

Memory Allocator Attack and Defense

Richard Johnson

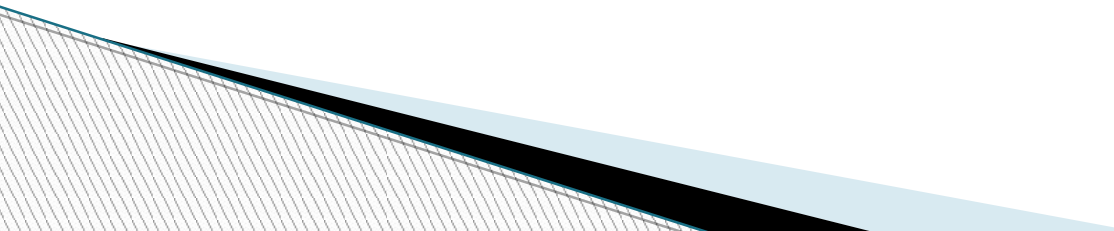
richardj@microsoft.com

switech@microsoft.com

Dynamic Memory Management

- ▶ The memory manager is responsible for tracking a program's dynamic data storage.
- ▶ Unlike stacks which work based upon a simple FIFO/LIFO concepts, heaps require management routines to track the location of free and allocated memory chunks

Dynamic Memory Management

- ▶ What approaches to dynamic memory management have been developed?
 - ▶ What are the security profiles of memory managers used in mainstream OS's today?
 - ▶ What is the impact of security research on memory manager design?
- 

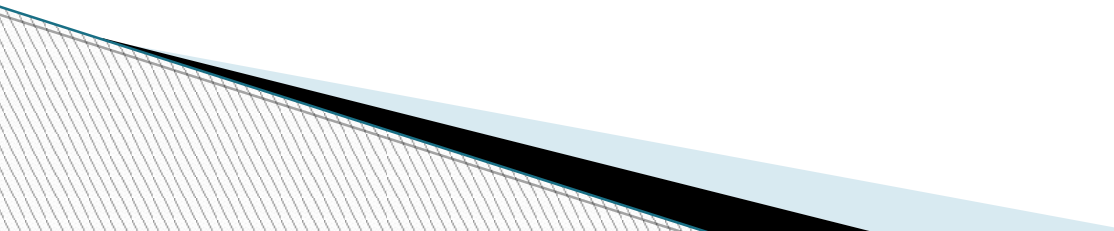
Dynamic Memory Management

- ▶ Today we will consider the following OS's and their memory allocators:
 - Windows
 - Linux
 - Apple OS X
 - OpenBSD

Dynamic Memory Management

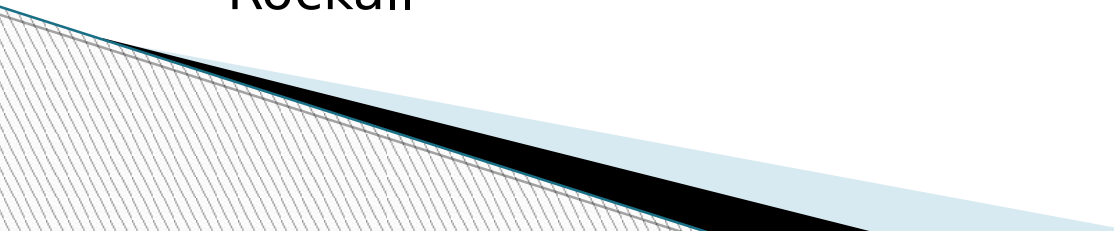
- ▶ Today we will consider the following OS's and their memory allocators:
 - Windows
 - ▢ Windows Heap Manager
 - ▢ Rockall Allocator
 - Linux
 - ▢ Doug Lea Malloc
 - Apple OS X
 - ▢ Poul-Henning Kamp Malloc
 - OpenBSD
 - ▢ OpenBSD Malloc

What's the Difference?

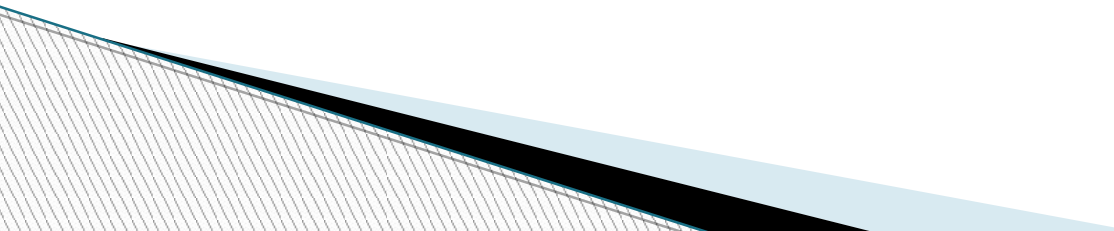
- ▶ The primary difference between the memory managers is how they track free buffers
 - ▶ We will split them into systems that inline management data on each chunk and those that do not
 - ▶ Management data inlined in the heap is susceptible to modification when a memory corruption occurs
- 

What's the Difference?

- ▶ Heaps with inlined management structs expose user APIs that walk linked lists of buffers to locate the appropriate buffer
 - Doug Lea
 - Windows Heap Manager

 - ▶ Heaps without inlined management data try to take advantage of kernel-supplied memory management APIs and utilize array indexing to locate buffers
 - Poul-Henning Kamp
 - OpenBSD Malloc
 - Rockall
- 

Security Research on Heap Allocators

- ▶ Offensive security researchers focus on adding reliability to exploitation methods or finding new ways to manipulate management routines to gain controllable memory corruption
 - ▶ Defensive security researchers aim to mitigate known attacks or (rarely) attempt new heap manager designs
- 

Security Research on Heap Allocators

- ▶ dmalloc
 - 2001 Michel "MaXX" Kaempf / Anonymous
 - 2005 Phantasmal Phantasmagoria

- ▶ Windows Heap
 - 2002 David Litchfield
 - 2004 Matt Conover / Oded Horovitz
 - 2005 SecurityPatrol

Security Research on Heap Allocators

- ▶ PHKMalloc
 - 2005 Yves Younan et al

- ▶ OpenBSD Malloc
 - 2006 Ben Hawkes

Heaps with inline data

▶ Basic mechanics:

- ▢ A region of memory is allocated to contain buffers
- ▢ An array of doubly linked lists tracking free buffers in multiples of a fixed size (usually 8) is created
- ▢ On allocation a free chunk is unlinked from the doubly linked list and the address is returned to the program
- ▢ On free, a 8 byte header is written to the beginning of a buffer and the chunk is added back to the list
- ▢ When two free buffers are adjacent they will be merged into one larger chunk of free memory
- ▢ Lookaside lists*

Heaps with inline data

▶ Attacks

◦ Unlink

- ▢ Free buffer is removed from doubly linked list with corrupted forward and backward pointers
- ▢ Attacker writes 4 bytes of controlled data to a controlled location

◦ Coalesce

- ▢ Manipulating the flag indicating whether the previous chunk is in use can be used with a fake chunk header to cause a 4 byte write to a controlled location

◦ Lookaside list

- ▢ The head of a lookaside list can be overwritten to later return a controlled address to the next allocation of that size

Heaps with inline data

▶ Unlink Attack

- Scenario: Heap-based buffer overflow allows for writing into adjacent free heap block
- Attack: Overwrite FLINK and BLINK values and wait for next allocation

```
mov dword ptr [ecx],eax  
mov dword ptr [eax+4],ecx  
EAX = Flink, ECX = Blink
```

FREE HEAP BLOCK

```
_HEAP_ENTRY  
+0x000 Size  
+0x002 PreviousSize  
+0x004 SmallTagIndex  
+0x005 Flags  
+0x006 UnusedBytes  
+0x007 SegmentIndex  
_LIST_ENTRY  
+0x000 Flink  
+0x004 Blink
```

- Result: Allows one or more 4-byte writes to controlled locations

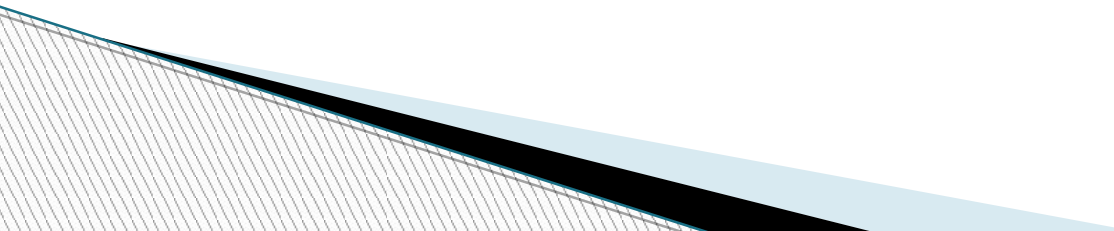
Heaps with inline data

▶ Lookaside Attack

- Scenario: Heap-based buffer overflow allows for control of lookaside list management structure
- Attack: First heap overwrite takes control of Flink value in a free chunk with a lookaside list entry
Allocation of the corrupted chunk puts the corrupt Flink value into the lookaside list
Next HeapAlloc() of the same sized chunk will return the corrupted pointer
- Result: Returns corrupted pointer from the next allocation from the lookaside list which allows for arbitrary length overwrites

Heaps without inline data

- ▶ Basic mechanics:

- ▢ Relies on and optimized for kernel provided virtual memory management system
 - ▢ Heap manager tracks allocated pages, allocated chunks and free pages in a series of directories
 - ▢ All chunks in a page are typically of the same size
 - ▢ Adjacent free pages are coalesced
- 

Heaps without inline data

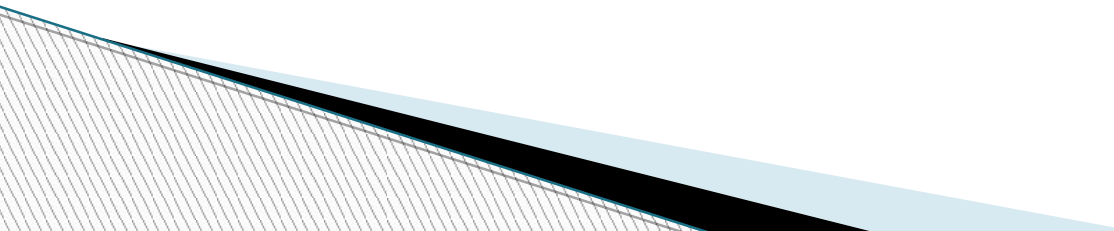
▶ Attacks

- free()

- ▢ Control of a pointer passed to free can be abused to free memory that contains one of the heap management structures.

- pginfo / pgfree

- ▢ Manipulate the value returned by an allocation



Heaps without inline data

- ▶ free() attack

- Scenario: Heap-based buffer overflow allows for control of pointers later passed to free()
- Attack: Free pages with control structures on them
- Result: Later allocations will eventually return the page with the control structures and allow for further exploitation

Heaps without inline data

▶ pginfo attack

- Scenario: Heap-based buffer overflow allows for control of the pginfo structure leading to arbitrary memory corruption
- Attack: Heap overflow allows for modification of the pginfo->free page pointer.
Overwrite bits array to make pages seem free
- Result: Allocation requests walk the structs to find the appropriate sized buffers so returning corrupted pointer allows for writes to arbitrary locations.

PGFREE

```
struct pgfree {
    struct pgfree *next;
    struct pgfree *prev;
    // free pages
    void *page;
    // base page dir
    void *pdir;
    // bytes free
    size_t size;
};
```

PGINFO

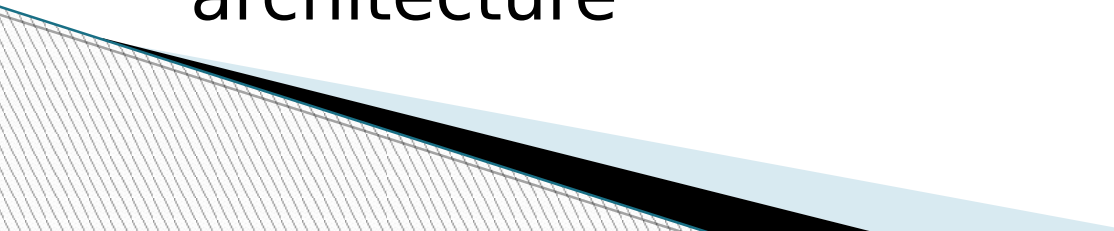
```
struct pginfo {
    struct pginfo *next;
    void *page;
    ushort size;
    ushort shift;
    ushort free;
    ushort total;
    uint bits[];
};
```

Heap Allocator Defense

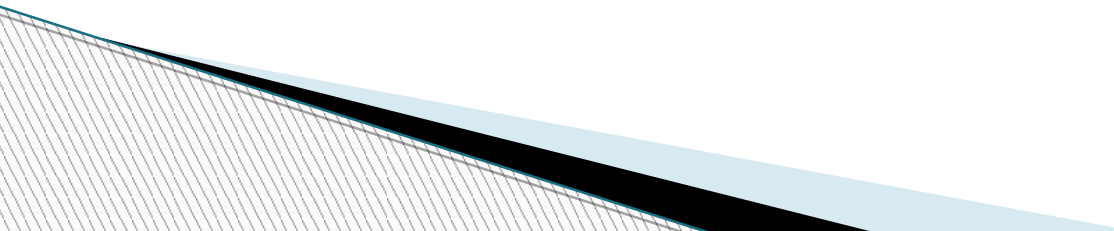
- ▶ **dlmalloc**
 - glibc added safe unlinking

- ▶ **Windows Heap**
 - Safe unlinking
 - Checksum for size and flags
 - XOR size, flags, checksum, and prevsize fields
 - Lookaside list replaced by LFH in Vista

Heap Allocator Defense

- ▶ phkmalloc
 - Nada
 - ▶ OpenBSD malloc
 - Nada
 - ▶ System defenses such as ASLR and NX also apply but are not part of the heap manager's architecture
- 

So what's next?



Windows Kernel Pool Manager

“The Month of Kernel Bugs is a serious wake-up call about the vulnerability of the most fundamental element of the operating system. Begin preparing now for more, and more damaging, attacks against the OS kernel.”

Rich Mogul - Gartner Nov. 2006

http://www.gartner.com/resources/144700/144700/learn_from_month_of_kernel_b_144700.pdf

Windows Kernel Pool Manager

- ▶ 2005 SoBelt “How to exploit Windows kernel memory pool”
- ▶ Basic unlink() technique applies to the kernel pool

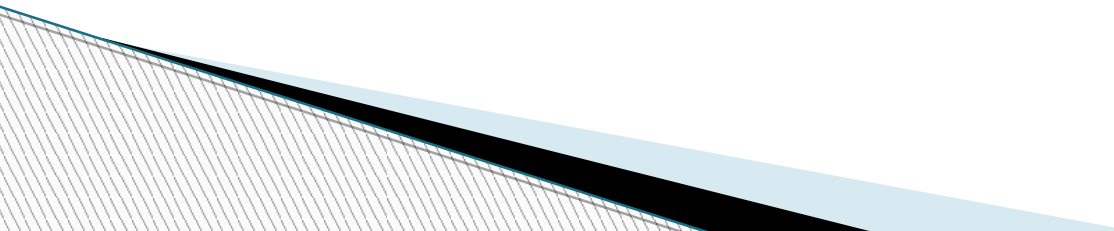
Windows Kernel Pool Manager

- ▶ Pools are managed by a pool descriptor, chunks are managed by a pool chunk header

```
lkd> dt -v -r nt!POOL_DESCRIPTOR
struct _POOL_DESCRIPTOR, 14 elements, 0x1034 bytes
+0x000 PoolType           : Enum _POOL_TYPE
+0x004 PoolIndex          : Uint4B
+0x008 RunningAllocs     : Int4B
+0x00c RunningDeAllocs   : Int4B
+0x010 TotalPages        : Int4B
+0x014 TotalBigPages     : Int4B
+0x018 Threshold         : Uint4B
+0x01c LockAddress       : Ptr32 to Void
+0x020 PendingFrees      : Ptr32 to Ptr32 to Void
+0x024 ThreadsProcessingDeferrals : Int4B
+0x028 PendingFreeDepth : Int4B
+0x02c TotalBytes        : Uint4B
+0x030 Spare0            : Uint4B
+0x034 ListHeads         : [512] struct _LIST_ENTRY, 2 elements, 0x8 bytes
    +0x000 Flink          : Ptr32 to struct _LIST_ENTRY, 2 elements, 0x8 bytes
        +0x000 Flink      : Ptr32 to struct _LIST_ENTRY, 2 elements, 0x8 bytes
        +0x004 Blink      : Ptr32 to struct _LIST_ENTRY, 2 elements, 0x8 bytes
    +0x004 Blink          : Ptr32 to struct _LIST_ENTRY, 2 elements, 0x8 bytes
        +0x000 Flink      : Ptr32 to struct _LIST_ENTRY, 2 elements, 0x8 bytes
        +0x004 Blink      : Ptr32 to struct _LIST_ENTRY, 2 elements, 0x8 bytes
```

```
lkd> dt -v -r nt!POOL_HEADER
struct _POOL_HEADER, 8 elements, 0x8 bytes
+0x000 PreviousSize      : Bitfield Pos 0, 9 Bits
+0x000 PoolIndex         : Bitfield Pos 9, 7 Bits
+0x002 BlockSize         : Bitfield Pos 0, 9 Bits
+0x002 PoolType          : Bitfield Pos 9, 7 Bits
+0x000 Ulong1            : Uint4B
+0x004 PoolTag           : Uint4B
+0x004 AllocatorBackTraceIndex : Uint2B
+0x006 PoolTagHash       : Uint2B
```


Windows Kernel Pool Manager

- ▶ The good news
 - We're active researching how to add appropriate mitigations to the kernel memory management code
 - ▶ The bad news
 - Unlike user heaps, the kernel pool is globally managed
 - There aren't any free bytes to use for checksums and cookies
 - Performance and compatibility concerns sometimes trump security
- 

Windows Kernel Pool Manager

- ▶ You can help. Contact us at switech@microsoft.com if you are interested in this research and want your ideas heard!

Questions?

