eax, [77ED73B4h]
77E7E5AA    mov dword ptr ds:[77ED73B4h], ecx
77E7E5B0    ret 4
```

  - UEF = 0x77ED73B4

Exploiting the UEF

- Unhandled Exception Filter (UEF)
  - Windows XP
    - EDI contains a pointer to an EXCEPTION_POINTERS structure on the stack when UEF is called
      - 0x78 bytes past EDI there's a pointer to the end of our buffer (we could make that the start of our shellcode!)
  - Use our arbitrary 32-bit overwrite to patch the UEF address to point to:
    - WinXP: call dword ptr [edi+0x78]
      - Found in netapi.dll, user32.dll, rpcrt4.dll
    - Win2000: call dword ptr [esi+0x4c]
      - Or: call dword ptr [ebp+0x74]
  - An unhandled exception will trigger the UEF

Exploiting the VEH

- Vectored Exception Handler (VEH)
  - New feature starting in Windows XP
  - Vectored exception handling occurs before any frame-based handlers (like SEH)
  - Pointer to first VEH node is at a hardcoded address
    - Our goal is to overwrite this pointer

Exploiting the VEH

- Vectored Exception Handler (VEH)
  - `_VECTORED_EXCEPTION_NODE` stored on heap
  - Windows XP SP2, `0x77FC3210`
    - Pointer to first `_VECTORED_EXCEPTION_NODE`

```
struct _VECTORED_EXCEPTION_NODE {
    DWORD m_pNextNode;
    DWORD m_pPreviousNode;
    PVOID m_pfnVectorHandler;
}
```

Exploiting the VEH

- Vectored Exception Handler (VEH)
  - "Create" our own VEH structure
    - Fix the first VEH pointer to point to our VEH struct
  - We find a pointer to our buffer on the stack
    - Set the first VEH pointer to [buf_ptr - 8]

Exploiting the PEB

- Process Environment Block (PEB)
  - Stored in heap
  - Each process has a single modifiable PEB
    - Contains function pointers to:
      - RtlEnterCriticalSection (+0x20 in PEB)
        - Called FastPebLockRoutine in PEB
      - RtlExitCriticalSection (+0x24 in PEB)
        - Called FastPebUnlockRoutine in PEB
    - Example use:
      - ExitProcess() → RtlAcquirePebLock() → PEB.FastPebLockRoutine → RtlEnterCriticalSection()

Exploiting the PEB

- Process Environment Block (PEB)
  - The PEB address is predictable
    - In WinXP, NT4, 2000, 2003
    - Not exploitable in Win2003
      - Function pointers are randomized
  - Exploit
    - Overwrite RtlEnterCriticalSection with pointer to instruction that executes the back of our buffer

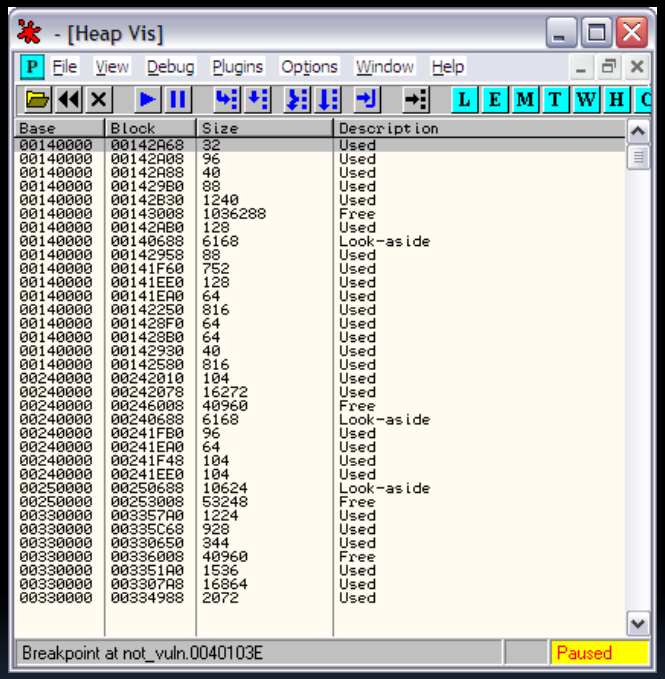
Exploiting the TEB

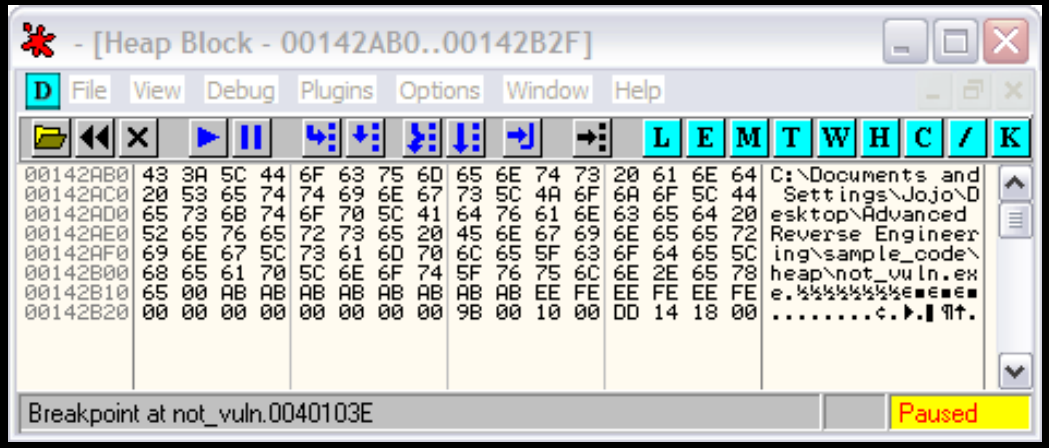
- Thread Environment Block (TEB)
  - TEB exception handler pointer
    - First TEB has a base address of `0x7FFDE000`
      - Grows towards `0x00000000`
    - Say the thread exits
      - A new thread will be assigned the old thread's TEB address
      - Leads to a "messy" and sometimes unreliable TEB
    - Reliable if the address is predictable
  - Once again, replace pointer with address to an instruction that will execute the back of our buffer

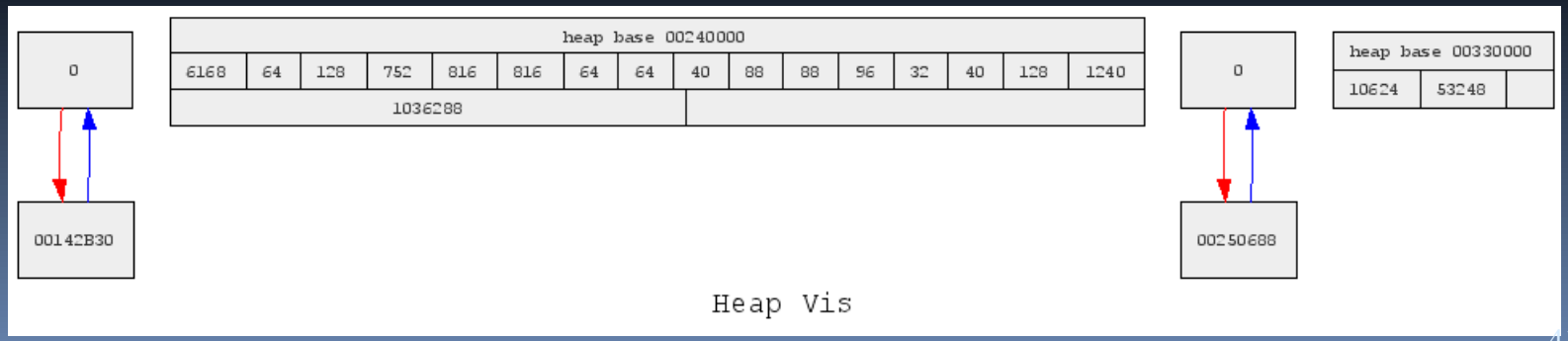
Repairing the Heap

- Repairing the Heap
  - Necessary step for exploit stability
  - Reset the heap to look like a brand new heap
    - Generic, reusable method
    - All allocated blocks will stay intact
    - New allocations are still possible
    - See NGSSoftware code (asm-repair-heap)

OllyDbg HeapVis Plugin







Questions/Comments?