Scoop the Windows 10 pool!

Corentin Bayet and Paul Fariello
corentin.bayet@synacktiv.com
paul.fariello@synacktiv.com

Synacktiv

Abstract. Heap Overflow are a fairly common vulnerability in applications. Exploiting such vulnerabilities often rely on a deep understanding of the underlying mechanisms used to manage the heap. Windows 10 recently changed the way it managed its heap in kernel land. This article aims to present the recent evolution of the heap mechanisms in Windows NT Kernel and to present new exploitation techniques specific to the kernel Pool.

1 Introduction

The pool is the heap reserved to the kernel land on Windows systems. For years, the pool allocator has been very specific and different from the allocator in user land. This has changed since the `19H1` update of Windows 10, in March 2019. The well-known and documented `Segment Heap` [7] used in user land has been brought to the kernel.

However, some differences remain between the allocator implemented in the kernel and in user land, since there are still some specific materials required in kernel land. This paper focuses on the internals that are custom to the kernel `Segment Heap` from an exploitation point of view.

The research presented in this paper is tailored to the **x64** architecture. The adjustment needed for different architectures has not been studied.

After a quick reminder of the historic pool internals, the paper will explain how the `Segment Heap` is implemented in the kernel, and the impact it had on the materials specific to the kernel pool. Then, the paper will present a new attack on the pool internals when exploiting a heap overflow vulnerability in the kernel pool. Finally, a generic exploit using a minimal controled heap overflow and allowing a local privilege escalation from a Low Integrity level to SYSTEM will be presented.

1.1 Pool internals

This paper will not go too deep on the internals of the pool allocator, since this subject has already been widely covered [5] , but for a full understanding of the paper, a quick reminder of some internals is nonetheless required.

This section will present a few pool internals as they were in Windows 7 as well as the various mitigations and changes brought to the pool during the past few years. The internals explained here will focus on chunks that fit in a single page, which are the most common allocation in the kernel. The allocations with a size greater than `0xFE0` behave differently and are not the subject covered here.

Allocating memory in the pool The main functions for allocating and freeing memory in the Windows kernel are respectively **ExAllocatePoolWithTag** and **ExFreePoolWithTag**.

```
void * ExAllocatePoolWithTag(POOL_TYPE      PoolType,
                             size_t         NumberOfBytes,
                             unsigned int   Tag);
```

Fig. 1. ExAllocatePoolWithTag prototype

```
void ExFreePoolWithTag(void * P, unsigned int Tag);
```

Fig. 2. ExFreePoolWithTag prototype

The `PoolType` is a bitfield, with this associated enumeration:

```
NonPagedPool                            = 0
PagedPool                               = 1
NonPagedPoolMustSucceed                 = 2
DontUseThisType                         = 3
NonPagedPoolCacheAligned                = 4
PagedPoolCacheAligned                   = 5
NonPagedPoolCacheAlignedMustSucceed     = 6
MaxPoolType                             = 7
PoolQuota                               = 8
NonPagedPoolSession                     = 20h
PagedPoolSession                        = 21h
NonPagedPoolMustSucceedSession          = 22h
DontUseThisTypeSession                  = 23h
NonPagedPoolCacheAlignedSession         = 24h
PagedPoolCacheAlignedSession            = 25h
NonPagedPoolCacheAlignedMustSSession    = 26h
```

```
NonPagedPoolNx                          = 200h
NonPagedPoolNxCacheAligned              = 204h
NonPagedPoolSessionNx                   = 220h
```

Several information can be stored in the `PoolType`:

- the type of the memory used, which can be `NonPagedPool`, `PagedPool`, `SessionPool` or `NonPagedPoolNx`;
- if the allocation is critical (bit 1) and must succeed. If the allocation fails, it triggers a `BugCheck`;
- if the allocation is aligned on the cache size (bit 2);
- if the allocation is using the `PoolQuota` mechanism (bit 3);
- others undocumented mechanisms.

The type of memory used is important because it isolates allocations in different memory ranges. The two main types of memory used are the `PagedPool` and `NonPagedPool`. The MSDN documentation describes it as following:

"Nonpaged pool is nonpageable system memory. It can be accessed from any IRQL, but it is a scarce resource and drivers should allocate it only when necessary. Paged pool is pageable system memory and can only be allocated and accessed at IRQL < DISPATCH_LEVEL."

As explained in section 1.2, the `NonPagedPoolNx` has been introduced in Windows 8 and must be used instead of the `NonPagedPool`.

The `SessionPool` is used for session space allocations and is unique to each user session. It's mainly used by `win32k`.

Finally, the tag is a non-zero character literal of one to four characters (for example, 'Tag1'). It is recommended for kernel developers to use a unique pool tag by code path to help debuggers and verifiers identify the code path.

The POOL_HEADER In the pool, all chunks that fit in a single page begin with a `POOL_HEADER` structure. This header contains information required by the allocator, and the tag. When trying to exploit a heap overflow vulnerability in the Windows kernel, the first thing to be overwritten is the `POOL_HEADER` structure. Two options are available for an attacker: properly rewrite the `POOL_HEADER` structure and attack the data of the next chunk, or directly attack the `POOL_HEADER` structure.

In both cases, the POOL_HEADER structure will be overwritten, and a good understanding of each field and how it is used is necessary to be able to exploit this kind of vulnerability. This paper will focus on the attacks directly aimed at the POOL_HEADER.

```
struct POOL_HEADER
{
  char  PreviousSize;
  char  PoolIndex;
  char  BlockSize;
  char  PoolType;
  int   PoolTag;
  Ptr64 ProcessBilled;
};
```

Fig. 3. Simplified POOL_HEADER structure in Windows 1809

The POOL_HEADER structure, presented in figure 3, has slightly evolved over time but always kept the same main fields. In Windows 1809, Before Windows 19H1, all fields were used :

PreviousSize is the size of the previous chunk divided by 16;
PoolIndex is an index in an array of PoolDescriptor;
BlockSize is the size of the current allocation divided by 16;
PoolType is a bitfield containing information on the allocation type;
ProcessBilled is a pointer to the KPROCESS that made the allocation. It is set only if the PoolQuota Flag is set in the PoolType.

1.2 Attacks and mitigations since Windows 7

Tarjei Mandt and its paper *Kernel Pool Exploitation on Windows 7* [5] is the reference about the attacks targeting the kernel pool. It presented the entire pool internals and numerous attacks, and some targeting the POOL_HEADER.

Quota Process Pointer Overwrite Allocation can be charging the quota against a given process. To do so, the ExAllocatePoolWithQuotaTag will leverage the ProcessBilled field of the POOL_HEADER to store a pointer to the _KPROCESS charged with the allocation.

An attack described in the paper is the `Quota Process Pointer Overwrite`. This attack uses an heap overflow to overwrite the `ProcessBilled` pointer of an allocated chunk. When the chunk is freed, if the `PoolType` of the chunk contains the `PoolQuota` flag (0x8), the pointer is used to dereference a value. Controlling this pointer provides an arbitrary dereference primitive, which is enough to elevate privileges from user land. Figure 4 present this attack.

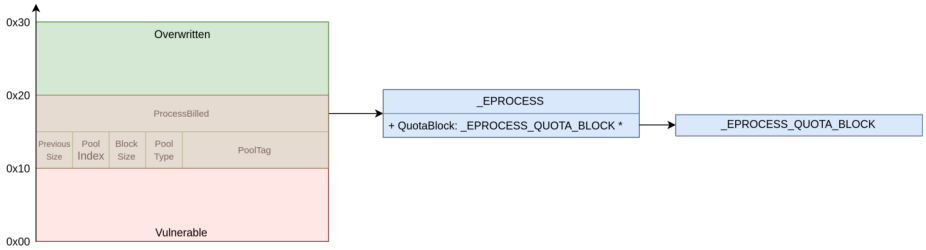

Fig. 4. Exploitation of a Quota Process Pointer Overwrite

This attack has been mitigated since Windows 8, with the introduction of the `ExpPoolQuotaCookie`. This cookie is randomly generated at boot and is used to protect pointers from being overwritten by an attacker. For example, it is used to XOR the `ProcessBilled` field:

```
ProcessBilled = KPROCESS_PTR ^ ExpPoolQuotaCookie ^ CHUNK_ADDR
```

When the chunk is freed, the kernel checks that the encoded pointer is a valid KPROCESS pointer:

```
process_ptr = (struct _KPROCESS *)(chunk_addr ^ ExpPoolQuotaCookie ^
    chunk_addr->process_billed);
if ( process_ptr )
{
    if (process_ptr < 0xFFFF800000000000 || (process_ptr->Header.
        Type & 0x7F) != 3 )
          KeBugCheckEx([...])
    [...]
}
```

Without knowing the address of the chunk nor the value of the `ExpPoolQuotaCookie`, it is impossible to provide a valid pointer, and

to obtain an arbitrary dereference. It is however still possible to properly rewrite the POOL_HEADER and do a full data attack by not setting the PoolQuota flag in the PoolType. For more information on the Quota Process Pointer Overwrite attack, it has been covered in a conference at Nuit du Hack XV [1].

NonPagedPoolNx With Windows 8, a new kind of pool memory type has been introduced: NonPagedPoolNx. It works exactly like NonPagedPool, except that the memory pages are not executable anymore, mitigating all exploits using this kind of memory to store shellcodes.

The allocations that were previously done in the NonPagedPool are now using the NonPagedPoolNx, but the NonPagedPool type was kept for compatibility reasons with the third-party drivers. Even today in Windows 10, a lot of third-party drivers are still using the executable NonPagedPool.

The various mitigations introduced overtime made the POOL_HEADER not interesting to attack using a heap overflow. Nowadays, it is simpler to properly rewrite the POOL_HEADER and attack the data of the next chunk. However, the introduction of the Segment Heap in the pool has changed how the POOL_HEADER is used, and this paper shows how it can be attacked again to exploit a heap overflow in the kernel pool.

2 The Pool Allocator with the Segment Heap

2.1 Segment Heap internals

The Segment Heap is used in kernel land since Windows 10 19H1 and is quite similar to the Segment Heap used in user land. This section aims to present the main features of the Segment Heap and to focus on the differences with the one used in user land. A very detailed explanation of the internals of the user land Segment Heap is available in [7].

Just as for the one used in user land, the Segment Heap aims at providing different features depending on the size of the allocations. To do so, four so-called backends are defined.

- Low Fragmentation Heap (abbr LFH): RtlHpLfhContextAllocate
- Variable Size (abbr VS): RtlHpVsContextAllocateInternal
- Segment Alloc (abbr Seg): RtlHpSegAlloc
- Large Alloc: RtlHpLargeAlloc

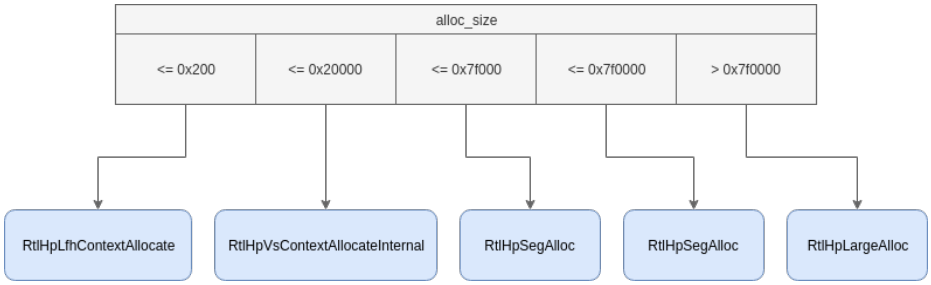

Fig. 5. Mapping between Allocation size and backend

The mapping between the requested allocation size and the chosen backend is shown in figure 5.

The three first backends, `Seg`, `VS` and `LFH`, are associated with a context, respectively: `_HEAP_SEG_CONTEXT`, `_HEAP_VS_CONTEXT` and `_HEAP_LFH_CONTEXT`. Backend contexts are stored in the `_SEGMENT_HEAP` structure.

```
1: kd> dt nt!_SEGMENT_HEAP
   +0x000 EnvHandle         : RTL_HP_ENV_HANDLE
   +0x010 Signature         : Uint4B
   +0x014 GlobalFlags       : Uint4B
   +0x018 Interceptor       : Uint4B
   +0x01c ProcessHeapListIndex : Uint2B
   +0x01e AllocatedFromMetadata : Pos 0, 1 Bit
   +0x020 CommitLimitData   : _RTL_HEAP_MEMORY_LIMIT_DATA
   +0x020 ReservedMustBeZero1 : Uint8B
   +0x028 UserContext       : Ptr64 Void
   +0x030 ReservedMustBeZero2 : Uint8B
   +0x038 Spare             : Ptr64 Void
   +0x040 LargeMetadataLock : Uint8B
   +0x048 LargeAllocMetadata : _RTL_RB_TREE
   +0x058 LargeReservedPages : Uint8B
   +0x060 LargeCommittedPages : Uint8B
   +0x068 StackTraceInitVar : _RTL_RUN_ONCE
   +0x080 MemStats          : _HEAP_RUNTIME_MEMORY_STATS
   +0x0d8 GlobalLockCount   : Uint2B
   +0x0dc GlobalLockOwner   : Uint4B
   +0x0e0 ContextExtendLock : Uint8B
   +0x0e8 AllocatedBase     : Ptr64 UChar
   +0x0f0 UncommittedBase   : Ptr64 UChar
   +0x0f8 ReservedLimit     : Ptr64 UChar
   +0x100 SegContexts       : [2] _HEAP_SEG_CONTEXT
   +0x280 VsContext         : _HEAP_VS_CONTEXT
   +0x340 LfhContext        : _HEAP_LFH_CONTEXT
```

5 such structures exist, corresponding to different `_POOL_TYPE` values:

– `NonPaged` pools (bit 0 unset)

– `NonPagedNx` pool (bit 0 unset and bit 9 set)
– `Paged` pools (bit 0 set)
– `PagedSession` pool (bit 5 and 1 set)

A fifth `_SEGMENT_HEAP` is allocated but the authors could not find its purpose. The 3 firsts `_SEGMENT_HEAP`, corresponding to `NonPaged`, `NonPagedNx` and `Paged` pools, are stored in the `HEAP_POOL_NODES`. As for `PagedPoolSession` the corresponding `_SEGMENT_HEAP` is stored in the current thread. The figure 6 summarizes the five `_SEGMENT_HEAP`.

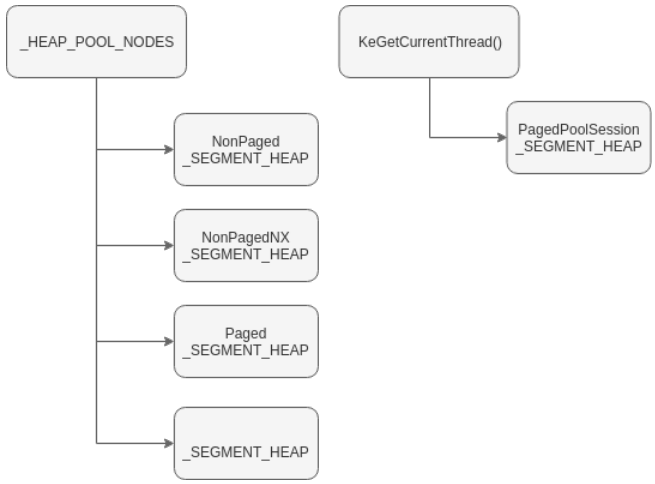

Fig. 6. Segment backend internal structures

Although the user land `Segment Heap` uses only one `Segment Allocation` context for allocations between 128 KiB and 508 KiB, in kernel land the `Segment Heap` uses 2 `Segment Allocation` contexts. The second one is used for allocations between 508 KiB and 7 GiB.

Segment Backend

The segment backend is used to allocate memory chunks of size between 128 KiB and 7 GiB. It is also used behind the scene, to allocate memory for `VS` and `LFH` backends.

The Segment Backend context is stored in a structure called `_HEAP_SEG_CONTEXT`.

```
1: kd> dt nt!_HEAP_SEG_CONTEXT
```

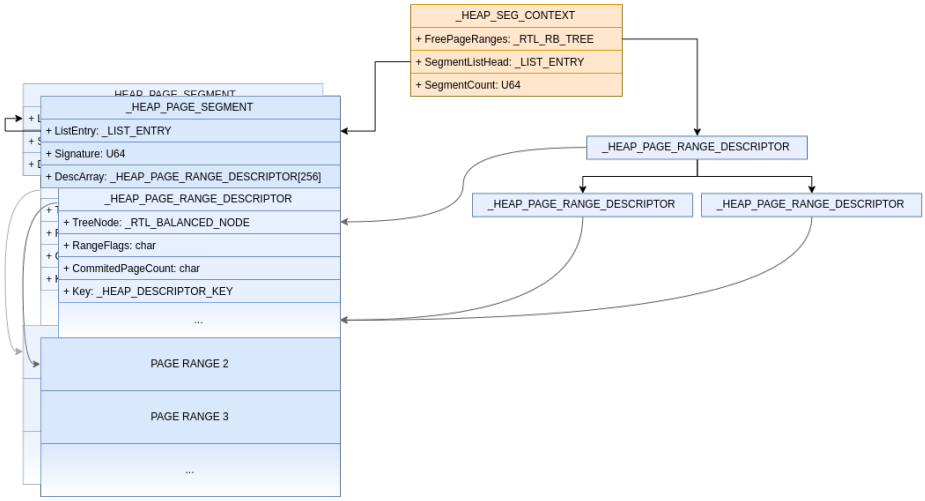

Fig. 7. Segment backend internal structures

```
   +0x000 SegmentMask      : Uint8B
   +0x008 UnitShift        : UChar
   +0x009 PagesPerUnitShift : UChar
   +0x00a FirstDescriptorIndex : UChar
   +0x00b CachedCommitSoftShift : UChar
   +0x00c CachedCommitHighShift : UChar
   +0x00d Flags            : <anonymous-tag>
   +0x010 MaxAllocationSize : Uint4B
   +0x014 OlpStatsOffset   : Int2B
   +0x016 MemStatsOffset   : Int2B
   +0x018 LfhContext       : Ptr64 Void
   +0x020 VsContext        : Ptr64 Void
   +0x028 EnvHandle        : RTL_HP_ENV_HANDLE
   +0x038 Heap             : Ptr64 Void
   +0x040 SegmentLock      : Uint8B
   +0x048 SegmentListHead  : _LIST_ENTRY
   +0x058 SegmentCount     : Uint8B
   +0x060 FreePageRanges   : _RTL_RB_TREE
   +0x070 FreeSegmentListLock : Uint8B
   +0x078 FreeSegmentList  : [2] _SINGLE_LIST_ENTRY
```

The `Segment Backend` allocates memory by chunk of variable sizes called segments. Each segment is composed of multiple allocatable pages.

Segments are stored within a linked list stored in `SegmentListHead`. Segments are headed with a `_HEAP_PAGE_SEGMENT` followed by 256 `_HEAP_PAGE_RANGE_DESCRIPTOR` structures.

```
1: kd> dt nt!_HEAP_PAGE_SEGMENT
   +0x000 ListEntry        : _LIST_ENTRY
```

```
   +0x010 Signature        : Uint8B
   +0x018 SegmentCommitState : Ptr64 _HEAP_SEGMENT_MGR_COMMIT_STATE
   +0x020 UnusedWatermark  : UChar
   +0x000 DescArray        : [256] _HEAP_PAGE_RANGE_DESCRIPTOR
```

```
1: kd> dt nt!_HEAP_PAGE_RANGE_DESCRIPTOR
   +0x000 TreeNode          : _RTL_BALANCED_NODE
   +0x000 TreeSignature     : Uint4B
   +0x004 UnusedBytes       : Uint4B
   +0x008 ExtraPresent      : Pos 0, 1 Bit
   +0x008 Spare0            : Pos 1, 15 Bits
   +0x018 RangeFlags        : UChar
   +0x019 CommittedPageCount : UChar
   +0x01a Spare             : Uint2B
   +0x01c Key               : _HEAP_DESCRIPTOR_KEY
   +0x01c Align             : [3] UChar
   +0x01f UnitOffset        : UChar
   +0x01f UnitSize          : UChar
```

In order to provide fast lookup for free page ranges, a Red-Black tree is also maintained in `_HEAP_SEG_CONTEXT`.

Each `_HEAP_PAGE_SEGMENT` has a signature computed as follow:

```
Signature = Segment ^ SegContext ^ RtlpHpHeapGlobals ^ 0
    xA2E64EADA2E64EAD;
```

This signature is used to retrieve the owning `_HEAP_SEG_CONTEXT` and the corresponding `_SEGMENT_HEAP` from any allocated memory chunk.

Figure 7 summarizes the internal structures used in the segment backend.

The original segment can easily be computed from any address by masking it with the `SegmentMask` stored in the `_HEAP_SEG_CONTEXT`. `SegmentMask` is valued `0xfffffffffff00000`.

```
  Segment = Addr & SegContext->SegmentMask;
```

The corresponding PageRange can easily be computed from any address by using the `UnitShift` from the `_HEAP_SEG_CONTEXT`. `UnitShift` is set to 12.

```
  PageRange = Segment + sizeof(_HEAP_PAGE_RANGE_DESCRIPTOR) * (Addr
      - Segment) >> SegContext->UnitShift;
```

When the `Segment Backend` is used by one of the other backend, the fields `RangeFlags` of the `_HEAP_PAGE_RANGE_DESCRIPTOR` are used to store which backend requested the allocation.

Variable Size Backend

Variable Size backend allocate chunk of sizes between 512 B and 128 KiB. It aims at providing easy reuse of free chunk.

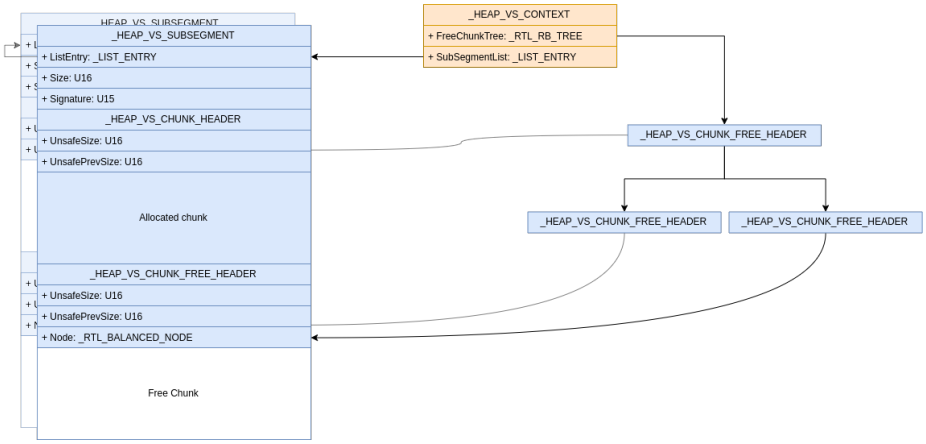

Fig. 8. Variable Size backend internal structures

The Variable Size Backend context is stored in a structure called `_HEAP_VS_CONTEXT`.

```
0: kd> dt nt!_HEAP_VS_CONTEXT
   +0x000 Lock              : Uint8B
   +0x008 LockType          : _RTLP_HP_LOCK_TYPE
   +0x010 FreeChunkTree     : _RTL_RB_TREE
   +0x020 SubsegmentList    : _LIST_ENTRY
   +0x030 TotalCommittedUnits : Uint8B
   +0x038 FreeCommittedUnits : Uint8B
   +0x040 DelayFreeContext  : _HEAP_VS_DELAY_FREE_CONTEXT
   +0x080 BackendCtx        : Ptr64 Void
   +0x088 Callbacks         : _HEAP_SUBALLOCATOR_CALLBACKS
   +0x0b0 Config            : _RTL_HP_VS_CONFIG
   +0x0b4 Flags             : Uint4B
```

Free chunks are stored in a Red-Black tree called `FreeChunkTree`. When an allocation is requested, the Red-Black tree is used to find any free chunk of the exact size or the first free chunk bigger than the requested size.

The freed chunk are headed with a dedicated struct called `_HEAP_VS_CHUNK_FREE_HEADER`.

```
0: kd> dt nt!_HEAP_VS_CHUNK_FREE_HEADER
   +0x000 Header           : _HEAP_VS_CHUNK_HEADER
   +0x000 OverlapsHeader   : Uint8B
   +0x008 Node             : _RTL_BALANCED_NODE
```

Once a free chunk is found, it is split to the right size with a call to `RtlpHpVsChunkSplit`.

The allocated chunk are all headed with a dedicated struct called `_HEAP_VS_CHUNK_HEADER`.

```
0: kd> dt nt!_HEAP_VS_CHUNK_HEADER
   +0x000 Sizes            : _HEAP_VS_CHUNK_HEADER_SIZE
   +0x008 EncodedSegmentPageOffset : Pos 0, 8 Bits
   +0x008 UnusedBytes      : Pos 8, 1 Bit
   +0x008 SkipDuringWalk   : Pos 9, 1 Bit
   +0x008 Spare            : Pos 10, 22 Bits
   +0x008 AllocatedChunkBits : Uint4B
0: kd> dt nt!_HEAP_VS_CHUNK_HEADER_SIZE
   +0x000 MemoryCost       : Pos 0, 16 Bits
   +0x000 UnsafeSize       : Pos 16, 16 Bits
   +0x004 UnsafePrevSize   : Pos 0, 16 Bits
   +0x004 Allocated        : Pos 16, 8 Bits
   +0x000 KeyUShort        : Uint2B
   +0x000 KeyULong         : Uint4B
   +0x000 HeaderBits       : Uint8B
```

All fields inside this header are xored with `RtlpHpHeapGlobals` and the address of the chunk.

```
Chunk->Sizes = Chunk->Sizes ^ Chunk ^ RtlpHpHeapGlobals;
```

Internally, `VS` allocator uses the `Segment` allocator. It is used in `RtlpHpVsSubsegmentCreate` through the `_HEAP_SUBALLOCATOR_CALLBACKS` field of the `_HEAP_VS_CONTEXT`. The suballocator callbacks are all xored with the addresses of the `VS` context and of `RtlpHpHeapGlobals`.

```
  callbacks.Allocate = RtlpHpSegVsAllocate;
  callbacks.Free = RtlpHpSegLfhVsFree;
  callbacks.Commit = RtlpHpSegLfhVsCommit;
  callbacks.Decommit = RtlpHpSegLfhVsDecommit;
  callbacks.ExtendContext = NULL;
```

If no chunk, big enough, is present in the `FreeChunkTree` a new `Subsegment`, whose size range from 64 KiB to 256 KiB, is allocated and inserted in the `SubsegmentList`. It is headed with `_HEAP_VS_SUBSEGMENT`

structure. All the remaining space is used as a free chunk and inserted in the `FreeChunkTree`.

```
0: kd> dt nt!_HEAP_VS_SUBSEGMENT
   +0x000 ListEntry         : _LIST_ENTRY
   +0x010 CommitBitmap      : Uint8B
   +0x018 CommitLock        : Uint8B
   +0x020 Size              : Uint2B
   +0x022 Signature         : Pos 0, 15 Bits
   +0x022 FullCommit        : Pos 15, 1 Bit
```

Figure 8 summarize the memory organisation of the VS Backend.

When a VS chunk is freed, if it's smaller than 1 KiB and the VS backend as been configured correctly (bit 4 of `Config.Flags` set to 1) it is temporarily stored in a list inside the `DelayFreeContext`. Once the `DelayFreeContext` is filled with 32 chunks they are all really freed at once. The `DelayFreeContext` is never used for direct allocation.

When a VS chunk is really freed, if it is contiguous with 2 other freed chunks, all 3 will be merged together with a call to `RtlpHpVsChunkCoalesce`. Then it will be inserted into the `FreeChunkTree`.

Low Fragmentation Heap Backend

`Low Fragmentation Heap` is a backend dedicated to small allocations from 1 B to 512 B.

The LFH Backend context is stored in a structure called `_HEAP_LFH_CONTEXT`.

```
0: kd> dt nt!_HEAP_LFH_CONTEXT
   +0x000 BackendCtx        : Ptr64 Void
   +0x008 Callbacks         : _HEAP_SUBALLOCATOR_CALLBACKS
   +0x030 AffinityModArray  : Ptr64 UChar
   +0x038 MaxAffinity       : UChar
   +0x039 LockType          : UChar
   +0x03a MemStatsOffset    : Int2B
   +0x03c Config            : _RTL_HP_LFH_CONFIG
   +0x040 BucketStats       : _HEAP_LFH_SUBSEGMENT_STATS
   +0x048 SubsegmentCreationLock : Uint8B
   +0x080 Buckets           : [129] Ptr64 _HEAP_LFH_BUCKET
```

The main feature of the LFH backend is to use buckets of different sizes to avoid fragmentation.

Bucket	Allocation Size	Bucket granularity
1 – 64	1 B – 1008 B	16 B
65 – 80	1009 B – 2032 B	64 B
81 – 96	2033 B – 4080 B	128 B
97 – 112	4081 B – 8176 B	256 B
113 – 128	8177 B – 16 368 B	512 B

Each bucket is composed of SubSegments allocated by the segment allocator. The segment allocator is used through the `_HEAP_SUBALLOCATOR_CALLBACKS` field of the `_HEAP_LFH_CONTEXT`. The suballocator callbacks are all xored with the addresses of the `LFH` context and of `RtlpHpHeapGlobals`.

```
  callbacks.Allocate = RtlpHpSegLfhAllocate;
  callbacks.Free = RtlpHpSegLfhVsFree;
  callbacks.Commit = RtlpHpSegLfhVsCommit;
  callbacks.Decommit = RtlpHpSegLfhVsDecommit;
  callbacks.ExtendContext = RtlpHpSegLfhExtendContext;
```

LFH subsegment are headed with a `_HEAP_LFH_SUBSEGMENT` structure.

```
0: kd> dt nt!_HEAP_LFH_SUBSEGMENT
   +0x000 ListEntry         : _LIST_ENTRY
   +0x010 Owner             : Ptr64 _HEAP_LFH_SUBSEGMENT_OWNER
   +0x010 DelayFree         : _HEAP_LFH_SUBSEGMENT_DELAY_FREE
   +0x018 CommitLock        : Uint8B
   +0x020 FreeCount         : Uint2B
   +0x022 BlockCount        : Uint2B
   +0x020 InterlockedShort  : Int2B
   +0x020 InterlockedLong   : Int4B
   +0x024 FreeHint          : Uint2B
   +0x026 Location          : UChar
   +0x027 WitheldBlockCount : UChar
   +0x028 BlockOffsets      : _HEAP_LFH_SUBSEGMENT_ENCODED_OFFSETS
   +0x02c CommitUnitShift   : UChar
   +0x02d CommitUnitCount   : UChar
   +0x02e CommitStateOffset : Uint2B
   +0x030 BlockBitmap       : [1] Uint8B
```

Each subsegment is then split into different LFH blocks with corresponding bucket size.

In order to know which bucket are used, a bitmap is maintained in each SubSegment header.

When an allocation is requested, the LFH allocator will first look for the `FreeHint` field of the `_HEAP_LFH_SUBSEGMENT` structure in order to find the offset of the last freed block in the subsegment. Then it will scan

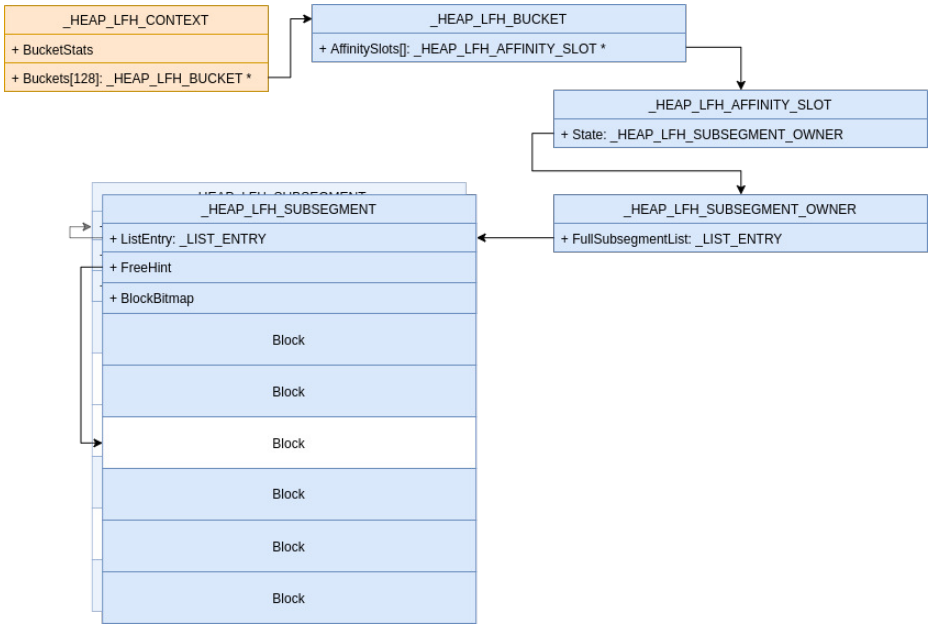

Fig. 9. Low Fragmentation Heap backend internal structures

the BlockBitmap, by group of 32 blocks, looking for a free block. This scan is randomized thanks to the `RtlpLowFragHeapRandomData` table.

Depending on the contention on a given bucket, a mechanism can be enable to ease allocation by dedicating SubSegment to each CPU. This mechanism is called Affinity Slot.

Figure 9 present the main architecture of the LFH backend.

Dynamic Lookaside

Freed chunk of size between 0x200 and 0xF80 bytes can be temporarily stored in a lookaside list in order to provide fast allocation. While they are in the lookaside these chunks wont go through their respective backend free mechanism.

Lookaside are represented by the `_RTL_DYNAMIC_LOOKASIDE` structure and are stored in the `UserContext` field of the `_SEGMENT_HEAP`.

```
0: kd> dt nt!_RTL_DYNAMIC_LOOKASIDE
   +0x000 EnabledBucketBitmap : Uint8B
   +0x008 BucketCount         : Uint4B
   +0x00c ActiveBucketCount   : Uint4B
   +0x040 Buckets             : [64] _RTL_LOOKASIDE
```

Each freed block is stored in a `_RTL_LOOKASIDE` corresponding to its size (as expressed in the `POOL_HEADER`). Size correspondance follows the same pattern as for Bucket in `LFH`.

```
0: kd> dt nt!_RTL_LOOKASIDE
   +0x000 ListHead          : _SLIST_HEADER
   +0x010 Depth             : Uint2B
   +0x012 MaximumDepth      : Uint2B
   +0x014 TotalAllocates    : Uint4B
   +0x018 AllocateMisses    : Uint4B
   +0x01c TotalFrees        : Uint4B
   +0x020 FreeMisses        : Uint4B
   +0x024 LastTotalAllocates : Uint4B
   +0x028 LastAllocateMisses : Uint4B
   +0x02c LastTotalFrees    : Uint4B
```

Free List	Allocation Size	Bucket granularity
1 – 32	512 B – 1024 B	16 B
33 – 48	1025 B – 2048 B	64 B
49 – 64	2049 B – 3967 B	128 B

Only a subset of the available buckets are enabled at the same time (field `ActiveBucketCount` of `_RTL_DYNAMIC_LOOKASIDE`). Each time an allocation is requested, metrics of the corresponding lookaside are updated.

Every 3 scan of the Balance Set Manager, the dynamic lookaside are rebalanced. The most used since last rebalance are enabled. The size of each lookaside depends on its usage but it can't be more than `MaximumDepth` or less than `4`. While the number of new allocation is less than 25 the depth is reduced by 10. Otherwhile, the depth is reduced by 1 if the miss ratio is lower than 0.5%, else it is grown with the following formula.

$$Depth = \frac{MissRatio(MaximumDepth - Depth)}{2} + 5$$

2.2 POOL_HEADER

As presented in section 1.1 the `POOL_HEADER` structure was heading all allocated chunks in the kernel land heap allocator predating Windows 10 `19H1`. All fields were used, back then. With the update of the kernel land heap allocator most of the fields of the `POOL_HEADER` are useless, yet small allocated memory are still headed with it.

The `POOL_HEADER` definition is recalled in figure 10.

```
struct POOL_HEADER
{
  char  PreviousSize;
  char  PoolIndex;
  char  BlockSize;
  char  PoolType;
  int   PoolTag;
  Ptr64 ProcessBilled;
};
```

Fig. 10. POOL_HEADER definition

The only fields set by the allocator are the following:

```
  PoolHeader->PoolTag = PoolTag;
  PoolHeader->BlockSize = BucketBlockSize >> 4;
  PoolHeader->PreviousSize = 0;
  PoolHeader->PoolType = changedPoolType & 0x6D | 2;
```

Here is a summary of the purpose of each of the `POOL_HEADER` fields since Windows 19H1.

PreviousSize Unused and kept to 0.
PoolIndex Unused.
BlockSize Size of the chunk. Only used to eventually store the chunk in the Dynamic Lookaside list (see 2.1).
PoolType Usage did not change ; used to keep the requested `POOL_TYPE`.
PoolTag Usage did not change ; used to keep the `PoolTag`.
ProcessBilled Usage did not change ; used to keep track of which process required the allocation, if the `PoolType` is `PoolQuota` (bit 3). The value is computed as follow:

```
    ProcessBilled = chunk_addr ^ ExpPoolQuotaCookie ^ KPROCESS;
```

CacheAligned

When calling **ExAllocatePoolWithTag**, if the `PoolType` has the `CacheAligned` bit set (bit 2), returned memory is aligned on the cache line size. The cache line size value is CPU dependent, but is typically 0x40.

First the allocator will grow the allocation size of `ExpCacheLineSize`:

```
if ( PoolType & 4 )
{
    request_alloc_size += ExpCacheLineSize;
    if ( request_alloc_size > 0xFE0 )
    {
        request_alloc_size -= ExpCacheLineSize;
        PoolType = PoolType & 0xFB;
    }
}
```

If the new allocation size cannot fit in a single page, then the `CacheAligned` bit will be ignored.

Then, the allocated chunk must respect three conditions :

- the final allocation address must be aligned on `ExpCacheLineSize`;
- the chunk must have a `POOL_HEADER` at the very beginning of the chunk;
- the chunk must have a `POOL_HEADER` at the address of allocation minus `sizeof(POOL_HEADER)`.

So if the allocation address is not properly aligned, the chunk might have two headers.

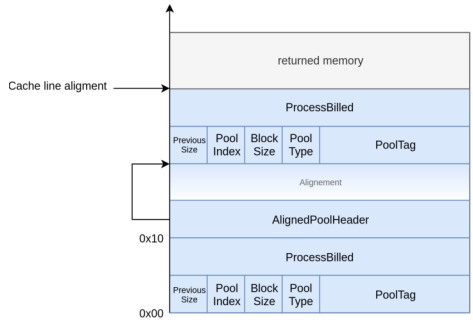

Fig. 11. Layout of cache aligned memory

The first `POOL_HEADER` will be at the begining of the chunk, as usual, while the second will be aligned on `ExpCacheLineSize - sizeof(POOL_HEADER)`, making the final allocation address aligned on `ExpCacheLineSize`. The `CacheAligned` bit is removed from the first `POOL_HEADER`, and the second `POOL_HEADER` is filled with the following values:

PreviousSize Used to store the offset between the two headers.

PoolIndex Unused.
BlockSize Size of the allocated bucket in first `POOL_HEADER`, reduced size in the second one.
PoolType As usual, but the CacheAligned bit is set.
PoolTag As usual, same on both `POOL_HEADER`.
ProcessBilled Unused.

Additionaly, a pointer, that we named AlignedPoolHeader, might be stored after the first `POOL_HEADER`, if there is enough space in the alignement padding. It points on the second `POOL_HEADER`, and is xored with the ExpPoolQuotaCookie.

The figure 11 summarizes the layout of the two `POOL_HEADER` used in case of cache alignment.

2.3 Summary

Since Windows 19H1 and the `Segment Heap` introduction, some informations that were stored into the `POOL_HEADER` of each chunk aren't required anymore. However, others like the `Pooltype`, the `Pooltag`, or the ability to use the `CacheAligned` and the `PoolQuota` mechanism are still needed.

This is why every allocations under 0xFE0 are still preceded with at least one `POOL_HEADER`. The usage of the fields of the `POOL_HEADER` since Windows 19H1 is described in section 2.2. The figure 12 represents a chunk allocated using the LFH backend, thus only preceded with a `POOL_HEADER`.

Fig. 12. Returned memory for a LFH chunk

As explained in 2.1, depending on the backend, memory may be headed by some specific header. For example, a chunk of size 0x280 would use the VS backend, thus would be preceded by a `_HEAP_VS_CHUNK_HEADER` of size 0x10. The figure 13 represents a chunk allocated using the VS segment, thus preceded with a a VS header and a `POOL_HEADER`.

Finally, if the allocation is requested to be aligned on the cache line, the chunk might contain two `POOL_HEADER`. The second one will have

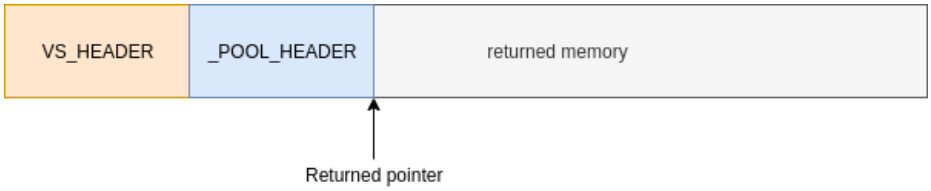

Fig. 13. Returned memory for a VS chunk

the `CacheAligned` bit set, and will be use to retrieve the first one, and the address of the actual allocation. The figure 14 represents a chunk allocated using the LFH, and requested to be aligned on the cache size, thus preceded with two `POOL_HEADER`.

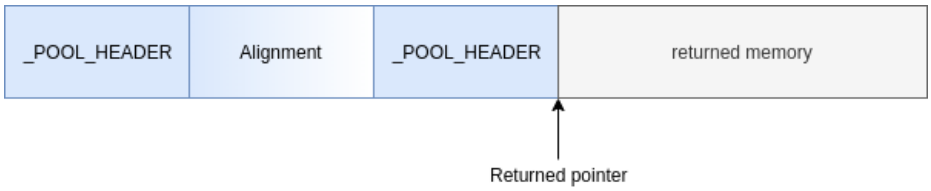

Fig. 14. Returned memory for a LFH chunk aligned on the cache size

The figure 15 summarizes the decision tree used when an allocation is made.

From the exploitation perspective, two conclusions can be drawn. First, the new usage of the `POOL_HEADER` will ease the exploitation: since most of the fields are unused, less care should be taken while overriding them. The other outcome might be to leverage the new usage of the `POOL_HEADER` to find new exploitation techniques.

3 Attacking the POOL_HEADER

If a heap overflow vulnerability allows a really good control on written data and its size, the simplest solution is probably to rewrite the `POOL_HEADER` and directly attack the data of the next chunk. The only thing to do is to make sure the `PoolQuota` bit is not set in the `PoolType`, to avoid an integrity check on the `ProcessBilled` field when the corrupted chunk is freed.

However, this section will provide some attacks that can be done with a heap overflow of a few bytes only, by targeting the `POOL_HEADER`.

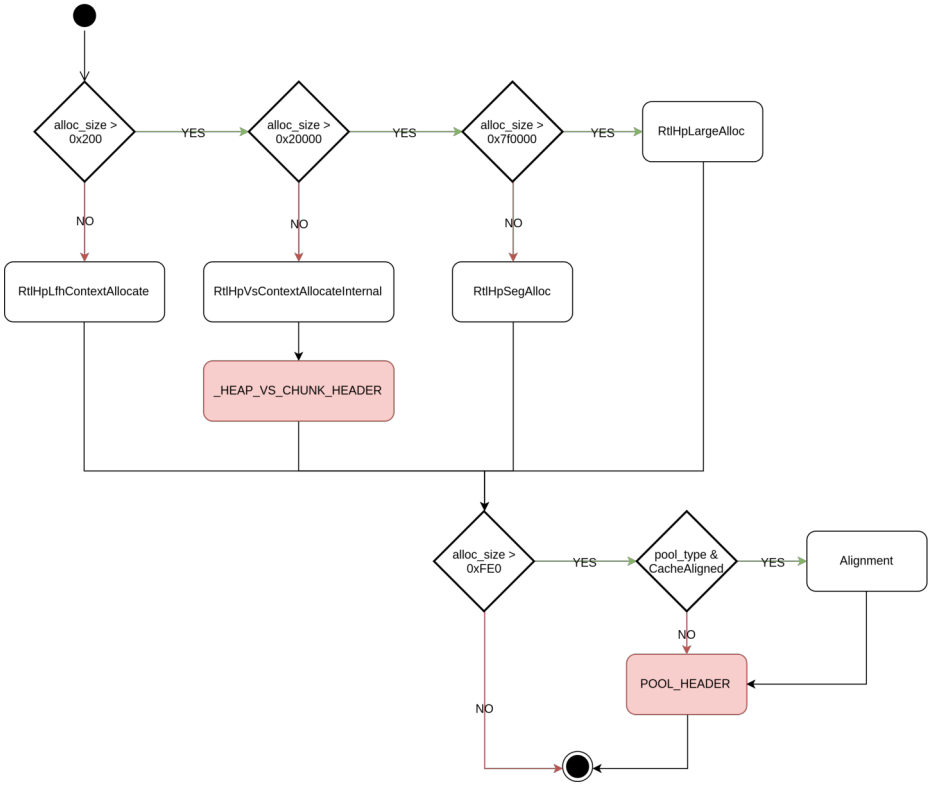

Fig. 15. Decision flow of the Segment Heap allocator

3.1 Targeting the BlockSize

From Heap Overflow to bigger Heap Overflow

As explained in section 2.1, the `BlockSize` field is used in the free mechanism to store some chunks in the `Dynamic Lookaside`.

An attacker might use a heap overflow to change the value of the `BlockSize` field to a bigger one, larger than `0x200`. If the corrupted chunk is freed, the controlled `BlockSize` will be used to store the chunk in a lookaside of the wrong size. The next allocation of this size might use a too small allocation to store all the required data, triggering another heap overflow.

By using spraying techniques and specific objects, an attacker might turn a 3-byte heap overflow into a heap overflow up to `0xFD0` bytes, depending on the vulnerable chunk size. It also allows the attacker to chose the object that is overflowing, and perhaps have more control on the overflow conditions.

3.2 Targeting the PoolType

Most of the time, the information stored in the `PoolType` is just informative; it was given at the allocation time and is stored in the `PoolType`, but will not be used in the free mechanism.

For example, changing the type of memory stored in the `PoolType` will not actually change the type of memory used by the allocation. It is not possible to turn a `NonPagedPoolNx` memory into a `NonPagedPool` just by changing this bit.

But this is not true for the `PoolQuota` and the `CacheAligned` bits. Setting the `PoolQuota` bit will trigger the use of the `ProcessBilled` pointer in the `POOL_HEADER` to dereference the quota upon freeing. As presented in 1.2, the attacks on the `ProcessBilled` pointer have been mitigated.

So the only bit that remain is the `CacheAligned` bit.

Aligned Chunk Confusion

As seen in section 2.2, if an allocation is requested with the `CacheAligned` bit set in the `PoolType`, the layout of the chunk is different.

When the allocator is freeing such an allocation, it will try to find the original chunk address to free the chunk at the correct address. It will use the `PreviousSize` field of the aligned `POOL_HEADER`. The allocator does a simple substraction to compute the original chunk address:

```
if ( AlignedHeader->PoolType & 4 )
{
    OriginalHeader = (QWORD)AlignedHeader - AlignedHeader->
        PreviousSize * 0x10;
    OriginalHeader->PoolType |= 4;
}
```

Before the `Segment Heap` was introduced in the kernel, there were several checks after this operation.

- The allocator checked if the original chunk had the `MustSucceed` bit set in the PoolType.
- The offset between the two headers was recomputed using the `ExpCacheLineSize`, and was verified to be the same than the actual offset between the two headers.

- The allocator checked if the `BlockSize` of the aligned header was equal to the `BlockSize` of the original header plus the `PreviousSize` of the aligned header.
- The allocator checked if the pointer stored at `OriginalHeader + sizeof(POOL_HEADER)` is equal to the address of the aligned header xored with the `ExpPoolQuotaCookie`.

Since Windows 10 `19H1`, with the pool allocator using the `Segment Heap`, all these checks are gone. The xored pointer is still present after the original header, but it's never checked by the free mechanism. The authors suppose that some of the checks have been removed by error. It is likely that some checks will be re-enabled in future releases, but the prebuild of Windows 10 `20H1` show no such patch.

For now, the lack of checks allows an attacker to use the `PoolType` as an attack vector. An attacker might use a heap overflow to set the `CacheAligned` bit of the `PoolType` of the next chunk, and to fully control the `PreviousSize` field. When the chunk is freed, the free mechanism uses the controlled `PreviousSize` to find the original chunk, and free it. Because the `PreviousSize` field is stored on one byte, the attacker can free any address aligned on `0x10` up to `0xFF * 0x10 = 0xFF0` before the original chunk address.

The final part of this paper aims to demonstrate a generic exploit using the techniques presented here. It presents generic objects that are interesting to control in a pool overflow or a Use-After-Free situation, and multiple objects and techniques to reuse a free allocation with controlled data.

4 Generic Exploitation

4.1 Required conditions

This section aims to present techniques to exploit a vulnerability in order to elevate privileges on a Windows system. It is supposed that the attacker is at Low Integrity level.

The ultimate purpose is to develop the most generic exploit possible, that could be used on different types of memory, `PagedPool` and `NonPagedPoolNx`, with different sizes of chunk and with any heap overflow vulnerability that provides the following required conditions.

– When targeting the `BlockSize`, the vulnerability needs to provide the ability to rewrite the 3rd byte of the next chunk's `POOL_HEADER` with a controlled value.
– When targeting the `PoolType`, the vulnerability needs to provide the ability to rewrite the 1st and 4th byte of the next chunk's `POOL_HEADER` with controlled values.
– In all cases, it is required to control the allocation and deallocation of the vulnerable object, in order to maximize the spraying success.

4.2 Exploitation strategies

The chosen exploitation strategy uses the ability to attack the `PoolType` and `PreviousSize` fields of the next chunk's `POOL_HEADER`. The chunk that is vulnerable to the heap overflow will be called the "vulnerable chunk", the chunk placed after will be called the "overwritten chunk".

As describe in section 3.2, by controlling the `PoolType` and the `PreviousSize` fields of the next chunk's `POOL_HEADER` , an attacker can change where the overwritten chunk will actually be freed. This primitive can be exploited in several ways.

This can allow to turn the pool overflow in a Use-After-Free situation, when the attacker set the `PreviousSize` field at exactly the size of the vulnerable chunk. Thus, upon requesting the freeing of the overwritten chunk, the vulnerable chunk will be freed instead, and put in a Use-After-Free situation. Figure 16 present this technique.

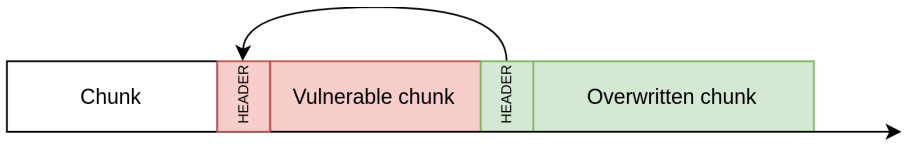

Fig. 16. Exploitation using vulnerable chunk Use-After-Free

However, another technique was chosen. The primitive can also be used to trigger the free of the overwritten chunk in the middle of the vulnerable chunk. It is possible to forge a fake `POOL_HEADER` in the vulnerable chunk (or in a chunk that replaces it), and use the `PoolType` attack to redirect the free on this chunk. This would allow to create a fake chunk in the middle of a legit chunk, and be in a really good overflowing situation. This chunk corresponding will be called "ghost chunk".

The ghost chunk is overriding at least two chunks, the vulnerable chunk, and the overwritten chunk. Figure 17 present this technique.

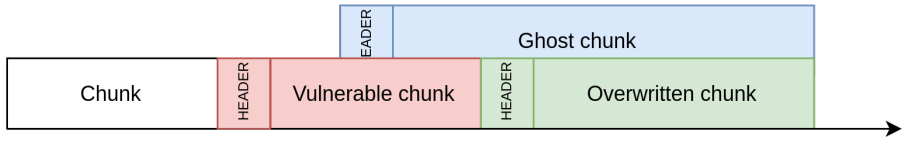

Fig. 17. Chosen exploitation technique

This last technique seems more exploitable than the Use-After-Free, because it puts the attacker in a better situation to control the content of an arbitrary object.

The vulnerable chunk can then be reallocated with an object that allows arbitrary data control. That allows an attacker to partially control the object allocated in the ghost chunk.

An interesting object has to be found in order to be placed in the ghost chunk. In order to have the most generic exploit possible, the object should have the following requirements:

- provides an arbitrary read/write primitive if fully or partially controlled;
- ability to control its allocation and deallocation;
- have a variable size of minimum 0x210 in order to be allocated in the ghost chunk from the corresponding lookaside, but be the smallest possible (to avoid trashing too much of the heap when allocating it).

Since the vulnerable chunk can be placed in both `PagedPool` and `NonPagedPoolNx`, two objects of this kind are needed, one allocated in the `PagedPool`, and the other allocated in the `NonPagedPoolNx`.

This kind of object is not common, and the authors did not find this kind of perfect object. That's why an exploitation strategy was developed using an object that only provides an arbitrary read primitive. The attacker is still able to control the `POOL_HEADER` of the ghost chunk. This means the Quota Pointer Process Overwrite attack can be used to get an arbitrary decrementation primitive. The ExpPoolQuotaCookie and the address of the ghost chunk can be recovered using the arbitary read primitive.

The developed exploit is using this last technique. By leveraging heap massaging and objects interesting to overflow, a 4 byte controlled overflow can be turn into an Elevation of Privilege, from Low Integrity Level to SYSTEM.

4.3 Targeted objects

Paged Pool Afther the creation of a pipe, a user has the ability to add attributes to the pipe. The attributes are a key-value pair, and are stored into a linked list. The PipeAttribute[1] object is allocated in the `PagedPool`, and is defined in the kernel by the structure in Figure 18.

```
struct PipeAttribute {
  LIST_ENTRY list;
  char * AttributeName;
  uint64_t AttributeValueSize;
  char * AttributeValue;
  char data[0];
};
```

Fig. 18. Structure of a PipeAttribute

The size of the allocation and the data is fully controlled by an attacker. The `AttributeName` and `AttributeValue` are pointers pointing on different offset of the data field.

A pipe attribute can be created on a pipe using the NtFsControlFile syscall, and the 0x11003C control code, as shown in Figure 19.

The attribute's value can then be read using the 0x110038 control code. The `AttributeValue` pointer and the `AttributeValueSize` will be used to read the attribute's value and return it to the user. The attributes value can be changed, but this will trigger the deallocation of the previous PipeAttribute and the allocation of a new one.

It means that if an attacker can control the `AttributeValue` and `AttributeValueSize` fields of the PipeAttribute, it can read arbitrary data in kernel, but cannot arbitrary write. This object is also perfect to put arbitrary data in the kernel. It means it can be used to realloc the vulnerable chunk and control the ghost chunk content.

NonPagedPoolNx The ability to use WriteFile into a pipe is a known technique to spray the `NonPagedPoolNx`. When writing into a pipe, the function NpAddDataQueueEntry creates the structure defined in Figure 20.

[1] The structure is not public and has bee named after reverse-engineering

```
HANDLE read_pipe;
HANDLE write_pipe;
char attribute[] = "attribute_name\00attribute_value"
char output[0x100];

CreatePipe(read_pipe, write_pipe, NULL, bufsize);

NtFsControlFile(write_pipe,
    NULL,
    NULL,
    NULL,
    &status,
    0x11003C,
    attribute,
    sizeof(attribute),
    output,
    sizeof(output)
);
```

Fig. 19. Creation of a pipe attribute

The data and size of the PipeQueueEntry[2] is user controlled, since the data is directly stored behind the structure.

When using the entry in the function NpReadDataQueue, the kernel will walk the entry list, and use each entry to retrieve the data.

If the `isDataInKernel` field equals 1, the data is not stored directly behind the structure, but the pointer is stored in an IRP, pointed by `linkedIRP`. If an attacker can fully control this structure, he might set `isDataInKernel` to 1, and make point `linkedIRP` in userland. The `SystemBuffer` field (offset 0x18) of the `linkedIRP` in userland is then used to read the data from the entry. This provides an arbitrary read primitive. This object is also perfect to put arbitrary data in the kernel. It means it can be used to realloc the vulnerable chunk and control the ghost chunk content.

4.4 Spraying

This section describes techniques to spray the kernel heap in order to get the wanted memory layout.

In order to obtain the required memory layout presented in section 4.2, some heap spraying has to be done. Heap spraying depends on the size of the vulnerable chunk since it will end up in different allocation backend.

[2] The structure is not public and has bee named after reverse-engineering

```
struct PipeQueueEntry
{
  LIST_ENTRY list;
  IRP *linkedIRP;
  __int64 SecurityClientContext;
  int isDataInKernel;
  int remaining_bytes__;
  int DataSize;
  int field_2C;
  char data[1];
};
```

Fig. 20. Pipe Queue Entry structure

```
if ( PipeQueueEntry->isDataAllocated == 1 )
  data_ptr = (PipeQueueEntry->linkedIRP->SystemBuffer);
else
  data_ptr = PipeQueueEntry->data;
[...]
memmove((void *)(dst_buf + dst_len - cur_read_offset), &data_ptr[
    PipeQueueEntry->DataSize - cur_entry_offset], copy_size);
```

Fig. 21. Use of a pipe queue entry

In order to ease the spray it can be useful to ensure the corresponding lookaside are empty. Allocating more than 256 chunks of the right size will ensure that.

If the vulnerable chunk is smaller than 0x200 it will be located in the LFH backend. Then, spraying is to be done with chunks of the exact same size, modulo the corresponding bucket granularity, to ensure they all are allocated from the same bucket. As presented in section 2.1, when an allocation is requested, the LFH backend will scan the BlockBitmap by group of at most 32 blocks, and randomly choose a free block. Allocating more than 32 chunk right before and after the allocation of the vulnerable chunk should help defeat the randomization.

If the vulnerable chunk is bigger than 0x200 but smaller than 0x10000 it will end up in the Variable Size backend. Then spraying is to be done with size equals to the size of the vulnerable chunk. Bigger chunk could be split and thus fail the spray. First, allocate thousands of chunk of the chosen size in order to ensure, first, that the FreeChunkTree is emptied of all chunks bigger than the chosen size, then that the allocator will allocate a new VS subsegment of 0x10000 bytes and put it in the FreeChunkTree.

Then allocate another thousands of chunk that will endup in the new big free chunk and thus be contiguous. Then free one third of the last allocated chunk in order to fill the FreeChunkTree. Freing only one third will ensure that no chunk will be coalesced. Then let the vulnerable chunk be allocated. Finaly, the freed chunk can be reallocated in order to maximize spray chances.

Since the full exploitation technique requires to free and reallocate both the vulnerable chunk and the ghost chunk, it can be really interesting to enable the corresponding dynamic lookaside to ease the free chunk recover. To do so, an easy solution is to allocate thousands of chunk of the corresponding size, wait 2 seconds, allocate another thousands of chunk and wait 1 second. Thus we can ensure the Balance Set Manager has rebalanced the corresponding lookaside. Allocating thousands of chunk ensure that the lookaside will be in the top used lookaside and thus will be enabled and it also ensure that it will have enough room in it.

4.5 Exploitation

Demonstration setup To demonstrate the following exploit, a fake vulnerability has been created.

A Windows kernel driver was developed, that exposes several IOCTL that allows to :

- Allocate a chunk with a controlled size in the PagedPool
- Triggers a controlled memcpy in this chunk that allows a fully controlled pool overflow
- Free the allocated chunk

This is of course just for demonstration and provides more control that is actually needed for the exploit to work.

This setup allows an attacker to:

- Control the size of the vulnerable chunk. This is not mandatory, but it's preferable, since the exploit is easier with controlled sizes.
- Control the allocation and deallocation of the vulnerable chunk.
- Overwrite the 4 first bytes of the `POOL_HEADER` of the next chunk with a controlled value

Also, the vulnerable chunk is allocated in the `PagedPool`. This is important since the type of the pool might change the objects used in the exploits, and then have a big impact on the exploitation itself. However, the exploit targeting the `NonPagedPoolNx` is very similar, and only use

PipeQueueEntry for spraying and getting an arbitrary read instead of the PipeAttribute.

For this example, the chosen size of the vulnerable chunk will be 0x180. The discussion about the size of the vulnerable chunk and its impact on the exploit is discussed in the section 4.6.

Creating the ghost chunk The first step here is to massage the heap in order to place a controlled object after the vulnerable chunk.

The object in the overwritten chunk might be anything, the only requirement is to control when it is freed. To simplify the exploit, it's better to chose an object that can be sprayed, see section 4.2.

The vulnerability can now be triggered, the `POOL_HEADER` of the overwritten chunk is replaced with the following values:

PreviousSize : 0x15. This size will be multiplied by 0x10. 0x180 - 0x150 = 0x30, the offset of the fake `POOL_HEADER` in the vulnerable chunk.
PoolIndex : 0, or any value, this is not used.
BlockSize : 0. or any value, this is not used.
PoolType : `PoolType` | 4. The CacheAligned bit is set.

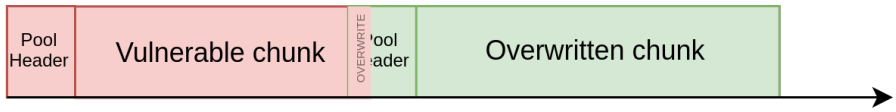

Fig. 22. Triggering the overflow

A fake `POOL_HEADER` must be placed in the vulnerable chunk at a known offset. This is done by freeing the vulnerable object and reallocate the chunk with a PipeAttribute object.

For the demonstration, the offset of the fake `POOL_HEADER` in the vulnerable chunk will be 0x30. The fake `POOL_HEADER` is in the following form:

PreviousSize : 0, or any value, this is not used.
PoolIndex : 0, or any value, this is not used.
BlockSize : 0x21. This size will be multiplied by 0x10, and will be the size of the freed chunk.
PoolType : `PoolType`. The CacheAligned and PoolQuota bits are NOT set.

The chosen `BlockSize` is not random, it's the size of the chunk that will actually be freed. Since the goal is to reuse this allocation afterwards, it's required to pick a size that is easy to reuse. Since all size under 0x200 are in the LFH, such sizes must be avoided. The smallest size that is not the LFH, is an allocation of 0x200, which is a chunk of size 0x210. A size of 0x210 use the VS allocation, and is eligible to use the *Dynamic Lookaside* lists described in section 2.1.

The *Dynamic Lookaside* list for the size 0x210 can be enabled by spraying and freeing chunks of 0x210 bytes.

The overwritten chunk can now be freed, and this will trigger the cache alignement. Instead of freeing the chunk at the address of the overwritten chunk, it will free the chunk at `OverwrittenChunkAddress - (0x15 * 0x10)`, which is also `VulnerableChunkAddress + 0x30`. The `POOL_HEADER` used for the free is the fake `POOL_HEADER` , and instead of freeing the vulnerable chunk, the kernel frees a chunk of size 0x210, and place it on the top of the *Dynamic Lookaside*. This is shown by figure 23.

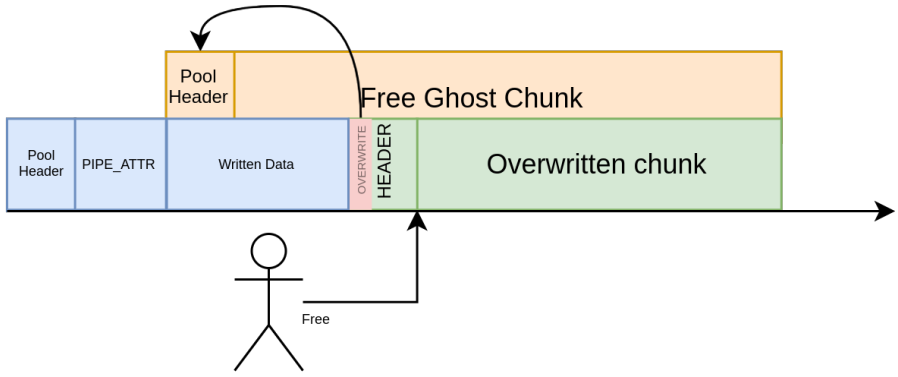

Fig. 23. Freeing the overwritten chunk

Unfortunately, the `PoolType` of the fake `POOL_HEADER` has no impact whether the freed chunk is placed in the `PagedPool` or `NonPagedPoolNx`. The *Dynamic Lookaside* list is picked using the segment of the allocation, which is derived from the address of the chunk. It means that if the vulnerable chunk is in the Paged Pool, this ghost chunk will also be placed in the Paged Pool's lookaside list.

The overwritten chunk is now in "lost" state ; the kernel thinks it's freed, and all reference on the chunk has been dropped. It won't be used anymore.

Leaking the content of the ghost chunk The ghost chunk can now be reallocated with also a PipeAttribute object. The PipeAttribute structure overwrites the value of the attribute placed in the vulnerable chunk. By reading the value of this pipe attribute, the data can be read, and the content of the PipeAttribute of the ghost chunk is leaked. The address of the ghost chunk, and thus of the vulnerable chunk is now known. This step is presented in figure 24.

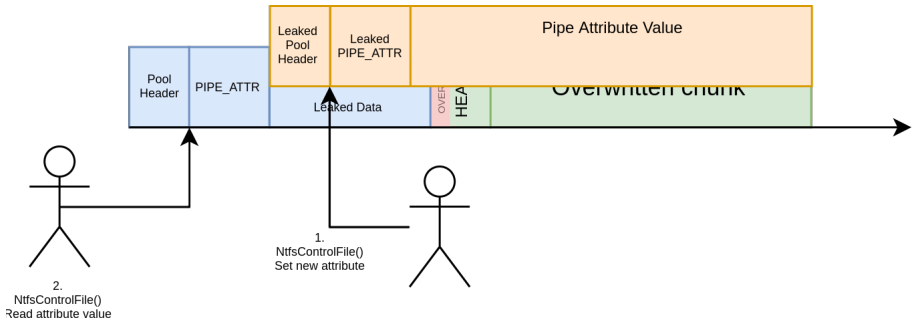

Fig. 24. Leak the ghost chunk PipeAttribute

Getting an arbitrary read The vulnerable chunk can be freed another time and reallocated with an other PipeAttribute. This time, the data of the PipeAttribute will overwrite the PipeAttribute of the ghost chunk. Thus, the PipeAttribute of the ghost chunk can be fully controlled. A new PipeAttribute is injected in the linked list, which is located in userland. This step is presented in figure 25.

Now, by requesting the read of the attribute on the ghost's PipeAttribute, the kernel will use the PipeAttribute that is in userland and thus fully controlled. As seens before, by controlling the AttributeValue pointer and the AttributeValueSize, this provides an arbitrary read primitive. The figure 26 represents an arbitrary read.

Using the first pointer leak and the arbitrary read, a pointer on npfs's text section can be retrieved. By reading the import table, a pointer on the ntoskrnl's text section can be read, which provides the base of the kernel. From there, the attacker can read the value of the ExpPoolQuotaCookie, and retrieve the address of the EPROCESS structure for the exploit process, and the address of its TOKEN.

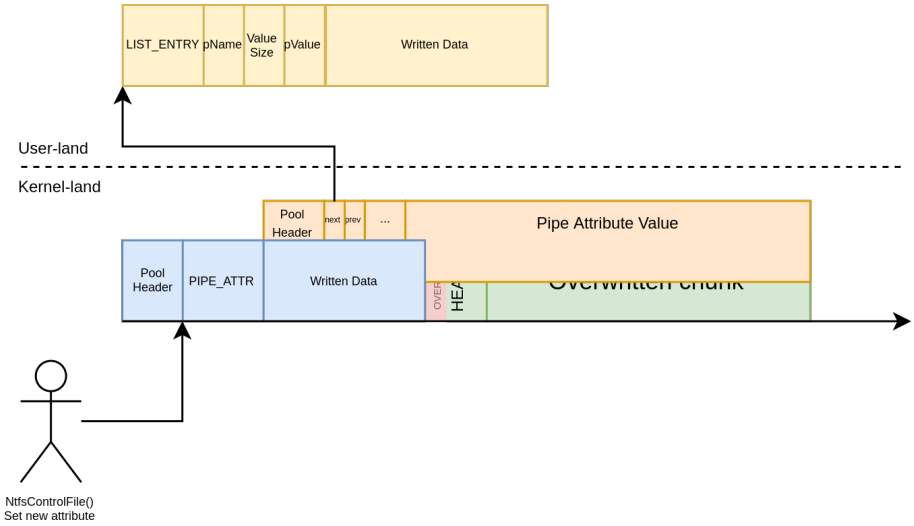

Fig. 25. Rewrite the ghost chunk PipeAttribute

Getting an arbitrary decrementation First, a fake EPROCESS structure is crafted in kernel land using a PipeQueueEntry[3] and its address is retrieved using the arbitrary read.

Then, the exploit can one more time free and reallocate the vulnerable chunk, to change the content of the ghost chunk and its `POOL_HEADER`.

The `POOL_HEADER` of the ghost chunk is overwritten with the following values:

PreviousSize : 0, or any value, this is not used.
PoolIndex : 0, or any value, this is not used.
BlockSize : 0x21. This size will be multiplied by 0x10.
PoolType : 8. The PoolQuota bit IS set.
PoolQuota : `ExpPoolQuotaCookie XOR FakeEprocessAddress XOR GhostChunkAddress`

Upon freeing the ghost chunk, the kernel will try to dereference the Quota counter of the related EPROCESS. It will use the fake EPROCESS structure to find the pointer to the value to dereference.

This provides an arbitrary decrement primitive. The value of the decrementation is the `BlockSize` in the PoolHeader, so it's aligned on 0x10 and between 0 and 0xff0.

[3] See section 4.2

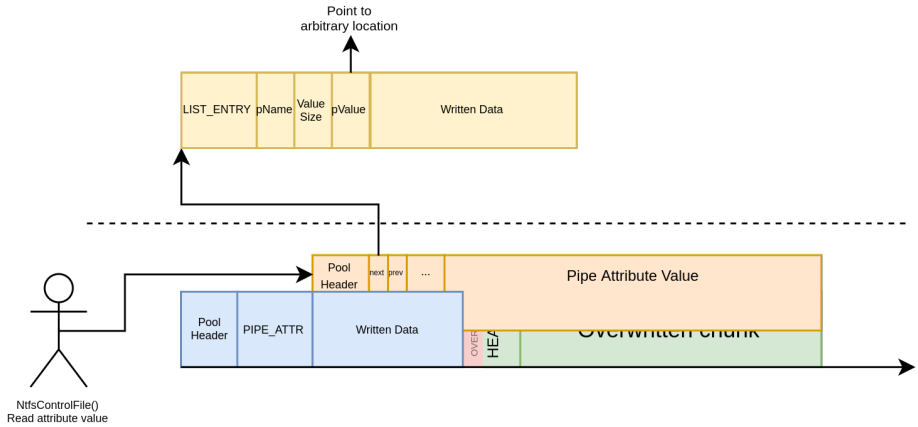

Fig. 26. Use the injected PipeAttribute to arbitrary read

From arbitrary decrementation to SYSTEM In 2012, Cesar Cerrudo [3] described a technique to elevate its privileges by setting the field `Privileges.Enabled` of the TOKEN structure. The `Privileges.Enabled` field is holding the privileges enabled for this process. By default, a token in Low Integrity Level has a `Privileges.Enabled` set to the value 0x0000000000800000, which only gives the SeChangeNotifyPrivilege. By substracting one on this bitfield, it becomes 0x000000000007ffff, which enables a lot more privileges.

The SeDebugPrivilege is enabled by setting the bit 20 on this bitfield. The SeDebugPrivilege allows a process to debug any process on the system, thus gives the ability to inject any code in a privileged process.

The exploit explained in [1] presented a Quota Pointer Process Overwrite that would use the arbitrary decrementation to set the SeDebugPrivilege on its process. The figure 27 present this technique.

However, since Windows 10 v1607, the kernel now also checks the value of the Privileges.Present field of the Token. The `Privileges.Present` field of the token is the list of privileges that **CAN** be enabled for this token, by using the AdjustTokenPrivileges API. So the actual privileges of the TOKEN is now the bitfield resulting of `Privileges.Present & Privileges.Enabled`.

By default, a token in Low Integrity Level has a `Privileges.Present` set to 0x602880000. Because `0x602880000 & (1<<20) == 0`, setting the SeDebugPrivilege in the `Privileges.Enabled` is not enough to obtain the SeDebugPrivilege.

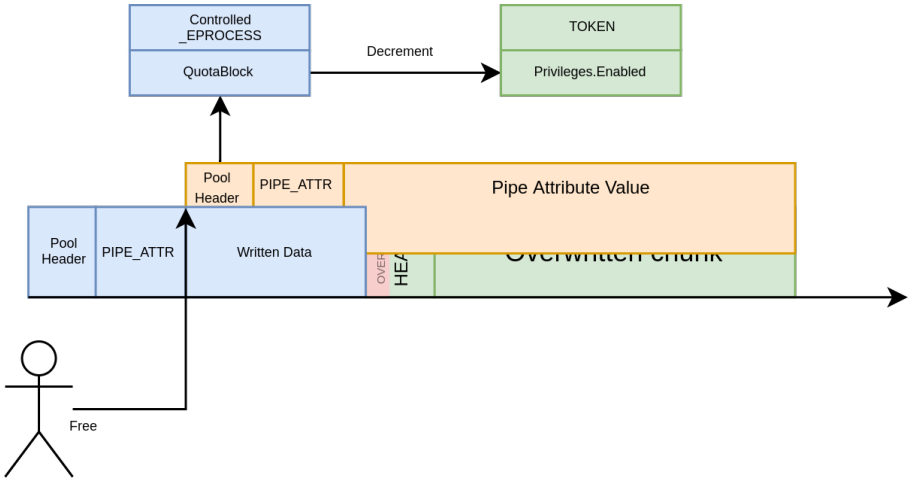

Fig. 27. Exploitation using arbitrary decrement to gain SYSTEM privilege

An idea could be to decrement the `Privileges.Present` bitfield, in order to get the SeDebugPrivilege in the `Privileges.Present` bitfield. Then, the attacker can use the AdjustTokenPrivileges API to enable the SeDebugPrivilege. However, the SepAdjustPrivileges function makes additional checks, and depending on the integrity of the TOKEN, a process cannot enable any privileges, even if the wanted privilege is in the `Privileges.Present` bitfield. For the High Integrity Level, a process can enable any privileges that is in the `Privileges.Present` bitfield. For the Medium Integrity Level, a process can only enable privileges that are in the `Privileges.Present` **AND** in the bitfield 0x1120160684. For the Low Integrity Level, a process can only enable privileges that are in the `Privileges.Present` **AND** in the bitfield 0x202800000.

This means that this technique to get SYSTEM from a single arbitrary decrementation is dead.

However, it can perfectly be done in two arbitrary decrementation, by decrementing first `Privileges.Enabled`, and then `Privileges.Present`.

The ghost chunk can be reallocated and its `POOL_HEADER` overwritten a second time, to get a second arbitrary decrementation.

Once the SeDebugPrivilege is obtained, the exploit can open any SYSTEM process, and injects a shellcode insided that pops a shell as SYSTEM.

4.6 Discussion on the presented exploit

The code of the exploit presented is available at [2], along with the vulnerable driver. This exploit is only a Proof Of Concept and can always be improved.

4.7 Discussion on the size of the vulnerable object

Depending on the size of the vulnerable object, the exploit might have different requirements.

The exploit presented above only works for vulnerable chunk of size 0x130 minimum. This is because of the size of the ghost chunk, which must be at least 0x210. With a vulnerable chunk with a size under 0x130, the allocation of the ghost chunk will overwrite the chunk behind the overwritten chunk, and would trigger a crash when freed. This is fixable, but left as an exercise for the reader.

There is a few differences between a vulnerable object in the LFH (chunks under 0x200), and a vulnerable object in the VS segment (chunks $>$ 0x200). Mainly, a VS chunk has an additional header in front of the chunk. It means that to be able to control the `POOL_HEADER` of the next chunk in the VS segment, a heap overflow of at least 0x14 bytes is required. It also means that when the overwritten chunk will be freed its `_HEAP_VS_CHUNK_HEADER` must have been fixed. Additionnaly, care must be taken to not free the 2 chunks sprayed right after the overwritten chunk because the free mechanism of VS might read the VS header of the overwritten chunk in an attempt to merge 3 free chunks.

Finally, the heap massaging in LFH and in VS are quite different, as explained in section 4.4.

5 Conclusion

This paper described the state of the pool internals since Windows 10 `19H1` update. The `Segment Heap` has been brought to the kernel, and it does not need chunk metadata to properly work. However, the old `POOL_HEADER` that was at the top of each chunk is still present, but used differently.

We demonstrated some attacks that can be done using a heap overflow in the Windows kernel, by attacking the internals specific to the pool.

The demonstrated exploit can be adapted to any vulnerability that provide a minimal heap overflow, and allows a local privilege escalation from a Low Integrity level to SYSTEM.

References

1. Corentin Bayet. Exploit of CVE-2017-6008 with Quota Process Pointer Overwrite attack. `https://github.com/cbayet/Exploit-CVE-2017-6008/blob/master/Windows10PoolParty.pdf`, 2017.
2. Corentin Bayet and Paul Fariello. PoC exploiting Aligned Chunk Confusion on Windows kernel Segment Heap. `https://github.com/synacktiv/Windows-kernel-SegmentHeap-Aligned-Chunk-Confusion`, 2020.
3. Cesar Cerrudo. Tricks to easily elevate its privileges. `https://media.blackhat.com/bh-us-12/Briefings/Cerrudo/BH_US_12_Cerrudo_Windows_Kernel_WP.pdf`, 2012.
4. Matt Conover and w00w00 Security Development. w00w00 on Heap Overflows. `http://www.w00w00.org/files/articles/heaptut.txt`, 1999.
5. Tarjei Mandt. Kernel Pool Exploitation on Windows 7. *Blackhat DC*, 2011.
6. Haroon Meer. Memory Corruption Attacks The (almost) Complete History. *Blackhat USA*, 2010.
7. Mark Vincent Yason. Windows 10 Segment Heap Internals. *Blackhat US*, 2016.