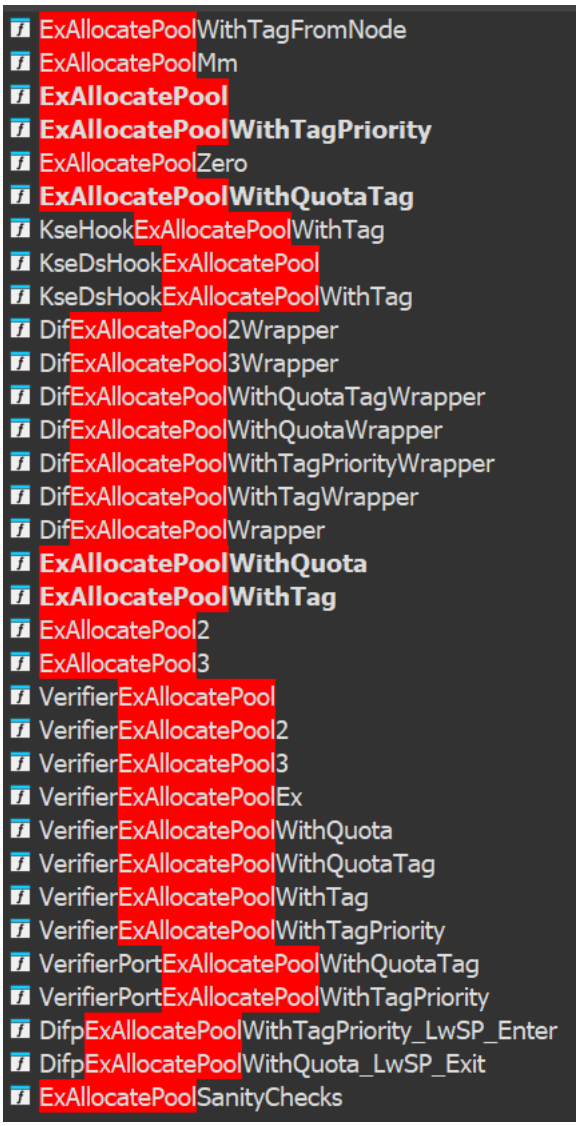
Kernel Pool Internals

r0keb.github.io/posts/Windows-Kernel-Pool-Internals

July 5, 2025



Good morning! In today's blog post we're going to dive into a topic that has interested me for quite some time, the Windows kernel pool. It's a topic that tends to have "scarce" documentation online and can be somewhat intricate. That's precisely why it has captured my attention from the beginning.

In this post we'll explore its internals and see how it works behind the scenes, aiming to gain a deeper understanding of this core component of the Windows OS.

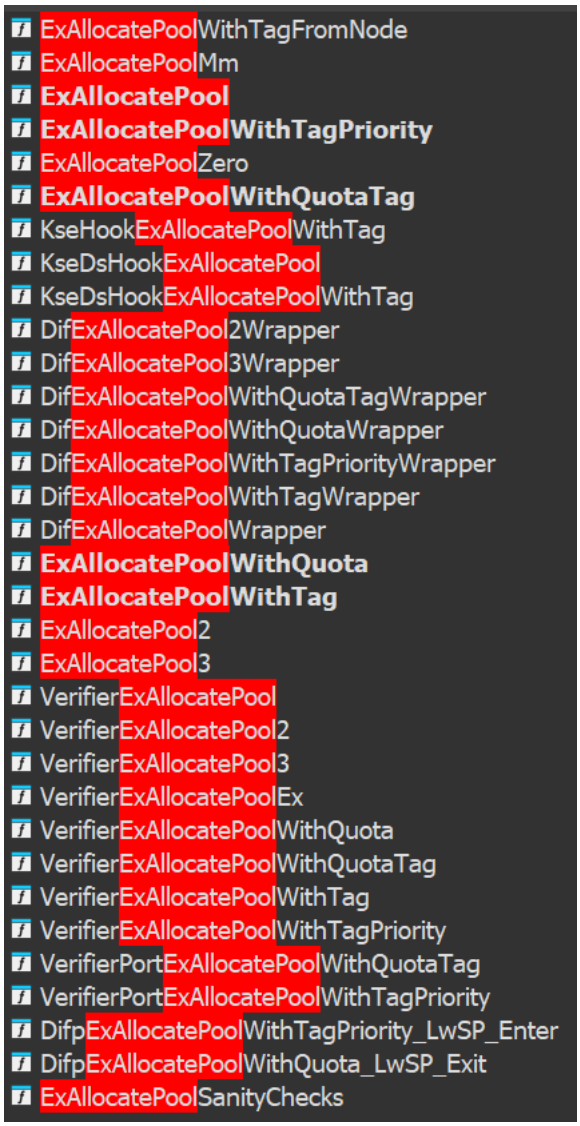
Basics

To begin with, the **Windows Kernel Pool** is a range of memory within the kernel's address space used for dynamic memory allocations by drivers or the kernel itself, in other words, it's the kernel-mode equivalent of the user-mode heap.

When the system initializes, the memory manager creates two memory pools:

- **Non-paged pool** - ranges of (kernel) virtual addresses that always reside in RAM and can be accessed without triggering any paging errors such as page faults. This is very important because this memory can be accessed at any IRQL, and at IRQLs **DPC/dispatch** or higher, only non-paged pool memory must be accessed.
- **Paged Pool** - a region of the system's virtual memory that can be paged out without any issues. Drivers that do not need to access memory from IRQL levels at or above **DPC/dispatch** can safely use this pool.

Here are some pool-related functions found in `ntoskrnl.exe` (24H2).



The number of paged pools can be found at `nt!ExpNumberOfPagedPools`.

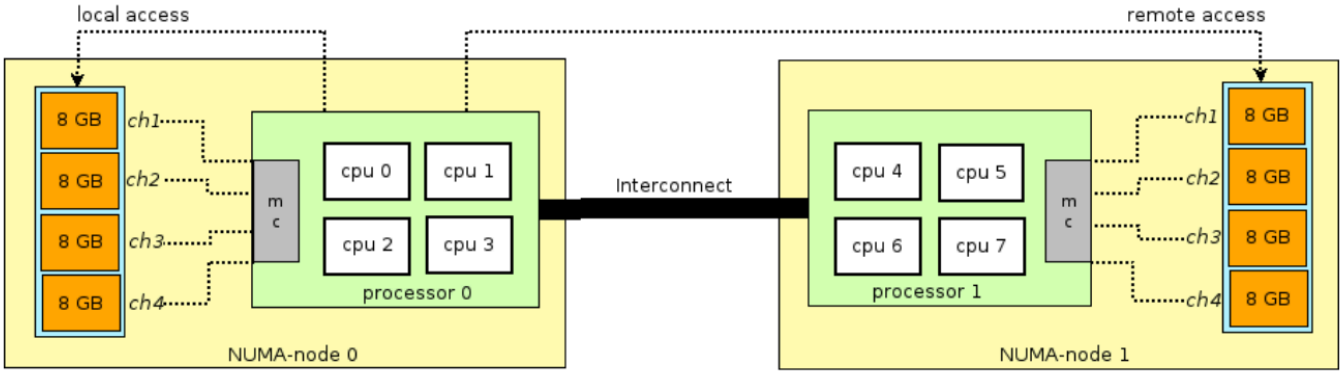
- On single-processor systems, the `nt!ExpPagedPoolDescriptor` array contains 4 paged pool descriptors.
- On multiprocessor systems, a paged pool descriptor is defined per node (`_KNODE`).

What's a `_KNODE`, you ask? Well, to truly learn it (not just memorize it) we first need to understand what a *node* is. But to understand what a *node* is, we need to introduce the concept of **NUMA** architecture (*Non-Uniform Memory Access*).

NUMA is an architecture used to optimize how memory is allocated in multiprocessor systems. Within this architecture, a **node** represents:

- A group of CPUs that share a portion of physical memory, this memory is the fastest and closest for that particular group.
- The RAM directly associated with the node.

This image from John's post Multithreading and the Memory Subsystem helps visualize how this architecture works.



That said, what is a `_KNODE`?
A `_KNODE` is the internal Windows structure that represents a NUMA node within the kernel. It contains data about the processors assigned to the node, memory allocation lists, pointers to memory pools, etc…

You can find them in `nt!KeNodeBlock[]`, an array of pointers to all `_KNODE` structures in the system.

Windows assigns memory to the NUMA node local to the CPU making the call to improve performance, reduce latency, and distribute load among nodes, in short, to be more efficient.
If we take a look at the kernel in IDA we'll see multiple references to `_KNODE` or **`KeGetCurrentNodeNumber()`**, which are used to decide from which pool memory should be allocated.

With that said, let's look at the structure itself:

```
0: kd> dt nt!_KNODE
   +0x000 NodeNumber       : Uint2B
   +0x002 PrimaryNodeNumber : Uint2B
   +0x004 ProximityId      : Uint4B
   +0x008 MaximumProcessors : Uint2B
   +0x00a Flags            : <unnamed-tag>
   +0x00b GroupSeed        : UChar
   +0x00c PrimaryGroup     : UChar
   +0x00d Padding          : [3] UChar
   +0x010 ActiveGroups     : _KGROUP_MASK
   +0x020 SchedulerSubNodes : [32] Ptr64 _KSCHEDULER_SUBNODE
   +0x120 ActiveTopologyElements : [5] Uint4B
   +0x134 PerformanceSearchRanks : [8] _KNODE_SUBNODE_SEARCH_RANKS
   +0x234 EfficiencySearchRanks : [8] _KNODE_SUBNODE_SEARCH_RANKS
```

Now, regarding the non-paged pool, the number of pages currently in use resides in `nt!ExpNumberOfNonPagedPools`.

- On single-processor 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 `nt!ExpNonPagedPoolDescriptor`.

There's also *session pool memory*, which is used for pool allocations in per-user sessions, located in the pool descriptor `nt!MM_SESSION_SPACE`. This enables session isolation between users in the Windows graphical subsystem (win32k).

Pool Descriptor

This is essentially the management structure for kernel pools. It's responsible for current allocations, pages in use, reusable pool tracking, and basically all pool-related data.

kernels/x86/windows-7/sp1/_POOL_DESCRIPTOR

```
//0x1140 bytes (sizeof)
struct _POOL_DESCRIPTOR
{
    enum _POOL_TYPE PoolType;                                               //0x
    union
    {
        struct _KGUARDED_MUTEX PagedLock;                                   //0x
        ULONG NonPagedLock;                                                 //0x
    };
    volatile LONG RunningAllocs;                                            //0x
    volatile LONG RunningDeAllocs;                                          //0x
    volatile LONG TotalBigPages;                                            //0x
    volatile LONG ThreadsProcessingDeferrals;                               //0x
    volatile ULONG TotalBytes;                                              //0x
    ULONG PoolIndex;                                                        //0x
    volatile LONG TotalPages;                                               //0x
    VOID** volatile PendingFrees;                                           //0x
    volatile LONG PendingFreeDepth;                                         //0x
    struct _LIST_ENTRY ListHeads[512];                                      //0x
};
```

NOTE: This structure comes from Windows 7

Windows Pool APIs

`ExAllocatePoolWithTag` is deprecated. According to MSDN:

```
ExAllocatePoolWithTag has been deprecated in Windows 10, version 2004 and has been replaced by [ExAllocatePool2](https://learn.microsoft.com/en-us/windows-h
```

Now `ExAllocatePool2` is used, which includes functionality like zero-initializing the allocated memory, i.e., it wipes it clean.

ExAllocatePoolWithTag

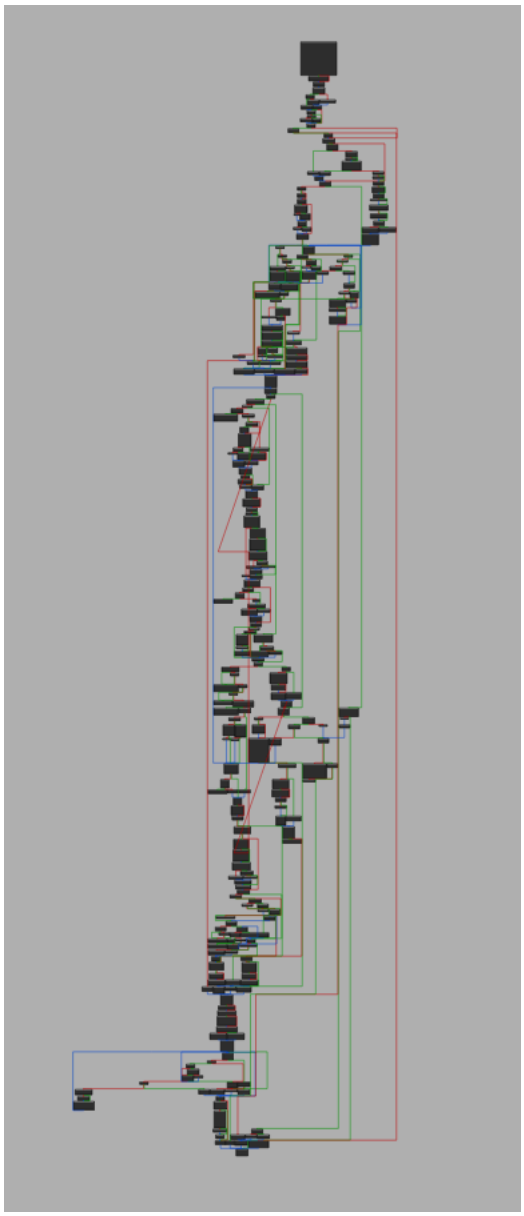
Here's the pseudocode from IDA:

```
PVOID __stdcall ExAllocatePoolWithTag(POOL_TYPE PoolType, SIZE_T NumberOfBytes, ULONG T
{
  __int64 v6; // rcx
  __int64 v7; // rdx
  ULONG_PTR v8; // rcx
  __int64 v9; // rdx
  __int64 v10; // r8
  ULONG v11; // r9d
  PVOID result; // rax

  v6 = 256;
  if ( (PoolType & 1) == 0 )
  {
    v6 = 128;
    if ( (PoolType & 0x200) != 0 )
      v6 = 64;
  }
  if ( PoolType < NonPagedPool )
    v6 = 64;
  v7 = v6 | 4;
  if ( (PoolType & 0x20) == 0 )
    v7 = v6;
  v8 = v7 | 2;
  if ( (PoolType & 0x400) != 0 )
    v8 = v7;
  if ( (PoolType & 0xDE) != 0 )
  {
    v9 = v8 | 8;
    if ( (PoolType & 4) == 0 )
      v9 = v8;
    v10 = v9 | 0x200;
    if ( (PoolType & 0x80u) == 0 )
      v10 = v9;
    v8 = v10 | 0x400;
    if ( (PoolType & 0x40) == 0 )
      v8 = v10;
    if ( (PoolType & 0x10) != 0 )
      v8 |= 0x20uLL;
  }
  v11 = Tag & 0x7FFFFFFF;
  if ( !v11 )
    v11 = 811884866;
  result = (PVOID)ExAllocatePool2(v8, NumberOfBytes, v11);
  if ( !result && (PoolType & 2) != 0 )
    KeBugCheckEx(0x41u, NumberOfBytes, 0, 0, 0);
  return result;
}
```

As we can see, it's just a "wrapper" around `ExAllocatePool2`.
But that can be misleading. As the MSDN notes, this function is deprecated.

So let's look at it in **Windows 10 1507**:



As we can see, it differs from the version in `24H2`. But now we'll focus on the widely used and current function, **`ExAllocatePool2`**.

ExAllocatePool2 (24h2)

Here's the pseudocode:

```
ULONG_PTR __fastcall ExAllocatePool2(ULONG_PTR BugCheckParameter3, ULONG_PTR a2, ULONG_PTR a3)
{
  __int64 PoolWithTagFromNode; // rsi
  ULONG v4; // edi
  ULONG_PTR v5; // rbx
  __int64 v6; // rcx
  ULONG_PTR v7; // r9
  ULONG_PTR v9; // r14
  _KPROCESS *Process; // r15
  ULONG_PTR v11; // rax
  ULONG_PTR v12; // rbp
  ULONG_PTR v13; // rdx
  __int16 v14; // cx
  __int64 v15; // r10
  _KSCHEDULING_GROUP *SchedulingGroup; // rax
  unsigned __int64 *v17; // r12
  char v18; // r8
  unsigned __int64 v19; // r13
  unsigned __int64 v20; // rax
  unsigned __int64 v21; // rdx
  bool v22; // zf
  signed __int64 v23; // rax
  unsigned __int64 v24; // rax
  signed __int64 v25; // rdx
  unsigned __int64 v26; // rdx
  unsigned __int64 v27; // rax
  unsigned __int64 v28; // rcx
  __int64 HeapFromVA; // rax
  ULONG_PTR v30; // rbx
  _BYTE *BugCheckParameter4; // rdx
  KIRQL v32; // al
  int v33; // r8d
  unsigned __int64 v34; // r12
  unsigned int v35; // r9d
  char *v36; // rcx
  unsigned __int64 v37; // rcx
  unsigned __int64 v38; // rcx
  char v39; // al
  signed __int32 v40[8]; // [rsp+0h] [rbp-A8h] BYREF
  __int64 v41; // [rsp+40h] [rbp-68h]
  unsigned __int64 v42; // [rsp+48h] [rbp-60h] BYREF
  __int64 v43; // [rsp+50h] [rbp-58h]
  ULONG_PTR v44; // [rsp+58h] [rbp-50h]
  __int64 v45[2]; // [rsp+60h] [rbp-48h] BYREF
  __int64 retaddr; // [rsp+A8h] [rbp+0h]
  char v47; // [rsp+B0h] [rbp+8h]
  __int64 v48; // [rsp+C8h] [rbp+20h] BYREF

  PoolWithTagFromNode = 0;
  v4 = a3;
  v5 = BugCheckParameter3;
  *(_OWORD *)v45 = 0;
  if ( (BugCheckParameter3 & 0x1C0) == 0
    || (((BugCheckParameter3 & 0x1C0) - 1) & BugCheckParameter3 & 0x1C0) != 0
    || (BugCheckParameter3 & 0xFFFFF000) != 0
    || (BugCheckParameter3 & 0x10) != 0
    || (BugCheckParameter3 & 0x800) != 0
    || !(_DWORD)a3 )
  {
    v6 = 3221225485LL;
    goto LABEL_4;
  }
  if ( (ExpPoolFlags & 8) != 0 )
  {
    if ( (BugCheckParameter3 & 0x200) == 0 )
    {
      LODWORD(v45[1]) = 32;
      v45[0] = v45[0] & 0xFFFFFFFFFFFFF00uLL | 1;
      return VfHandlePoolAlloc(
               NonPagedPool,
               BugCheckParameter3 & 0xFFFFFFFFFFFFFFEuLL,
               a2,
               a3,
               LowPoolPriority,
               (__int64)v45,
               1,
               retaddr);
    }
    v5 = BugCheckParameter3 & 0xFFFFFFFFFFFFDFFuLL;
  }
  v7 = KeGetCurrentPrcb()->SchedulerSubNode->Affinity.Reserved[0];
  if ( (v5 & 1) == 0 )
  {
```

```
    LODWORD(v7) = v7 | 0x80000000;
    PoolWithTagFromNode = ExpAllocatePoolWithTagFromNode(v5, a2, a3, v7);
    if ( PoolWithTagFromNode )
      return PoolWithTagFromNode;
    v6 = 3221225626LL;
LABEL_4:
    if ( (v5 & 0x20) != 0 )
      RtlRaiseStatus(v6);
    return PoolWithTagFromNode;
  }
  v9 = v5;
  LODWORD(v48) = 0;
  v41 = 0;
  Process = KeGetCurrentThread()->ApcState.Process;
  if ( Process == PsInitialSystemProcess )
    v9 = v5 & 0xFFFFFFFFFFFFFFFEuLL;
  LODWORD(v7) = v7 | 0x80000000;
  v11 = ExpAllocatePoolWithTagFromNode(v9, a2, a3, v7);
  v12 = v11;
  if ( !v11
    || (v9 & 1) == 0
    || ExpSpecialAllocations && (HeapFromVA = ExGetHeapFromVA(v11), (unsigned int)ExpHpIsSpecialPoolHeap(HeapFromVA)
  {
    PoolWithTagFromNode = v12;
    if ( v12 )
      return PoolWithTagFromNode;
LABEL_46:
    v6 = 3221225626LL;
    goto LABEL_4;
  }
  v44 = v12 & 0xFFF;
  if ( (v12 & 0xFFF) != 0 )
  {
    v13 = v12 - 16;
    if ( (*(_BYTE *)(v12 - 13) & 4) != 0 )
      v13 += -16LL * (unsigned __int8)*(_WORD *)v13;
    v14 = *(_WORD *)(v13 + 2);
    v41 = 16LL * (unsigned __int8)v14;
    LODWORD(v48) = *(_DWORD *)(v13 + 4);
    if ( (v14 & 0x800) != 0 )
      *(_QWORD *)(v13 + 8) = ExpPoolQuotaCookie ^ v13;
  }
  else
  {
    v32 = ExAcquireSpinLockShared(&ExpLargePoolTableLock);
    v33 = 1;
    v34 = v32;
    v35 = (PoolBigPageTableSize - 1) & ((40543 * (v12 >> 12)) ^ ((40543 * (v12 >> 12)) >> 32));
    while ( 1 )
    {
      v36 = (char *)PoolBigPageTable + 32 * v35;
      if ( *(_QWORD *)v36 == v12 )
        break;
      if ( ++v35 >= (unsigned __int64)PoolBigPageTableSize )
      {
        if ( !v33 )
          goto LABEL_62;
        v35 = 0;
        v33 = 0;
      }
    }
    if ( !v36 )
LABEL_62:
      KeBugCheckEx(0x19u, 0x22u, v12, (unsigned int)v9, 0);
    if ( (*((_DWORD *)v36 + 3) & 0x100) != 0 )
    {
      *((_QWORD *)v36 + 3) = ExpPoolQuotaCookie ^ v12;
      LODWORD(v48) = *((_DWORD *)v36 + 2);
      v41 = *((_QWORD *)v36 + 2);
    }
    ExReleaseSpinLockSharedFromDpcLevel(&ExpLargePoolTableLock);
    if ( KiIrqlFlags )
      KiLowerIrqlProcessIrqlFlags(KeGetCurrentIrql());
    __writecr8(v34);
  }
  if ( Process != PsInitialSystemProcess )
  {
    v15 = (v9 & 0x100) != 0;
    SchedulingGroup = Process[1].SchedulingGroup;
    v43 = v15;
    v17 = (unsigned __int64 *)(&SchedulingGroup->Policy + 16 * v15);
    v18 = PspResourceFlags[8 * v15];
    v47 = v18;
    _m_prefetchw(v17);
```

```
    v19 = *v17;
    _InterlockedOr(v40, 0);
LABEL_29:
    v20 = v17[8];
LABEL_30:
    v42 = v20;
    while ( 1 )
    {
      v21 = v19 + v41;
      if ( v19 + v41 < v19 )
        break;
      if ( v21 <= v20 )
      {
        v23 = _InterlockedCompareExchange64((volatile signed __int64 *)v17, v21, v19);
        v22 = v19 == v23;
        v19 = v23;
        if ( !v22 )
          goto LABEL_29;
        _m_prefetchw(v17 + 1);
        v24 = v17[1];
        do
        {
          if ( v21 <= v24 )
            break;
          v37 = v24;
          v24 = _InterlockedCompareExchange64((volatile signed __int64 *)v17 + 1, v21, v24);
        }
        while ( v24 != v37 );
        if ( Process && (v18 & 4) != 0 )
        {
          v25 = _InterlockedExchangeAdd64((volatile signed __int64 *)&Process[1].ThreadListHead.Blink + v15, v41);
          v26 = v41 + v25;
          _m_prefetchw(&Process[1].DeepFreezeStartTime + v15);
          v27 = *(&Process[1].DeepFreezeStartTime + v15);
          do
          {
            if ( v26 <= v27 )
              break;
            v28 = v27;
            v27 = _InterlockedCompareExchange64(
                    (volatile signed __int64 *)&Process[1].DeepFreezeStartTime + v15,
                    v26,
                    v27);
          }
          while ( v27 != v28 );
        }
        goto LABEL_48;
      }
      if ( (v18 & 1) == 0 || !v17[10] )
        break;
      v38 = _InterlockedExchange64((volatile __int64 *)v17 + 9, 0);
      if ( v38 )
      {
        v20 = v38 + _InterlockedExchangeAdd64((volatile signed __int64 *)v17 + 8, v38);
        goto LABEL_30;
      }
      v39 = PspExpandQuota(v15, (_DWORD)v17, v19, v41, (__int64)&v42);
      v15 = v43;
      if ( !v39 )
        break;
      v20 = v42;
      v18 = v47;
    }
    if ( *(int *)&PspResourceFlags[8 * v15 + 4] < 0 )
    {
      ExFreePoolWithTag((PVOID)v12, v4);
      goto LABEL_46;
    }
  }
LABEL_48:
  v30 = 0;
  if ( v44 )
  {
    v30 = v12 - 16;
    if ( (*(_BYTE *)(v12 - 13) & 4) != 0 )
      v30 += -16LL * (unsigned __int8)*(_WORD *)v30;
    if ( (*(_BYTE *)(v30 + 3) & 8) == 0 )
      goto LABEL_57;
    BugCheckParameter4 = (_BYTE *)(ExpPoolQuotaCookie ^ *(_QWORD *)(v30 + 8) ^ v30);
    *(_QWORD *)(v30 + 8) = (unsigned __int64)Process ^ ExpPoolQuotaCookie ^ v30;
  }
  else
  {
    BugCheckParameter4 = (_BYTE *)ExpStampBigPoolEntry(v12, v9, (__int64)&v48);
```

```
  }
  if ( BugCheckParameter4
    && BugCheckParameter4 != (_BYTE *)-1LL
    && ((unsigned __int64)BugCheckParameter4 < 0xFFFF800000000000uLL || (*BugCheckParameter4 & 0x7F) != 3) )
  {
    if ( v30 )
      LODWORD(PoolWithTagFromNode) = *(_DWORD *)(v30 + 4);
    KeBugCheckEx(0xC2u, 0xDu, v12, (unsigned int)PoolWithTagFromNode, (ULONG_PTR)BugCheckParameter4);
  }
LABEL_57:
  ObfReferenceObjectWithTag(Process, v4);
  return v12;
}
```

It's important to highlight that this is essentially a "handler" for the real kernel memory allocator.
If we dig deeper, we'll encounter functions such as **`ExAllocateHeapPool()`**, which are much more complex and are actually responsible for performing the allocation.

Here's a snippet from **`ExAllocateHeapPool()`**:

```
2221 LABEL_422:
2222             v168 = v163;
2223             goto LABEL_253;
2224           }
2225         }
2226         v15 = v460;
2227         v8 = (unsigned int)((_DWORD)v168 - (v157 + 64));
2228         v252 = _R10 + 8 * v8;
2229         BugCheckParameter3e = v252;
2230         v253 = v460 * v252;
2231         *(_BYTE *)(v157 + 36) = v252 >> 6;
2232         v254 = (unsigned int)HIWORD(v470) + v253;
2233         if ( v499 <= 1u )
2234         {
2235           if ( v252 > *(unsigned __int16 *)(v157 + 48) )
2236           {
2237             RtlpHpLfhSubsegmentPrefetch(v18 + 832, v157, (unsigned int)v254);
2238             v157 = Size;
2239           }
2240 LABEL_378:
2241           Slow = v157 + v254;
2242           v133 = v5 == 0;
2243           v443 = Slow;
2244           v248 = v435;
2245           if ( v133 )
2246           {
2247             RtlHeapZero(Slow, ((unsigned int)v435 + 15LL) & 0xFFFFFFFFFFFFFFF0uLL);
2248             v157 = v449;
2249           }
2250 LABEL_380:
2251           if ( !Slow )
2252             goto LABEL_665;
2253         }
2254         else
2255         {
2256           v255 = RtlpHpLfhSubsegmentCommitBlock(v18 + 832, v157, (unsigned int)v254);
```

The function prototypes are as follows:

```
PVOID ExAllocatePoolWithTag(
  [in] __drv_strictTypeMatch(__drv_typeExpr)POOL_TYPE PoolType,
  [in] SIZE_T                                        NumberOfByt
  [in] ULONG                                         Tag
);
```

```
DECLSPEC_RESTRICT PVOID ExAllocatePoo
  POOL_FLAGS Flags,
  SIZE_T     NumberOfBytes,
  ULONG      Tag
);
```

As we can see, the first parameter is `PoolType` from the enum **POOL_FLAGS**, which specifies the type of memory to be allocated:

```
//0x4 bytes (sizeof)
enum _POOL_TYPE
{
    NonPagedPool = 0,
    NonPagedPoolExecute = 0,
    PagedPool = 1,
    NonPagedPoolMustSucceed = 2,
    DontUseThisType = 3,
    NonPagedPoolCacheAligned = 4,
    PagedPoolCacheAligned = 5,
    NonPagedPoolCacheAlignedMustS = 6,
    MaxPoolType = 7,
    NonPagedPoolBase = 0,
    NonPagedPoolBaseMustSucceed = 2,
    NonPagedPoolBaseCacheAligned = 4,
    NonPagedPoolBaseCacheAlignedMustS = 6,
    NonPagedPoolSession = 32,
    PagedPoolSession = 33,
    NonPagedPoolMustSucceedSession = 34,
    DontUseThisTypeSession = 35,
    NonPagedPoolCacheAlignedSession = 36,
    PagedPoolCacheAlignedSession = 37,
    NonPagedPoolCacheAlignedMustSSession =
    NonPagedPoolNx = 512,
    NonPagedPoolNxCacheAligned = 516,
    NonPagedPoolSessionNx = 544
};
```

More Pool Internals

Every single allocated pool block is preceded by a `_POOL_HEADER` structure (`0x10` or 16 bytes in size):

```
1: kd> dt nt!_POOL_HEADER
   +0x000 PreviousSize     : Pos 0, 8 Bits
   +0x000 PoolIndex        : Pos 8, 8 Bits
   +0x002 BlockSize        : Pos 0, 8 Bits
   +0x002 PoolType         : Pos 8, 8 Bits
   +0x000 Ulong1           : Uint4B
   +0x004 PoolTag          : Uint4B
   +0x008 ProcessBilled    : Ptr64 _EPROCESS
   +0x008 AllocatorBackTraceIndex : Uint2B
   +0x00a PoolTagHash      : Uint2B
```

This structure is very useful, as it provides plenty of information about the allocated block.

In Vergilius, it would look like this:

```
  //0x10 bytes (sizeof)
  struct _POOL_HEADER
  {
      union
      {
          struct
          {
              USHORT PreviousSize:8;                                  //
              USHORT PoolIndex:8;                                     //
              USHORT BlockSize:8;                                     //
              USHORT PoolType:8;                                      //
          };
          ULONG Ulong1;                                               //
      };
      ULONG PoolTag;                                                  //
      union
      {
          struct _EPROCESS* ProcessBilled;                            //
          struct
          {
              USHORT AllocatorBackTraceIndex;                         //
              USHORT PoolTagHash;                                     //
          };
      };
  };
```

As shown in the following example, we place a breakpoint in **`ExAllocatePool2()`**, step out, and then run the `!pool` utility on `rax`, which holds the result, the address of the allocated memory returned by the caller:

```
0: kd> bp nt!ExAllocatePool2
0: kd> g
Breakpoint 0 hit
nt!ExAllocatePool2:
fffff805`c21650f0 48895c2410      mov     qword ptr [rsp+10h],rbx
0: kd> gu
nt!KiInsertNewDpcRuntime+0x55:
fffff805`c1a251d5 4c8bf0          mov     r14,rax
0: kd> r rax
rax=ffffe18f1d5cf710
0: kd> !pool @rax
Pool page ffffe18f1d5cf710 region is Nonpaged pool
 ffffe18f1d5cf010 size:   30 previous size:    0  (Free)       PsIn
 ffffe18f1d5cf040 size:   30 previous size:    0  (Allocated)  FOCX
 ffffe18f1d5cf070 size:   30 previous size:    0  (Allocated)  FOCX
 ffffe18f1d5cf0a0 size:   30 previous size:    0  (Free)       IoUs
 ffffe18f1d5cf0d0 size:   30 previous size:    0  (Free)       PsIn
 ffffe18f1d5cf100 size:   30 previous size:    0  (Allocated)  IoFE
 ffffe18f1d5cf130 size:   30 previous size:    0  (Free)       PsIn
 ffffe18f1d5cf160 size:   30 previous size:    0  (Free)       PsIn
 ffffe18f1d5cf190 size:   30 previous size:    0  (Allocated)  FOCX
 ffffe18f1d5cf1c0 size:   30 previous size:    0  (Free)       Ipng
 ffffe18f1d5cf1f0 size:   30 previous size:    0  (Free)       IoUs
 ffffe18f1d5cf220 size:   30 previous size:    0  (Free)       IoSB
 ffffe18f1d5cf250 size:   30 previous size:    0  (Allocated)  FOCX
 ffffe18f1d5cf280 size:   30 previous size:    0  (Allocated)  FOCX
 ffffe18f1d5cf2b0 size:   30 previous size:    0  (Allocated)  IoFE
 ffffe18f1d5cf2e0 size:   30 previous size:    0  (Allocated)  IoFE
 ffffe18f1d5cf310 size:   30 previous size:    0  (Free)       IoUs
 ffffe18f1d5cf340 size:   30 previous size:    0  (Free)       IoUs
 ffffe18f1d5cf370 size:   30 previous size:    0  (Allocated)  IoFE
 ffffe18f1d5cf3a0 size:   30 previous size:    0  (Allocated)  IoFE
 ffffe18f1d5cf3d0 size:   30 previous size:    0  (Allocated)  IoCc
 ffffe18f1d5cf400 size:   30 previous size:    0  (Allocated)  FOCX
 ffffe18f1d5cf430 size:   30 previous size:    0  (Free)       IoUs
 ffffe18f1d5cf460 size:   30 previous size:    0  (Free)       IoUs
 ffffe18f1d5cf490 size:   30 previous size:    0  (Free)       IoUs
 ffffe18f1d5cf4c0 size:   30 previous size:    0  (Allocated)  IoFE
 ffffe18f1d5cf4f0 size:   30 previous size:    0  (Allocated)  FOCX
 ffffe18f1d5cf520 size:   30 previous size:    0  (Free)       Ipng
 ffffe18f1d5cf550 size:   30 previous size:    0  (Free)       IoCc
 ffffe18f1d5cf580 size:   30 previous size:    0  (Allocated)  FOCX
 ffffe18f1d5cf5b0 size:   30 previous size:    0  (Allocated)  FOCX
 ffffe18f1d5cf5e0 size:   30 previous size:    0  (Free)       CTMM
 ffffe18f1d5cf610 size:   30 previous size:    0  (Allocated)  IoFE
```

```
 ffffe18f1d5cf640 size:   30 previous size:    0  (Free)       IoFE
 ffffe18f1d5cf670 size:   30 previous size:    0  (Allocated)  IoFE
 ffffe18f1d5cf6a0 size:   30 previous size:    0  (Allocated)  IoFE
 ffffe18f1d5cf6d0 size:   30 previous size:    0  (Allocated)  FOCX
*ffffe18f1d5cf700 size:   30 previous size:    0  (Allocated) *Drht
                Owning component : Unknown (update pooltag.txt)
 ffffe18f1d5cf730 size:   30 previous size:    0  (Allocated)  IoCc
 ffffe18f1d5cf760 size:   30 previous size:    0  (Free)       IoUs
 ffffe18f1d5cf790 size:   30 previous size:    0  (Free)       IoUs
 ffffe18f1d5cf7c0 size:   30 previous size:    0  (Free)       FOCX
 ffffe18f1d5cf7f0 size:   30 previous size:    0  (Allocated)  IoFE
 ffffe18f1d5cf820 size:   30 previous size:    0  (Allocated)  FOCX
 ffffe18f1d5cf850 size:   30 previous size:    0  (Free)       IoFE
 ffffe18f1d5cf880 size:   30 previous size:    0  (Allocated)  FOCX
 ffffe18f1d5cf8b0 size:   30 previous size:    0  (Free)       IoUs
 ffffe18f1d5cf8e0 size:   30 previous size:    0  (Allocated)  IoFE
 ffffe18f1d5cf910 size:   30 previous size:    0  (Free)       FOCX
 ffffe18f1d5cf940 size:   30 previous size:    0  (Allocated)  IoFE
 ffffe18f1d5cf970 size:   30 previous size:    0  (Allocated)  FOCX
 ffffe18f1d5cf9a0 size:   30 previous size:    0  (Allocated)  NDFL
 ffffe18f1d5cf9d0 size:   30 previous size:    0  (Allocated)  FOCX
 ffffe18f1d5cfa00 size:   30 previous size:    0  (Free)       IoFE
 ffffe18f1d5cfa30 size:   30 previous size:    0  (Allocated)  IoFE
 ffffe18f1d5cfa60 size:   30 previous size:    0  (Free)       IoFE
 ffffe18f1d5cfa90 size:   30 previous size:    0  (Allocated)  FOCX
 ffffe18f1d5cfac0 size:   30 previous size:    0  (Allocated)  FOCX
 ffffe18f1d5cfaf0 size:   30 previous size:    0  (Allocated)  IoFE
 ffffe18f1d5cfb20 size:   30 previous size:    0  (Free)       FOCX
 ffffe18f1d5cfb50 size:   30 previous size:    0  (Free)       FOCX
 ffffe18f1d5cfb80 size:   30 previous size:    0  (Free)       IoFE
 ffffe18f1d5cfbb0 size:   30 previous size:    0  (Allocated)  FOCX
 ffffe18f1d5cfbe0 size:   30 previous size:    0  (Allocated)  IoFE
 ffffe18f1d5cfc10 size:   30 previous size:    0  (Free)       FOCX
 ffffe18f1d5cfc40 size:   30 previous size:    0  (Free)       IoFE
 ffffe18f1d5cfc70 size:   30 previous size:    0  (Free)       FOCX
 ffffe18f1d5cfca0 size:   30 previous size:    0  (Free)       FOCX
 ffffe18f1d5cfcd0 size:   30 previous size:    0  (Free)       IoFE
 ffffe18f1d5cfd00 size:   30 previous size:    0  (Allocated)  FOCX
 ffffe18f1d5cfd30 size:   30 previous size:    0  (Free)       FOCX
 ffffe18f1d5cfd60 size:   30 previous size:    0  (Allocated)  FOCX
 ffffe18f1d5cfd90 size:   30 previous size:    0  (Allocated)  IoFE
 ffffe18f1d5cfdc0 size:   30 previous size:    0  (Allocated)  FOCX
 ffffe18f1d5cfdf0 size:   30 previous size:    0  (Allocated)  IoFE
 ffffe18f1d5cfe20 size:   30 previous size:    0  (Allocated)  IoFE
 ffffe18f1d5cfe50 size:   30 previous size:    0  (Free)       IoFE
 ffffe18f1d5cfe80 size:   30 previous size:    0  (Allocated)  FOCX
 ffffe18f1d5cfeb0 size:   30 previous size:    0  (Allocated)  FOCX
 ffffe18f1d5cfee0 size:   30 previous size:    0  (Allocated)  IoFE
 ffffe18f1d5cff10 size:   30 previous size:    0  (Free)       FOCX
 ffffe18f1d5cff40 size:   30 previous size:    0  (Free)       IoFE
 ffffe18f1d5cff70 size:   30 previous size:    0  (Free)       IoFE
 ffffe18f1d5cffa0 size:   30 previous size:    0  (Free)       IoFE
 ffffe18f1d5cffd0 size:   30 previous size:    0  (Allocated)  FOCX
```

The one we're interested in appears to be:

```
...
*ffffe18f1d5cf700 size:   30 previous size:    0  (Allocated) *Drht
...
```

So we inspect the `_POOL_HEADER` structure for that pool allocation:

```
0: kd> dt nt!_POOL_HEADER ffffe18f1d5cf700
   +0x000 PreviousSize     : 0y00000000 (0)
   +0x000 PoolIndex        : 0y00000000 (0)
   +0x002 BlockSize        : 0y00000011 (0x3)
   +0x002 PoolType         : 0y00000010 (0x2)
   +0x000 Ulong1           : 0x2030000
   +0x004 PoolTag          : 0x74687244
   +0x008 ProcessBilled    : (null)
   +0x008 AllocatorBackTraceIndex : 0
   +0x00a PoolTagHash      : 0
```

As we can see, `ProcessBilled` is null because this was a standard allocation without `POOL_QUOTA`.
Why? Because some allocations don't use `POOL_QUOTA`. In fact, by default, allocations are made without it.
This means the kernel does not keep track of which process owns the allocated pool memory.
As a result, the `ProcessBilled` field is not filled in.

NOTE: When using `POOL_QUOTA`, the `_EPROCESS` pointer stored in `ProcessBilled` is obfuscated via XOR with a random cookie.

Segment Heap

The segment heap is code (everything is, really) that provides different behaviors based on the size of the allocation.

First, let's look at the top-level structure of the heap, `_HEAP`:

```
//0x2c0 bytes (sizeof)
struct _HEAP
{
    union
    {
        struct _HEAP_SEGMENT Segment;                                       //0x
        struct
        {
            struct _HEAP_ENTRY Entry;                                       //0x
            ULONG SegmentSignature;                                         //0x
            ULONG SegmentFlags;                                             //0x
            struct _LIST_ENTRY SegmentListEntry;                            //0x
            struct _HEAP* Heap;                                             //0x
            VOID* BaseAddress;                                              //0x
            ULONG NumberOfPages;                                            //0x
            struct _HEAP_ENTRY* FirstEntry;                                 //0x
            struct _HEAP_ENTRY* LastValidEntry;                             //0x
            ULONG NumberOfUnCommittedPages;                                 //0x
            ULONG NumberOfUnCommittedRanges;                                //0x
            USHORT SegmentAllocatorBackTraceIndex;                          //0x
            USHORT Reserved;                                                //0x
            struct _LIST_ENTRY UCRSegmentList;                              //0x
        };
    };
    ULONG Flags;                                                            //0x
    ULONG ForceFlags;                                                       //0x
    ULONG CompatibilityFlags;                                               //0x
    ULONG EncodeFlagMask;                                                   //0x
    struct _HEAP_ENTRY Encoding;                                            //0x
    ULONG Interceptor;                                                      //0x
    ULONG VirtualMemoryThreshold;                                           //0x
    ULONG Signature;                                                        //0x
    ULONGLONG SegmentReserve;                                               //0x
    ULONGLONG SegmentCommit;                                                //0x
    ULONGLONG DeCommitFreeBlockThreshold;                                   //0x
    ULONGLONG DeCommitTotalFreeThreshold;                                   //0x
    ULONGLONG TotalFreeSize;                                                //0x
    ULONGLONG MaximumAllocationSize;                                        //0x
    USHORT ProcessHeapsListIndex;                                           //0x
    USHORT HeaderValidateLength;                                            //0x
    VOID* HeaderValidateCopy;                                               //0x
    USHORT NextAvailableTagIndex;                                           //0x
    USHORT MaximumTagIndex;                                                 //0x
    struct _HEAP_TAG_ENTRY* TagEntries;                                     //0x
    struct _LIST_ENTRY UCRList;                                             //0x
    ULONGLONG AlignRound;                                                   //0x
    ULONGLONG AlignMask;                                                    //0x
    struct _LIST_ENTRY VirtualAllocdBlocks;                                 //0x
    struct _LIST_ENTRY SegmentList;                                         //0x
    USHORT AllocatorBackTraceIndex;                                         //0x
    ULONG NonDedicatedListLength;                                           //0x
    VOID* BlocksIndex;                                                      //0x
    VOID* UCRIndex;                                                         //0x
    struct _HEAP_PSEUDO_TAG_ENTRY* PseudoTagEntries;                        //0x
    struct _LIST_ENTRY FreeLists;                                           //0x
    struct _HEAP_LOCK* LockVariable;                                        //0x
    LONG (*CommitRoutine)(VOID* arg1, VOID** arg2, ULONGLONG* arg3);        //0x
    union _RTL_RUN_ONCE StackTraceInitVar;                                  //0x
    struct _RTLP_HEAP_COMMIT_LIMIT_DATA CommitLimitData;                    //0x
    VOID* UserContext;                                                      //0x
    ULONGLONG Spare;                                                        //0x
    VOID* FrontEndHeap;                                                     //0x
    USHORT FrontHeapLockCount;                                              //0x
    UCHAR FrontEndHeapType;                                                 //0x
    UCHAR RequestedFrontEndHeapType;                                        //0x
    USHORT* FrontEndHeapUsageData;                                          //0x
    USHORT FrontEndHeapMaximumIndex;                                        //0x
    volatile UCHAR FrontEndHeapStatusBitmap[129];                           //0x
    union
    {
        UCHAR ReadOnly:1;                                                   //0x
        UCHAR InternalFlags;                                                //0x
    };
    struct _HEAP_COUNTERS Counters;                                         //0x
    struct _HEAP_TUNING_PARAMETERS TuningParameters;                        //0x
};
```

As shown, it contains the `_LIST_ENTRY` of all heap segments. There are four well-known types of these segments.

This explanation is based on this excellent paper.

To handle allocation sizes, there are four “types” of allocation engines (or internal mechanisms) that the segment heap uses depending on allocation size (From smallest to largest):

1. Low Fragmentation Heap (**LFH**) -> alloc ≤ `0x200` (≤ 512B)
2. Variable Size (**VS**) -> alloc ≤ `0x20000` (513B – 128 KiB)
3. Segment Alloc -> alloc ≤ `0x7f000` (128 KiB – ~7 MiB)
4. Large Alloc -> alloc ≤ `0x200` (> ~7 MiB)

`_SEGMENT_HEAP`

is a kernel data structure that represents an instance of the segment heap. It is present in both user mode and kernel mode. It contains pointers and global configs, internal engines (LFH, VS, Segment), limits, segment lists… In summary, it organizes all the backends of the segment heap (pointers to `_HEAP_LFH_CONTEXT`, `_HEAP_VS_CONTEXT`, and `_HEAP_SEG_CONTEXT`), and also includes signatures and encoded pointers that hinder exploitation, among other things.

The structure is as follows:

```
0: 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
   +0x01e ReadOnly         : Pos 1, 1 Bit
   +0x01e InternalFlags    : Uint2B
   +0x020 CommitLimitData  : _RTLP_HEAP_COMMIT_LIMIT_DATA
   +0x030 ReservedMustBeZero : Uint8B
   +0x038 UserContext      : Ptr64 Void
   +0x040 LargeMetadataLock : Uint8B
   +0x048 LargeAllocMetadata : _RTL_RB_TREE
   +0x058 LargeReservedPages : Uint8B
   +0x060 LargeCommittedPages : Uint8B
   +0x068 Tag              : Uint8B
   +0x070 StackTraceInitVar : _RTL_RUN_ONCE
   +0x080 MemStats         : _HEAP_RUNTIME_MEMORY_STATS
   +0x0e0 GlobalLockOwner  : Uint4B
   +0x0e8 ContextExtendLock : Uint8B
   +0x0f0 AllocatedBase    : Ptr64 UChar
   +0x0f8 UncommittedBase  : Ptr64 UChar
   +0x100 ReservedLimit    : Ptr64 UChar
   +0x108 ReservedRegionEnd : Ptr64 UChar
   +0x110 CallbacksEncoded : _RTL_HP_HEAP_VA_CALLBACKS_ENCODED
   +0x140 SegContexts      : [2] _HEAP_SEG_CONTEXT
   +0x2c0 VsContext        : _HEAP_VS_CONTEXT
   +0x340 LfhContext       : _HEAP_LFH_CONTEXT
```

However, the structure from Vergilius sheds more light:

```
//0x70 bytes (sizeof)
struct _HEAP_SEGMENT
{
    struct _HEAP_ENTRY Entry;                                       //0
    ULONG SegmentSignature;                                         //0
    ULONG SegmentFlags;                                             //0
    struct _LIST_ENTRY SegmentListEntry;                            //0
    struct _HEAP* Heap;                                             //0
    VOID* BaseAddress;                                              //0
    ULONG NumberOfPages;                                            //0
    struct _HEAP_ENTRY* FirstEntry;                                 //0
    struct _HEAP_ENTRY* LastValidEntry;                             //0
    ULONG NumberOfUnCommittedPages;                                 //0
    ULONG NumberOfUnCommittedRanges;                                //0
    USHORT SegmentAllocatorBackTraceIndex;                          //0
    USHORT Reserved;                                                //0
    struct _LIST_ENTRY UCRSegmentList;                              //0
};
```

At first glance, there are four known structures for the different `_POOL_TYPE`s:

- `NonPaged`

- NonPagedNx
- Paged Pools
- Paged Session Pool

The first three are stored in `HEAP_POOL_NODES`

Segment Backend

The segment backend context is as follows:

```
0: kd> dt nt!_HEAP_SEG_CONTEXT
   +0x000 SegmentMask      : Uint8B
   +0x008 UnitShift        : UChar
   +0x009 PagesPerUnitShift : UChar
   +0x00a FirstDescriptorIndex : UChar
   +0x00b CachedCommitSoftShift : UChar
   +0x00c CachedCommitHighShift : UChar
   +0x00d Flags            : <unnamed-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 is a part of the heap manager that handles large allocations, and by large we mean from 128 kilobytes up to 7 gigabytes.

These sizes are too big to be managed by other backends like **VS** or **LFH**, as we've seen before.

Looking at the structure in Vergilius, it becomes clearer:

```
   //0xc0 bytes (sizeof)
   struct _HEAP_SEG_CONTEXT
   {
       ULONGLONG SegmentMask;                                  //0
       UCHAR UnitShift;                                        //0
       UCHAR PagesPerUnitShift;                                //0
       UCHAR FirstDescriptorIndex;                             //0
       UCHAR CachedCommitSoftShift;                            //0
       UCHAR CachedCommitHighShift;                            //0
       union
       {
           UCHAR LargePagePolicy:3;                            //0
           UCHAR FullDecommit:1;                               //0
           UCHAR ReleaseEmptySegments:1;                       //0
           UCHAR AllFlags;                                     //0
       } Flags;                                                //0
       ULONG MaxAllocationSize;                                //0
       SHORT OlpStatsOffset;                                   //0
       SHORT MemStatsOffset;                                   //0
       VOID* LfhContext;                                       //0
       VOID* VsContext;                                        //0
       struct RTL_HP_ENV_HANDLE EnvHandle;                     //0
       VOID* Heap;                                             //0
       ULONGLONG SegmentLock;                                  //0
       struct _LIST_ENTRY SegmentListHead;                     //0
       ULONGLONG SegmentCount;                                 //0
       struct _RTL_RB_TREE FreePageRanges;                     //0
       ULONGLONG FreeSegmentListLock;                          //0
       struct _SINGLE_LIST_ENTRY FreeSegmentList[2];           //0
   };
```

VS and LFH manage allocations visible to the user or kernel, and they in turn request large memory blocks from the segment backend to subdivide them into smaller chunks as needed.

In other words, the segment heap acts like a (loosely speaking) "lower ring" that provides large memory chunks for **VS** and **LFH** to subdivide.

The segment backend allocates memory in variable-sized blocks called segments, each composed of multiple assignable pages, all managed using a Red-Black tree `FreePageRanges` -> `_RTL_RB_TREE`

```
//0x10 bytes (sizeof)
struct _RTL_RB_TREE
{
    struct _RTL_BALANCED_NODE* Root;                                        //
    union
    {
        UCHAR Encoded:1;                                                    //
        struct _RTL_BALANCED_NODE* Min;                                     //
    };
};
```

As previously mentioned, the segments are stored in a list in `SegmentListHead`, and are preceded by a `_HEAP_PAGE_SEGMENT` structure followed by 256 `_HEAP_PAGE_RANGE_DESCRIPTOR`, as shown below:

```
0: 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
0: 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 UnitOffset       : UChar
   +0x01b Spare            : UChar
   +0x01c Key              : _HEAP_DESCRIPTOR_KEY
   +0x01c Align            : [3] UChar
   +0x01f UnitSize         : UChar
```

And in Vergilius:

```
//0x2000 bytes (sizeof)
union _HEAP_PAGE_SEGMENT
{
    struct
    {
        struct _LIST_ENTRY ListEntry;                                       //0
        ULONGLONG Signature;                                                //0
    };
    struct
    {
        union _HEAP_SEGMENT_MGR_COMMIT_STATE* SegmentCommitState;           //0
        UCHAR UnusedWatermark;                                              //0
    };
    struct _HEAP_PAGE_RANGE_DESCRIPTOR DescArray[256];                      //0
};
```

The `_HEAP_PAGE_RANGE_DESCRIPTOR` is:

```
//0x20 bytes (sizeof)
struct _HEAP_PAGE_RANGE_DESCRIPTOR
{
    union
    {
        struct _RTL_BALANCED_NODE TreeNode;                                 //0
        struct
        {
            ULONG TreeSignature;                                            //0
            ULONG UnusedBytes;                                              //0
            USHORT ExtraPresent:1;                                          //0
            USHORT Spare0:15;                                               //0
        };
    };
    volatile UCHAR RangeFlags;                                              //0
    UCHAR CommittedPageCount;                                               //0
    UCHAR UnitOffset;                                                       //0
    UCHAR Spare;                                                            //0
    union
    {
        struct _HEAP_DESCRIPTOR_KEY Key;                                    //0
        struct
        {
            UCHAR Align[3];                                                 //0
            UCHAR UnitSize;                                                 //0
        };
    };
};
```

Each `_HEAP_PAGE_SEGMENT` has a unique signature computed via XOR with several pointers and a constant, which helps validate the segment and recover its context.

Variable Size Backend

Variable Size or **VS** backend allocates blocks from 512 bytes up to 128 kilobytes, as seen before. It enables reuse of freed blocks, and its context resides in the `_HEAP_VS_CONTEXT` structure:

```
//0xc0 bytes (sizeof)
struct _HEAP_VS_CONTEXT
{
    ULONGLONG Lock;                                                         //0
    enum _RTLP_HP_LOCK_TYPE LockType;                                       //0
    SHORT MemStatsOffset;                                                   //0
    struct _RTL_RB_TREE FreeChunkTree;                                      //0
    struct _LIST_ENTRY SubsegmentList;                                      //0
    ULONGLONG TotalCommittedUnits;                                          //0
    ULONGLONG FreeCommittedUnits;                                           //0
    struct _HEAP_VS_DELAY_FREE_CONTEXT DelayFreeContext;                    //0
    VOID* BackendCtx;                                                       //0
    struct _HEAP_SUBALLOCATOR_CALLBACKS Callbacks;                          //0
    struct _RTL_HP_VS_CONFIG Config;                                        //0
    ULONG EliminatePointers:1;                                              //0
};
```

VS uses a Red-Black tree to efficiently locate and organize free blocks via `FreeChunkTree`.

Each memory block has a header encrypted with XOR for integrity, and if the found chunk is larger than needed, it is dynamically split. If no suitable chunk is available, the backend requests a new subsegment from the **Segment Backend** (as previously discussed).

Free blocks are preceded by a structure called `_HEAP_VS_CHUNK_FREE_HEADER`:

```
//0x20 bytes (sizeof)
struct _HEAP_VS_CHUNK_FREE_HEADER
{
    union
    {
        struct _HEAP_VS_CHUNK_HEADER Header;                            //
        struct
        {
            ULONGLONG OverlapsHeader;                                   //
            struct _RTL_BALANCED_NODE Node;                             //
        };
    };
};
```

Once the free block is found, it is split to the appropriate size via **`RtlpHpVsChunkSplit()`**
{thanks again to C. Bayet, P. Fariello :) }

The allocated block is preceded by the `_HEAP_VS_CHUNK_HEADER` structure:

```
//0x10 bytes (sizeof)
struct _HEAP_VS_CHUNK_HEADER
{
    union _HEAP_VS_CHUNK_HEADER_SIZE Sizes;                             //
    union
    {
        struct
        {
            ULONG EncodedSegmentPageOffset:8;                           //
            ULONG UnusedBytes:1;                                        //
            ULONG SkipDuringWalk:1;                                     //
            ULONG Spare:22;                                             //
        };
        ULONG AllocatedChunkBits;                                       //
    };
};
```

All header fields are XORed with `RtlpHpHeapGlobals`.

The **VS** allocator internally depends on the segment allocator to obtain large raw memory blocks (subsegments).

When **VS** doesn't find a suitable free chunk in the tree (`FreeChunkTree`), it reserves more memory from the OS through the Segment allocator using `_HEAP_SUBALLOCATOR_CALLBACKS`, found inside `_HEAP_VS_CONTEXT`:

```
//0x30 bytes (sizeof)
struct _HEAP_SUBALLOCATOR_CALLBACKS
{
    ULONGLONG Allocate;                                                 //0
    ULONGLONG Free;                                                     //0
    ULONGLONG Commit;                                                   //0
    ULONGLONG Decommit;                                                 //0
    ULONGLONG ExtendContext;                                            //0
    ULONGLONG TlsCleanup;                                               //0
};
```

`RtlpHpVsSubsegmentCreate()` is responsible for creating a new subsegment for the **VS** backend when no blocks are available in the `FreeChunkTree`:

```
__int64 __fastcall RtlpHpVsSubsegmentCreate(__int64 a1, int a2)
{
  __int64 v2; // rdi
  int v3; // r14d
  unsigned int v5; // edx
  unsigned int v6; // r14d
  unsigned int v7; // ecx
  unsigned int v8; // ebx
  __int64 v9; // rcx
  __int64 v10; // rax
  __int64 v11; // rbp
  unsigned int v12; // r14d
  __int64 v13; // rcx
  int v14; // eax
  __int64 v16; // rcx
  int v17; // [rsp+60h] [rbp+8h] BYREF
  unsigned int v18; // [rsp+68h] [rbp+10h] BYREF
  unsigned int v19; // [rsp+70h] [rbp+18h]

  v2 = 0;
  v3 = 16 * a2;
  v5 = 32 * a2 + 48;
  v18 = 0;
  v6 = (v3 + 4143) & 0xFFFFF000;
  v17 = 0;
  v19 = 0;
  if ( ((v5 - 1) & v5) != 0 )
  {
    _BitScanReverse(&v7, v5);
    v19 = v7;
    v5 = 1 << (v7 + 1);
  }
  v8 = 0x10000;
  if ( v5 > 0x10000 )
  {
    v8 = v5;
    if ( v5 >= 0x40000 )
      v8 = 0x40000;
  }
  while ( 1 )
  {
    v9 = *(_QWORD *)(a1 + 8) ^ a1;
    v10 = (__int64 (__fastcall *)(_QWORD, _QWORD, _QWORD, _QWORD))(a1 ^ RtlpHpHeapGlobals ^ *(_QWORD *)(a1 + 16)) == RtlpHpSegVsAlloc
        ? RtlpHpSegVsAllocate(v9, v8, &v17, &v18)
        : guard_dispatch_icall_no_overrides(v9);
    v11 = v10;
    if ( v10 )
      break;
    v8 = v18;
    if ( v18 < v6 )
      return v2;
  }
  v12 = 4096;
  if ( (v17 & 1) != 0 )
    v12 = v8;
  v13 = *(_QWORD *)(a1 + 8) ^ a1;
  if ( (__int64 (__fastcall *)(_QWORD, _QWORD, _QWORD, _QWORD))(a1 ^ RtlpHpHeapGlobals ^ *(_QWORD *)(a1 + 32)) == RtlpHpSegLfhVsCommi
    v14 = RtlpHpSegLfhVsCommit(v13, v10, v12, 0);
  else
    v14 = guard_dispatch_icall_no_overrides(v13);
  if ( v14 < 0 )
  {
    v16 = *(_QWORD *)(a1 + 8) ^ a1;
    if ( (__int64 (__fastcall *)(_QWORD, _QWORD, _QWORD))(a1 ^ RtlpHpHeapGlobals ^ *(_QWORD *)(a1 + 24)) == RtlpHpSegLfhVsFree )
      RtlpHpSegLfhVsFree(v16, v11, v8);
    else
      guard_dispatch_icall_no_overrides(v16);
  }
  else
  {
    _InterlockedAdd64((volatile signed __int64 *)(a1 + 80), (unsigned __int64)v12 >> 12);
    RtlpHpVsSubsegmentInitialize(v11, v8, v12);
    return v11;
  }
  return v2;
}
```

This would be the `_HEAP_VS_SUBSEGMENT` structure:

```
//0x28 bytes (sizeof)
struct _HEAP_VS_SUBSEGMENT
{
    struct _LIST_ENTRY ListEntry;                                           //0
    ULONGLONG CommitBitmap;                                                 //0
    ULONGLONG CommitLock;                                                   //0
    USHORT Size;                                                            //0
    USHORT Signature:15;                                                    //0
    USHORT FullCommit:1;                                                    //0
};
```

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.

Lastly, when a **VS** block is truly freed, if it's contiguous with two other freed blocks, the three are merged by calling **`RtlpHpVsChunkCoalesce()`**, and then inserted back into the `FreeChunkTree`.

However, during my research in Windows 24H2, I couldn't find **`RtlHpVsChunkCoalesce()`**, but I did find a function with similar behavior: **`RtlpHpVsSlotCompactChunks()`**, which apparently serves a similar purpose.

Low Fragmentation Heap Backend

The low fragmentation heap is a backend dedicated to small allocations ranging from 1 byte to 512 bytes.

Below is the `_HEAP_LFH_CONTEXT`:

```
//0x6c0 bytes (sizeof)
struct _HEAP_LFH_CONTEXT
{
    VOID* BackendCtx;                                                       //0x
    struct _HEAP_SUBALLOCATOR_CALLBACKS Callbacks;                          //0x
    UCHAR* AffinityModArray;                                                //0x
    UCHAR MaxAffinity;                                                      //0x
    UCHAR LockType;                                                         //0x
    SHORT MemStatsOffset;                                                   //0x
    struct _HEAP_LFH_CONFIG Config;                                         //0x
    ULONG TlsSlotIndex;                                                     //0x
    ULONGLONG EncodeKey;                                                    //0x
    ULONGLONG ExtensionLock;                                                //0x
    struct _SINGLE_LIST_ENTRY MetadataList[4];                              //0x
    struct _HEAP_LFH_HEAT_MAP HeatMap;                                      //0x
    struct _HEAP_LFH_BUCKET* Buckets[128];                                  //0x
    struct _HEAP_LFH_SLOT_MAP SlotMaps[1];                                  //0x
};
```

LFH subdivides the maximum memory (512 bytes) into 129 buckets with increasing granularities:

- Granularity 16 bytes: Index 1-64 -> 1B - 1008B allocation
- Granularity 64 bytes: Index 65-80 -> 1009B - 2032B allocation
- Granularity 128 bytes: Index 81-96 -> 2033B - 4080B allocation
- Granularity 256 bytes: Index 97-112 -> 4081B - 8176B allocation
- Granularity 512 bytes: Index 113-128 -> 8177B - 16368B allocation

Each **LFH** bucket is composed of one or more subsegments obtained via the segment allocator, invoked through the `_HEAP_SUBALLOCATOR_CALLBACKS` field inside `_HEAP_LFH_CONTEXT`, which contains function pointers for `Allocate`, `Free`, `Commit`, and `Decommit`.

```
//0x30 bytes (sizeof)
struct _HEAP_SUBALLOCATOR_CALLBACKS
{
    ULONGLONG Allocate;                                                     //0
    ULONGLONG Free;                                                         //0
    ULONGLONG Commit;                                                       //0
    ULONGLONG Decommit;                                                     //0
    ULONGLONG ExtendContext;                                                //0
    ULONGLONG TlsCleanup;                                                   //0
};
```

To protect the mentioned pointers, they are XORed with the context address and with `RtlpHpHeapGlobals`.

Once a subsegment is assigned, it is internally divided into multiple LFH blocks of the bucket's size. Ex: a subsegment for 128-byte blocks will be entirely divided into 128-byte blocks.

The `_HEAP_LFH_SUBSEGMENT` structure is the header of each subsegment and contains essential data:

```
//0x48 bytes (sizeof)
struct _HEAP_LFH_SUBSEGMENT
{
    struct _LIST_ENTRY ListEntry;                                           //0
    union _HEAP_LFH_SUBSEGMENT_STATE State;                                 //0
    union
    {
        struct _SINGLE_LIST_ENTRY OwnerFreeListEntry;                       //0
        struct
        {
            UCHAR CommitStateOffset;                                        //0
            UCHAR Spare0:4;                                                 //0
        };
    };
    USHORT FreeCount;                                                       //0
    USHORT BlockCount;                                                      //0
    UCHAR FreeHint;                                                         //0
    UCHAR WitheldBlockCount;                                                //0
    union
    {
        struct
        {
            UCHAR CommitUnitShift;                                          //0
            UCHAR CommitUnitCount;                                          //0
        };
        union _HEAP_LFH_COMMIT_UNIT_INFO CommitUnitInfo;                    //0
    };
    struct _HEAP_LFH_SUBSEGMENT_ENCODED_OFFSETS BlockOffsets;               //0
    USHORT BucketRef;                                                       //0
    USHORT PrivateSlotMapRef;                                               //0
    USHORT HighWatermarkBlockIndex;                                         //0
    UCHAR BitmapSearchWidth;                                                //0
    union
    {
        struct
        {
            UCHAR PrivateFormat:1;                                          //0
            UCHAR Spare1:7;                                                 //0
        };
        union _HEAP_LFH_SUBSEGMENT_UCHAR_FIELDS UChar;                      //0
    };
    ULONG Spare3;                                                           //0
    ULONGLONG CommitLock;                                                   //0
    ULONGLONG BlockBitmap[1];                                               //0
};
```

To track which blocks in a subsegment are free or occupied, the `BlockBitmap` is used, where each bit represents a block, 0 (off) for free, 1 (on) for allocated. There's also a field called `FreeHint`, which points to the last freed block and acts as a checkpoint to scan the bitmap for the next free block.

IMPORTANT: The bitmap scan is randomized using a table. This helps distribute allocations across the entire subsegment, preventing predictable patterns and improving security against heap feng shui-style techniques.

Dynamic Lookaside

An optimization mechanism for the heap allocator designed to speed up medium-sized memory allocations and deallocations (`0x200` (512 bytes) to `0xF80` (3968 bytes)).
It works by temporarily storing blocks in a lookaside list for faster future allocations. This allows blocks of the same size to be reallocated without extra cost.

Managed by `_RTL_DYNAMIC_LOOKASIDE`, which is referenced from the `UserContext` field in `_SEGMENT_HEAP`:

```
//0x1040 bytes (sizeof)
struct _RTL_DYNAMIC_LOOKASIDE
{
    ULONGLONG EnabledBucketBitmap;                                  //0
    ULONG BucketCount;                                              //0
    ULONG ActiveBucketCount;                                        //0
    struct _RTL_LOOKASIDE Buckets[64];                              //0
};
```

This structure contains up to 64 individual lists called `_RTL_LOOKASIDE`, each representing a block size class:

```
//0x40 bytes (sizeof)
struct _RTL_LOOKASIDE
{
    union _SLIST_HEADER ListHead;                                   //0
    USHORT Depth;                                                   //0
    USHORT MaximumDepth;                                            //0
    ULONG TotalAllocates;                                           //0
    ULONG AllocateMisses;                                           //0
    ULONG TotalFrees;                                               //0
    ULONG FreeMisses;                                               //0
    ULONG LastTotalAllocates;                                       //0
    ULONG LastAllocateMisses;                                       //0
    ULONG LastTotalFrees;                                           //0
};
```

The sizes follow this granularity:

- Granularity 16 bytes: Index 1-32 -> 512B - 1024B allocation
- Granularity 64 bytes: Index 33-48 -> 1025B - 2048B allocation
- Granularity 128 bytes: Index 49-64 -> 2049B - 3967 allocation

When a block is freed, the system ensures the block size fits Dynamic Lookaside (512B to 3967B). If the bucket is enabled, the block is placed into its `_RTL_LOOKASIDE` rather than returned to the heap. The block is ready for reuse if another similar-size block is requested. If the list is full (`Depth == MaximumDepth`), the block goes back to the regular backend.

To use it, the appropriate bucket is queried based on the requested size. If blocks are available in the `ListHead`, allocation is instant. If not, it falls back to the regular backend.

I won't go into more detail as this is not the main focus of the research, but in summary, it's a kind of "cache", instead of caching data, it caches memory allocations.

References

- Windows Internals 7th edition
- kernel-pool-exploitation-on-windows-7
- SSTIC2020-Article-pool_overflow_exploitation_since_windows_10_19h1-bayet_fariello

Closing

This overview of the Windows kernel pool and heap internals sets a solid base for future heap exploitation.

Good morning, and in case I don't see ya: Good afternoon, good evening, and good night!

Recently Updated

- VMware Guest To Host
- Tp-Link Router Deep Research
- Hyper-V Research
- Windows Kernel Pool Internals
- Modern (Kernel) Low Fragmentation Heap Exploitation

Trending Tags

kaslr pool hello hyperv junkcode patchguard router shellcode smep vmware

Contents

Further Reading