Secure Pool Internals : Dynamic KDP Behind The Hood

windows-internals.com/secure-pool

By Yarden Shafir

Starting with Windows `10` Redstone `5` (Version `1809` , Build `17763`), a lot has changed in the kernel pool. We won't talk about most of these changes, that will happen in a 70-something page paper that will be published at some point in the future when we can find enough time and ADHD meds to finish it.

One of the more exciting changes, which is being added in Version `2104` and above, is a new type of pool – the *secure pool*. In short, the secure pool is a pool managed by `Securekernel.exe` , which operates in Virtual Trust Level `1` (VTL `1`), and that cannot be directly modified by anything running in VTL `0` . The idea is to allow drivers to keep sensitive information in a location where it is safe from tampering, even by other drivers. Dave Weston first announced this feature, marketed as Kernel Data Protection (KDP), at his BlueHat Shanghai talk in 2019 and Microsoft recently published a blog post presenting it and some of its internal details.

Note that there are two parts to the full KDP implementation: Static KDP, which refers to protecting read-only data sections in driver images, and Dynamic KDP, which refers to the secure pool, the topic of our blog post, which will talk about how to use this new pool and some implementation details, but will not discuss the general implementation of heaps or any of their components that are not specific to the secure pool.

We'll also mention three separate design flaw vulnerabilities that were found in the original implementation in Build `20124` , which were all fixed in `20161` . These were identified and fixed through Microsoft's great Windows Insider Preview Bug Bounty Program for $20000 USD each.

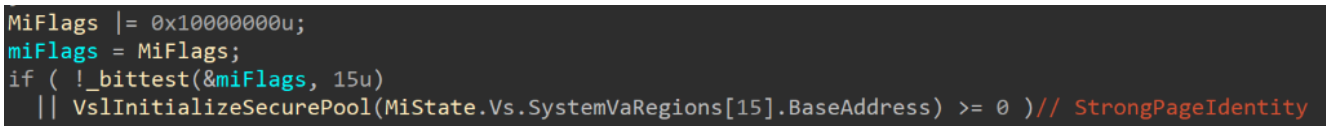
Initialization

The changes added for this new pool start at boot. In **`MiInitSystem`** we can now see a new check for bit `15` in `MiFlags` , which checks if secure pool is enabled on this machine. Since `MI_FLAGS` is now in the symbol files, we can see that it corresponds to:

```
+0x000 StrongPageIdentity : Pos 15, 1 Bit
```

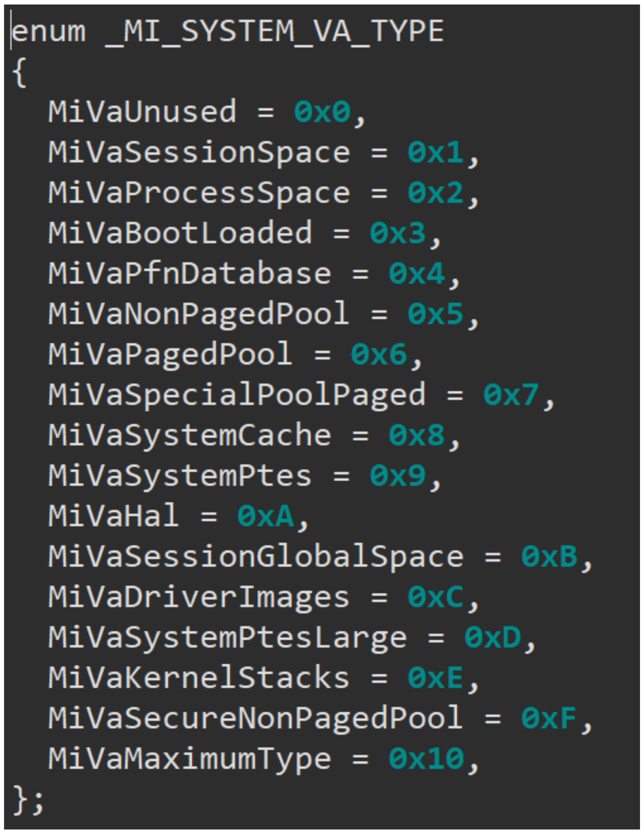
which is how the kernel knows that Virtualization Based Security (VBS) is enabled on a system with Secondary Level Address Table (SLAT) support. This allows the usage of Extended Page Table Entries (EPTEs) to add an additional, hypervisor-managed, layer of protection around physical memory. This is exactly what the secure pool will be relying on.

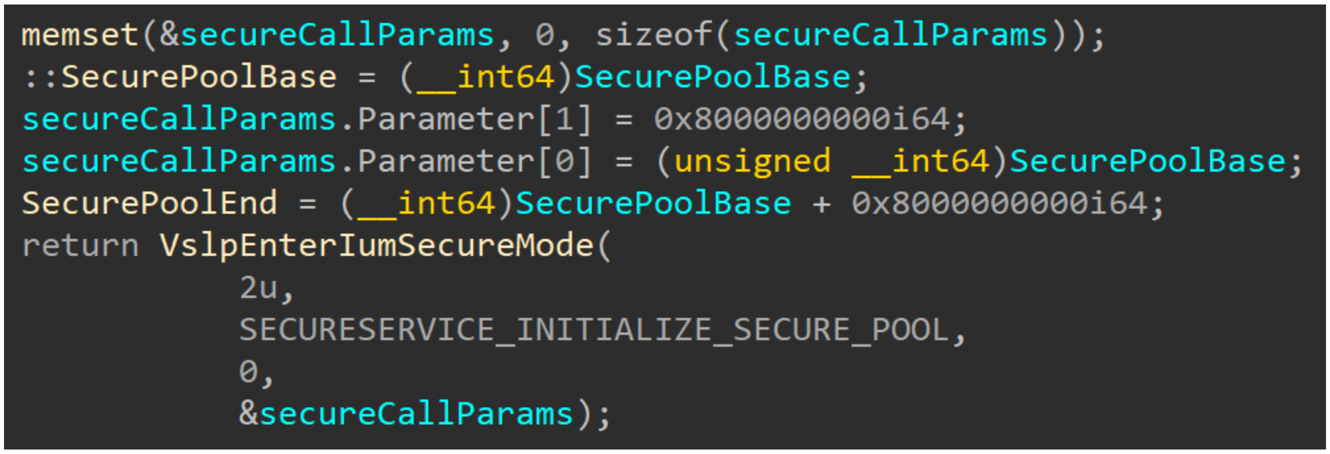
If the bit is set, **`MmInitSystem`** calls **`VslInitializeSecurePool`** , passing in `MiState.Vs.SystemVaRegions[MiVaSecureNonPagedPool].BaseAddress` :


```
MiFlags |= 0x10000000u;
miFlags = MiFlags;
if ( !_bittest(&miFlags, 15u)
  || VslInitializeSecurePool(MiState.Vs.SystemVaRegions[15].BaseAddress) >= 0 )// StrongPageIdentity
```

If we compare the symbol files and look at the `MI_SYSTEM_VA_TYPE` enum, we'll in fact see that a new member was added with a value of `15` : `MiVaSecureNonPagedPool` :

`VslInitializeSecurePool` initializes an internal structure sized `0x68` bytes with parameters for the secure call. This structure contains information used to make the secure call, such as the service code to be invoked and up to `12` parameters to be sent to `Securekernel` . In this case only `2` parameters are used – the requested size for the secure pool (`512` GB) and a pointer to receive its base address:

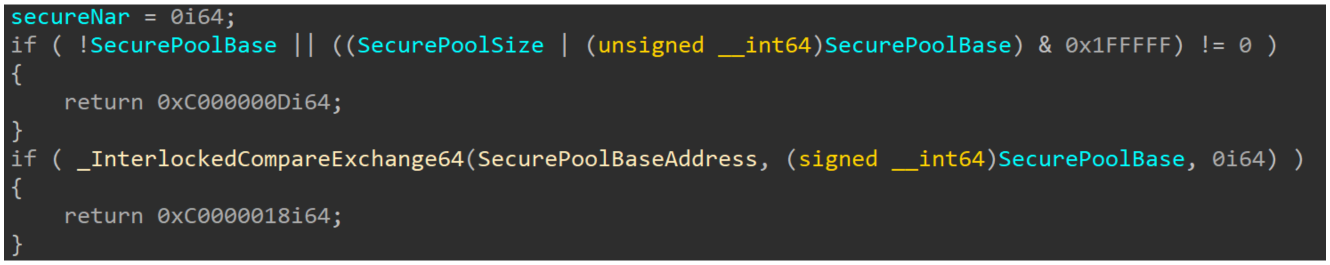

```
enum _MI_SYSTEM_VA_TYPE
{
  MiVaUnused = 0x0,
  MiVaSessionSpace = 0x1,
  MiVaProcessSpace = 0x2,
  MiVaBootLoaded = 0x3,
  MiVaPfnDatabase = 0x4,
  MiVaNonPagedPool = 0x5,
  MiVaPagedPool = 0x6,
  MiVaSpecialPoolPaged = 0x7,
  MiVaSystemCache = 0x8,
  MiVaSystemPtes = 0x9,
  MiVaHal = 0xA,
  MiVaSessionGlobalSpace = 0xB,
  MiVaDriverImages = 0xC,
  MiVaSystemPtesLarge = 0xD,
  MiVaKernelStacks = 0xE,
  MiVaSecureNonPagedPool = 0xF,
  MiVaMaximumType = 0x10,
};
```


```
memset(&secureCallParams, 0, sizeof(secureCallParams));
::SecurePoolBase = (__int64)SecurePoolBase;
secureCallParams.Parameter[1] = 0x8000000000i64;
secureCallParams.Parameter[0] = (unsigned __int64)SecurePoolBase;
SecurePoolEnd = (__int64)SecurePoolBase + 0x8000000000i64;
return VslpEnterIumSecureMode(
           2u,
           SECURESERVICE_INITIALIZE_SECURE_POOL,
           0,
           &secureCallParams);
```

It also initializes global variables **`SecurePoolBase`** and **`SecurePoolEnd`** , which will be used to validate secure pool handle (more on that later). Then it calls **`VslpEnterIumSecureMode`** to call into `SecureKernel` , which will initialize the secure pool itself, passing in the `secureCallParams` structure that contains that requested parameters. Before Alex's blog went down, he was working on an interesting series of posts on how the VTL `0` <-> VTL `1` communication infrastructure works, and hopefully it will return at some point, so we'll skip the details here.

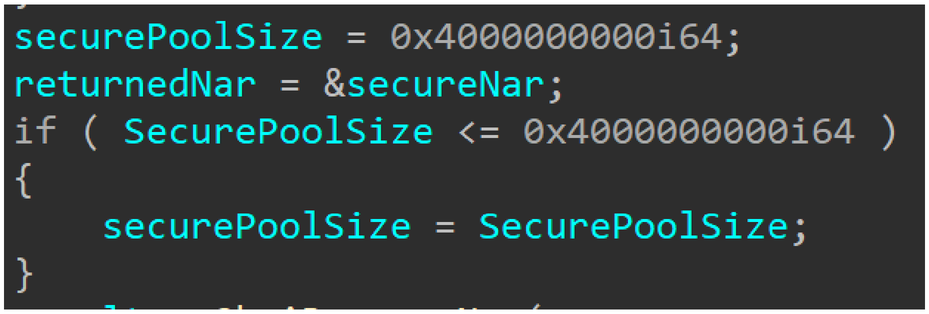
`Securekernel` unpacks the input parameters, finds the right path for the call, and eventually gets us to **`SkmmInitializeSecurePool`**. This function calls **`SecurePoolMgrInitialize`**, which does a few checks before initializing the pool.

First it validates that the input parameter `SecurePoolBase` is not zero and that it is aligned to `16` MB. Then it checks that the secure pool was not already initialized by checking if the global variable `SecurePoolBaseAddress` is empty:



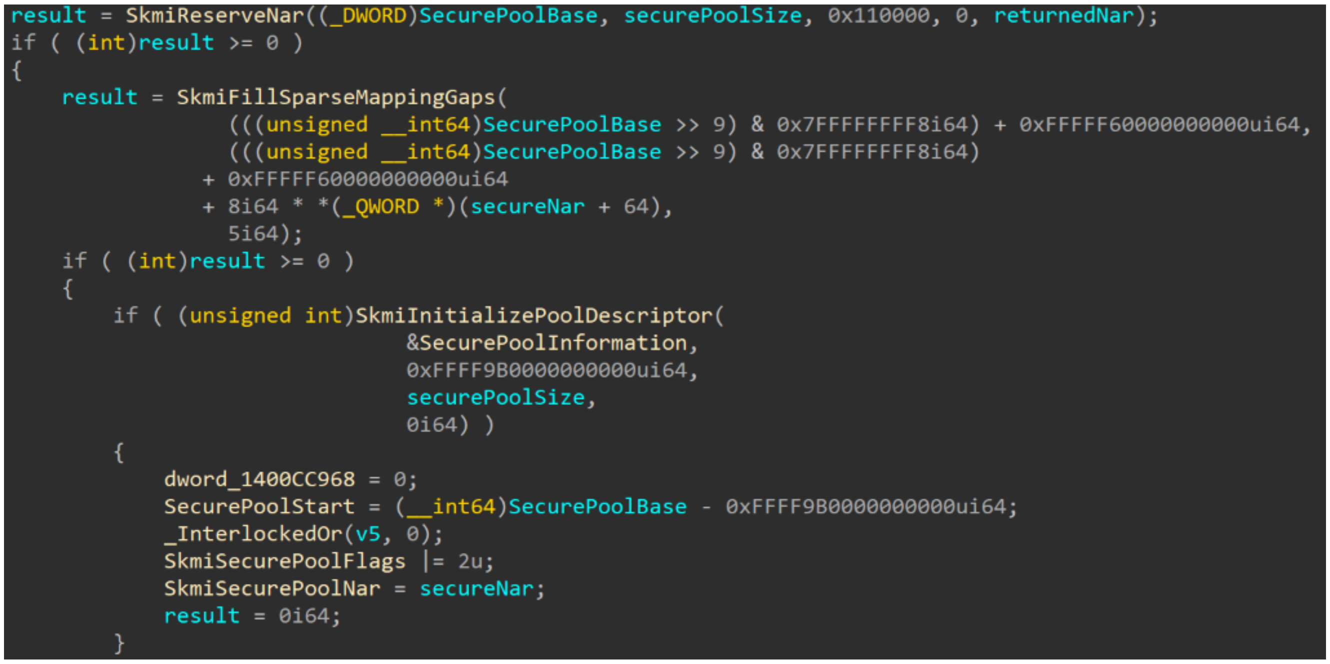
```
secureNar = 0i64;
if ( !SecurePoolBase || ((SecurePoolSize | (unsigned __int64)SecurePoolBase) & 0x1FFFFF) != 0 )
{
    return 0xC000000Di64;
}
if ( _InterlockedCompareExchange64(SecurePoolBaseAddress, (signed __int64)SecurePoolBase, 0i64) )
{
    return 0xC0000018i64;
}
```

The next check is for the size. If the supplied size is larger than `256 GB`, the function ignores the supplied size and sets it to `256` GB. This is explained in the blog post from Microsoft linked earlier, where the secure kernel is shown to use a `256` GB region for the kernel's `512` GB range. It's quote curious that this is done by having the caller supply `512` GB as a size, and the secure kernel ignoring the parameter and overriding it with `256`.



```
securePoolSize = 0x4000000000i64;
returnedNar = &secureNar;
if ( SecurePoolSize <= 0x4000000000i64 )
{
    securePoolSize = SecurePoolSize;
}
```

Once these checks are done **`SkmmInitializeSecurePool`** starts initializing the secure pool. It reserves a Normal Address Range (NAR) descriptor for the address range with **`SkmiReserveNar`** and then creates an initial pool descriptor and sets global variables `SkmiSecurePoolStart` and `SkmiSecurePoolNar`. Notice that the secure pool has a fixed, hard-coded address in `0xFFFF9B0000000000`:


```
result = SkmiReserveNar((_DWORD)SecurePoolBase, securePoolSize, 0x110000, 0, returnedNar);
if ( (int)result >= 0 )
{
    result = SkmiFillSparseMappingGaps(
               (((unsigned __int64)SecurePoolBase >> 9) & 0x7FFFFFFFF8i64) + 0xFFFFF60000000000ui64,
               (((unsigned __int64)SecurePoolBase >> 9) & 0x7FFFFFFFF8i64)
             + 0xFFFFF60000000000ui64
             + 8i64 * *(_QWORD *)(secureNar + 64),
               5i64);
    if ( (int)result >= 0 )
    {
        if ( (unsigned int)SkmiInitializePoolDescriptor(
                             &SecurePoolInformation,
                             0xFFFF9B0000000000ui64,
                             securePoolSize,
                             0i64) )
        {
            dword_1400CC968 = 0;
            SecurePoolStart = (__int64)SecurePoolBase - 0xFFFF9B0000000000ui64;
            _InterlockedOr(v5, 0);
            SkmiSecurePoolFlags |= 2u;
            SkmiSecurePoolNar = secureNar;
            result = 0i64;
        }
```

Side note: NAR stands for Normal Address Range. It's a data structure tracking kernel address space, like VADs are used for user-space memory. Windows Internals, 7th Edition, Part 2, has an amazing section on the secure kernel written by Andrea Allevi.

An interesting variable to look at here is `SkmiSecurePoolStart`, that gets a value of `<SecurePoolBaseInKernel> - <SecurePoolBaseInSecureKernel>`. Since the normal kernel and secure kernel have separate address spaces, the secure pool will be mapped in different addresses in each (as we've seen, it has a fixed address in the secure kernel and an ASLRed address in the normal kernel). This variable will allow `SecureKernel` to receive secure pool addresses from the normal kernel and translate them to secure kernel addresses, an ability that is necessary since this pool is meant to be used by the normal kernel and 3rd-party drivers.

After **`SkmmInitializeSecurePool`** returns there is another call to **`SkInitializeSecurePool`**, which calls **`SecurePoolMgrInitialize`**. This function initializes a pool state structure that we chose to call `SK_POOL_STATE` in the global variable `SecurePoolGlobalState`.

```
struct _SK_POOL_STATE

    LIST_ENTRY PoolLinks;
    PVOID Lock;
    RTLP_HP_HEAP_MANAGER HeapManager;
    PSEGMENT_HEAP SegmentHeap;
} SK_POOL_STATE, *PSK_POOL_STATE;
```

Then it starts the heap manager and initializes a bitmap that will be used to mark allocated addresses in the secure pool. Finally, **`SecurePoolMgrInitialize`** calls **`RtlpHpHeapCreate`** to allocate a heap and create a `SEGMENT_HEAP` for the secure pool.

The first design flaw in the original implementation is actually related to the `SEGMENT_HEAP` allocation. This is a subtle point unless someone has pre-read our 70 page book : due to how "metadata" allocations work, the `SEGMENT_HEAP` ended up being allocated as part of the secure pool, which, as per what we explained here and the Microsoft blog, means that it also ended up mapped in the VTL `0` region that encompasses the secure pool.

Since `SEGMENT_HEAP` contains pointers to certain functions owned by the heap manager (which, in the secure pool case, is hosted in `Securekernel.exe`), this resulted in an information leak vulnerability that could lead to the discovery of the VTL `1` base address of `SecureKernel.exe` (which is ASLRed).

This has now been fixed by no longer mapping the `SEGMENT_HEAP` structure in the VTL `0` region.

Creation & Destruction

Unlike the normal kernel pool, memory cannot be allocated from the secure pool directly as this would defeat the whole purpose. To get access to the secure pool, a driver first needs to call a new function – **`ExCreatePool`** . This function receives `Flags` , `Tag` , `Params` and an output parameter `Handle` . The function first validates the arguments:

- Flags must be equal to `3`
- Tag cannot be `0`
- Params must be `0`
- Handle cannot be `NULL`

After the arguments have been validates, the function makes a secure call to service `SECURESERVICE_SECURE_POOL_CREATE` , sending in the tag as the only parameter. This will reach the **`SkSpCreateSecurePool`** function in `Securekernel` . This function calls **`SkobCreateObject`** to allocate a secure object of type `SkSpStateType` , and then forwards the allocated structure together with the received `Tag` to **`SecurePoolInit`** , which will populate it. We chose to call this structure `SK_POOL` , and it contains the following fields:

```
struct _SK_POOL

    LIST_ENTRY PoolLinks;
    PSEGMENT_HEAP SegmentHeap;
    LONG64 PoolAllocs;
    ULONG64 Tag;
    PRTL_CSPARSE_BITMAP AllocBitmapTracker;
} SK_POOL, *PSK_POOL;
```

It then initializes `Tag` to the tag supplied by the caller, and `SegmentHeap` and `AllocBitmapTracker` to the heap and bitmap that were initialized at boot and is pointed to by `SecurePoolGlobalState.SegmentHeap` and a global variable `SecurePoolBitmapData` . This structure is added to a linked list stored in `SecurePoolGlobalState` , which we called `PoolLinks` , and will contain the number of allocations done from it (`PoolAllocs` is initially set to zero).

Finally, the function calls `SkobCreateHandle` to create a handle which will be returned to the caller. Now the caller can access the secure pool using this handle.

When the driver no longer needs access to the pool (usually right before unloading), it needs to call `ExDestroyPool` with the handle it received. This will reach `SecurePoolDestroy` which checks that this entry contains no allocations (`PoolAllocs = 0`) and wasn't modified (`PoolEntry.SegmentHeap == SecurePoolGlobalState.SegmentHeap`). If the validation was successful, the entry is removed from the list and the structure is freed. From that point the handle is no longer valid and cannot be used.

The second design bug identified in the original build was around what the `Handle` value contained. In the original design, `Handle` was an obfuscated value created through the XORing of certain virtual addresses, which was then validated (as you'll see in the Allocation section below) to point to a `SK_POOL` structure with the right fields filled out. However, due to the fact that the Secure Kernel does not use ASLR, the values part of the XOR computation were known to VTL `0` attackers.

Therefore, due to the fact that the contents of an `SK_POOL` can be inferred and built correctly (for the same reason), a VTL `0` attacker could first create a secure pool allocation that corresponds to a fake `SK_POOL` , compute the address of this allocation in the VTL `1` address range (since, as explained here and in Microsoft's blog post, there is a known delta), and then use the known XOR computation to supply this as a fake `Handle` to future Allocation, Update, Deallocation, and Destroy calls.

Among other things, this would allow an attacker to control operations such as the `PoolAllocs` counter shown earlier, which is incremented/decremented at various times, which would then corrupt an adjacent VTL `1` allocation or address (since only the first `16` bytes of `SK_POOL` are validated).

The fix, which is the new design shown here, leverages the Secure Kernel's Object Manager to allocate and define a real object, then to create a real secure handle associated with it. Secure objects/handles cannot be faked, other than stealing someone else's handle, but this results in VTL `0` data corruption, not VTL `1` arbitrary writes.

Allocation

After getting access to the secure pool, the driver can allocate memory through another new exported kernel function – `ExAllocatePool3` . Officially, this function is documented. But it is documented in such a useless way that it would almost be better if it wasn't documented at all:

The ***ExAllocatePool3*** *routine allocates pool memory of the specified type and returns a pointer to the allocated block. This routine is similar to* ***ExAllocatePool2*** *but it adds extended parameters.*

This tells us basically nothing. But the `POOL_EXTENDED_PARAMETER` is found in `Wdm.h` together with the rest of the information we need, so we can get a bit of information from that:

```
typedef enum POOL_EXTENDED_PARAMETER_TYPE {
    PoolExtendedParameterInvalidType = 0,
    PoolExtendedParameterPriority,
    PoolExtendedParameterSecurePool,
    PoolExtendedParameterMax
} POOL_EXTENDED_PARAMETER_TYPE, *PPOOL_EXTENDED_PARAMETER_TYPE;
```

```
#define POOL_EXTENDED_PARAMETER_TYPE_BITS    8
#define POOL_EXTENDED_PARAMETER_REQUIRED_FIELD_BITS    1
#define POOL_EXTENDED_PARAMETER_RESERVED_BITS    (64
- POOL_EXTENDED_PARAMETER_TYPE_BITS - POOL_EXTENDED_PARAMETER_REQUIRED_FIELD_BITS)
```

```
#define SECURE_POOL_FLAGS_NONE       0x0
#define SECURE_POOL_FLAGS_FREEABLE   0x1
#define SECURE_POOL_FLAGS_MODIFIABLE 0x2
```

```
typedef struct _POOL_EXTENDED_PARAMS_SECURE_POOL {
    HANDLE SecurePoolHandle;
    PVOID Buffer;
    ULONG_PTR Cookie;
    ULONG SecurePoolFlags;
} POOL_EXTENDED_PARAMS_SECURE_POOL;
```

```
typedef struct _POOL_EXTENDED_PARAMETER {
    struct {
        ULONG64 Type : POOL_EXTENDED_PARAMETER_TYPE_BITS;
        ULONG64 Optional : POOL_EXTENDED_PARAMETER_REQUIRED_FIELD_BITS;
        ULONG64 Reserved : POOL_EXTENDED_PARAMETER_RESERVED_BITS;
    } DUMMYSTRUCTNAME;
    union {
        ULONG64 Reserved2;
        PVOID Reserved3;
        EX_POOL_PRIORITY Priority;
        POOL_EXTENDED_PARAMS_SECURE_POOL* SecurePoolParams;
    } DUMMYUNIONNAME;
} POOL_EXTENDED_PARAMETER, *PPOOL_EXTENDED_PARAMETER;
```

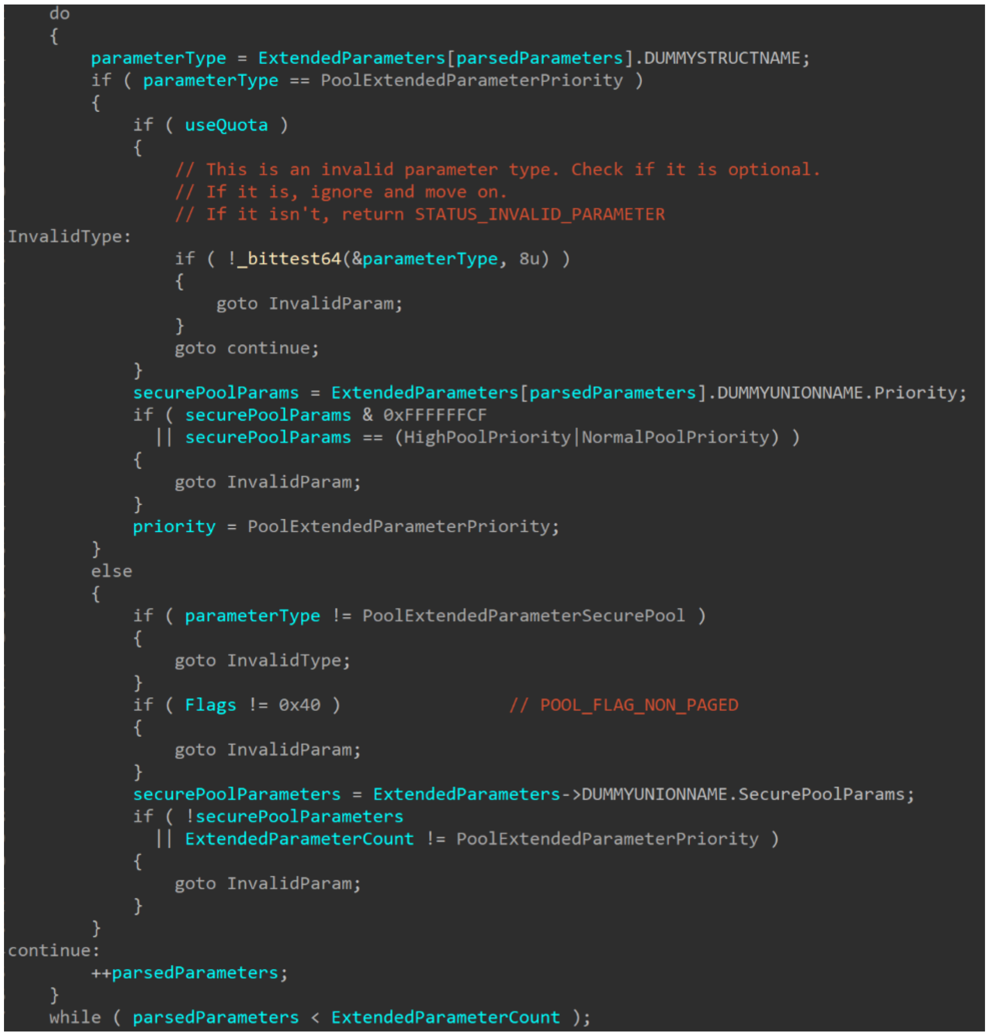
```
typedef CONST POOL_EXTENDED_PARAMETER *PCPOOL_EXTENDED_PARAMETER;
```

First, when we look at the `POOL_EXTENDED_PARAMETER_TYPE` enum, we can see `2` options – `PoolExtendedParametersPriority` and `PoolExtendedParametersSecurePool`. The official documentation has no mention of secure pool anywhere or which parameters it receives and how. By reading it, you'd think **`ExAllocatePool3`** is just **`ExAllocatePool2`** with an additional "priority" parameter.

So back to **`ExAllocatePool3`** – it takes in the same `POOL_FLAGS` parameter, but also two new ones – `ExtendedParameters` and `ExtendedParametersCount`:

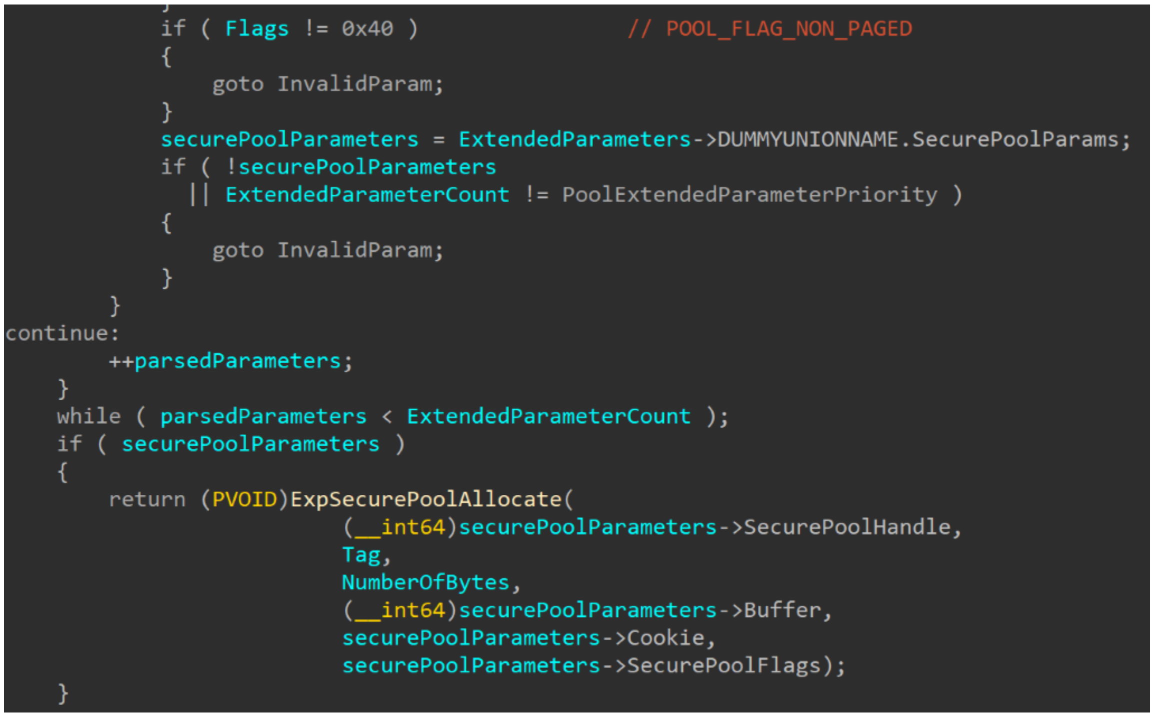
```
DECLSPEC_RESTRICT
PVOID
ExAllocatePool3 (
    _In_ POOL_FLAGS Flags,
    _In_ SIZE_T NumberOfBytes,
    _In_ ULONG Tag,
    _In_ PCPOOL_EXTENDED_PARAMETER ExtendedParameters,
    _In_ ULONG ExtendedParametersCount
);
```

`ExtendedParameters` has a `Type` member, which is one of the values in the `POOL_EXTENDED_PARAMETERS_TYPE` enum. This is the first thing that **`ExAllocatePool3`** looks at:



```
    do
    {
        parameterType = ExtendedParameters[parsedParameters].DUMMYSTRUCTNAME;
        if ( parameterType == PoolExtendedParameterPriority )
        {
            if ( useQuota )
            {
                // This is an invalid parameter type. Check if it is optional.
                // If it is, ignore and move on.
                // If it isn't, return STATUS_INVALID_PARAMETER
InvalidType:
                if ( !_bittest64(&parameterType, 8u) )
                {
                    goto InvalidParam;
                }
                goto continue;
            }
            securePoolParams = ExtendedParameters[parsedParameters].DUMMYUNIONNAME.Priority;
            if ( securePoolParams & 0xFFFFFFCF
              || securePoolParams == (HighPoolPriority|NormalPoolPriority) )
            {
                goto InvalidParam;
            }
            priority = PoolExtendedParameterPriority;
        }
        else
        {
            if ( parameterType != PoolExtendedParameterSecurePool )
            {
                goto InvalidType;
            }
            if ( Flags != 0x40 )                // POOL_FLAG_NON_PAGED
            {
                goto InvalidParam;
            }
            securePoolParameters = ExtendedParameters->DUMMYUNIONNAME.SecurePoolParams;
            if ( !securePoolParameters
              || ExtendedParameterCount != PoolExtendedParameterPriority )
            {
                goto InvalidParam;
            }
        }
continue:
        ++parsedParameters;
    }
    while ( parsedParameters < ExtendedParameterCount );
```

If the parameter type is `1` (`PoolExtendedParameterPriority`), the function reads the Priority field and later calls **`ExAllocatePoolWithTagPriority`** . If the type is `2` (`PoolExtendedParameterSecurePool`) the function reads the `POOL_EXTENDED_PARAMS_SECURE_POOL` structure from `ExtendedParameters` . Later the information from this structure is passed into **`ExpSecurePoolAllocate`** :



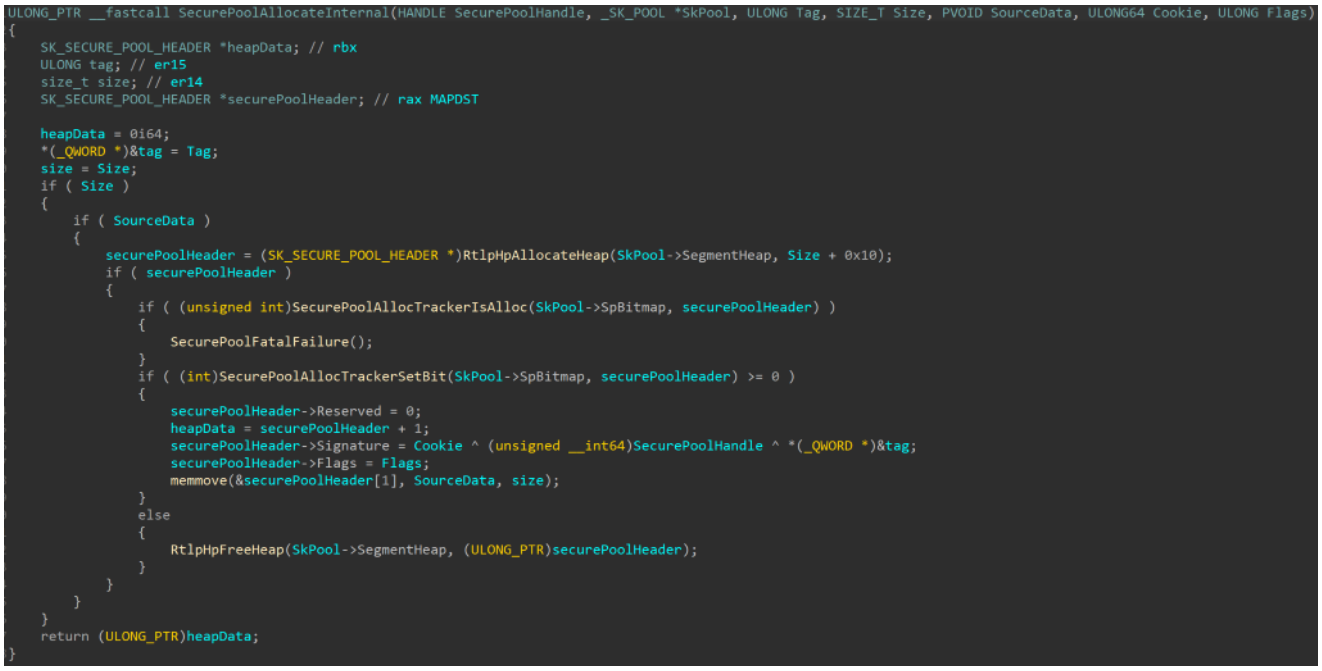
```
            if ( Flags != 0x40 )                // POOL_FLAG_NON_PAGED
            {
                goto InvalidParam;
            }
            securePoolParameters = ExtendedParameters->DUMMYUNIONNAME.SecurePoolParams;
            if ( !securePoolParameters
              || ExtendedParameterCount != PoolExtendedParameterPriority )
            {
                goto InvalidParam;
            }
        }
continue:
        ++parsedParameters;
    }
    while ( parsedParameters < ExtendedParameterCount );
    if ( securePoolParameters )
    {
        return (PVOID)ExpSecurePoolAllocate(
                        (__int64)securePoolParameters->SecurePoolHandle,
                        Tag,
                        NumberOfBytes,
                        (__int64)securePoolParameters->Buffer,
                        securePoolParameters->Cookie,
                        securePoolParameters->SecurePoolFlags);
    }
```

Another interesting thing to notice is that for secure pool allocations, `ExtendedParameterCount` must be one (meaning no other extended parameters are allowed other than the ones related to secure pool) and flags must be `POOL_FLAG_NON_PAGED`. We already know that secure pool only initializes one heap, which is `NonPaged`, so this requirement makes sense.

`ExAllocatePool3` reads from `ExtendedParameters` a handle, buffer, cookie and flags and passes them to `ExpSecurePoolAllocate` together with the tag and number of bytes for this allocation. Let's go over each of these new arguments:

- `SecurePoolHandle` is the handle received from `ExCreatePool`
- `Buffer` is a memory buffer containing the data to be written into this allocation. Since this is a secure pool that is not writable to drivers running in the normal kernel, `SecureKernel` must write the data into the allocation. The flags will determine whether this data can be modified later.
- `Flags` – The options for flags, as we saw in `wdm.h`, are `SECURE_POOL_FLAGS_MODIFIABLE` and `SECURE_POOL_FLAGS_FREEABLE`. As the names suggest, these determine whether the content of the allocation can be updated after it's been created and whether this allocation can be freed.
- `Cookie` is chosen by the caller and will be used to encode the signature in the header of the new entry, together with the tag.

`SkSecurePoolAllocate` forwards the parameters to `SecurePoolAllocate`, which calls `SecurePoolAllocateInternal`. This function calls `RtlpHpAllocateHeap` to allocate heap memory in the secure pool, but adds `0x10` bytes to the size requested by the user:



```
ULONG_PTR __fastcall SecurePoolAllocateInternal(HANDLE SecurePoolHandle, _SK_POOL *SkPool, ULONG Tag, SIZE_T Size, PVOID SourceData, ULONG64 Cookie, ULONG Flags)
{
    SK_SECURE_POOL_HEADER *heapData; // rbx
    ULONG tag; // er15
    size_t size; // er14
    SK_SECURE_POOL_HEADER *securePoolHeader; // rax MAPDST

    heapData = 0i64;
    *(_QWORD *)&tag = Tag;
    size = Size;
    if ( Size )
    {
        if ( SourceData )
        {
            securePoolHeader = (SK_SECURE_POOL_HEADER *)RtlpHpAllocateHeap(SkPool->SegmentHeap, Size + 0x10);
            if ( securePoolHeader )
            {
                if ( (unsigned int)SecurePoolAllocTrackerIsAlloc(SkPool->SpBitmap, securePoolHeader) )
                {
                    SecurePoolFatalFailure();
                }
                if ( (int)SecurePoolAllocTrackerSetBit(SkPool->SpBitmap, securePoolHeader) >= 0 )
                {
                    securePoolHeader->Reserved = 0;
                    heapData = securePoolHeader + 1;
                    securePoolHeader->Signature = Cookie ^ (unsigned __int64)SecurePoolHandle ^ *(_QWORD *)&tag;
                    securePoolHeader->Flags = Flags;
                    memmove(&securePoolHeader[1], SourceData, size);
                }
                else
                {
                    RtlpHpFreeHeap(SkPool->SegmentHeap, (ULONG_PTR)securePoolHeader);
                }
            }
        }
    }
    return (ULONG_PTR)heapData;
}
```

This is done because the first `0x10` bytes of this allocation will be used for a secure pool header:

```
struct _SK_SECURE_POOL_HEADER

    ULONG_PTR Signature;
    ULONG Flags;
    ULONG Reserved;
} SK_SECURE_POOL_HEADER, *PSK_SECURE_POOL_HEADER;
```

This header contains the Flags sent by the caller (specifying whether this allocation can be modified or freed) and a signature made up of the cookie, XORed with the tag and the handle for the pool. This header will be used by `SecureKernel` and is not known to the caller, which will receive a pointer to the data, that is being written immediately after this header (so the user receives a pointer to `<allocation start>+0x10`).

Before initializing the secure pool header, there is a call to **`SecurePoolAllocTrackerIsAlloc`** to validate that the header is inside the secure pool range and not inside an already allocated block. This check doesn't make much sense here, since the header is not a user-supplied address but one that was just allocated by the function itself, but is probably the result of some extra paranoid checks (or an inline macro) that were added as a result of the second design flaw we'll explain shortly.

Then there is a call to `SecurePoolAllocTrackerSetBit` , to set the bit in the bitmap to mark this address as allocated, and only then the header is populated. If the allocation was successful, `SkPool->PoolAllocs` is incremented by `1` .

When this address is eventually returned to **`SkSecurePoolAllocate`** , it is adjusted to a normal kernel address with `SkmiSecurePoolStart` and returned to the normal kernel:

```
poolEntry = SecurePoolAllocate(
                SkCall->SecurePoolHandle,
                SkCall->Tag,
                numberOfBytes,
                secureMdl->MappedSystemVa,
                SkCall->Cookie,
                SkCall->Flags);
SkCall->SizeOnInputAddressOnOutput = (ULONG64)poolEntry;
if ( poolEntry )
{
    SkCall->SizeOnInputAddressOnOutput = (ULONG64)poolEntry
                                          + SkmiSecurePoolStart;
}
```

Then the driver which requested the allocation can use the returned address to read it. But since this pool is protected from being written to by the normal kernel, if the driver wants to make any changes to the content, assuming that it created a modifiable allocation to begin with, it has to use another new API added for this purpose – `ExSecurePoolUpdate`.

Going back to the bitmap — why is it necessary to track the allocation? This takes us to the third and final design flaw, which is that a secure pool header could easily be faked, since the information stored in `Signature` is known — the `Cookie` is caller-supplied, the `Tag` is as well, and the `SecurePoolHandle` too. In fact, in combination with the first flaw this is even worse, as the allocation can then be made to point to a fake `SK_POOL`.

The idea behind this attack would be to first perform a legitimate allocation of, say, 0x40 bytes. Next, manufacture a fake `SK_SECURE_POOL_HEADER` at the beginning of the allocation. Finally, pass the address, plus `0x10` (the size of a header) to the Update or Free functions we'll show next. Now, these functions will use the fake header we've just constructed, which among things can be made to point to a fake `SK_POOL`, on top of causing issues such as pool shape manipulation, double-frees, and more.

By using a bitmap to track legitimate vs. non-legitimate allocations, fake pool headers immediately lead to a crash.

Updating Secure Pool Allocation

When a driver wants to update the contents of an allocation done in the secure pool, it has to call `ExSecurePoolUpdate` with the following arguments:

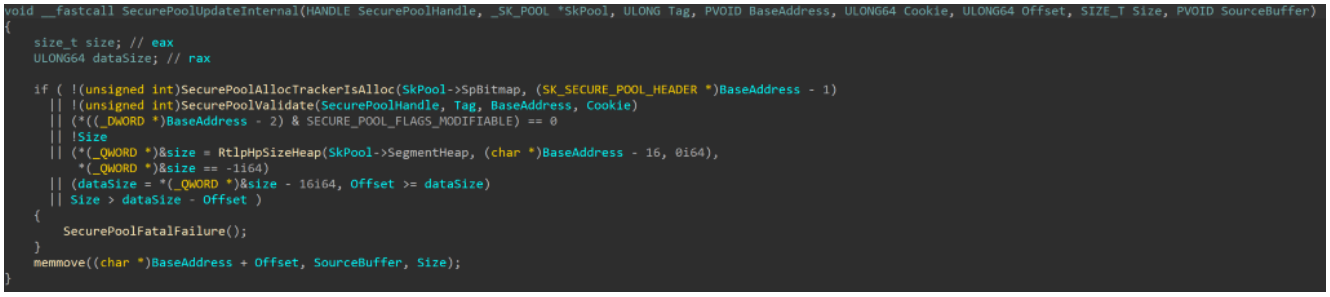
- `SecurePoolHandle` – the driver's handle to the secure pool
- The `Tag` that was used for the allocation that should be modified
- `Address` of the allocation to be modified
- `Cookie` that was used when allocating this memory
- `Offset` inside the allocation

- `Size` of data to be written
- `Pointer` to a buffer containing the new data to write into this allocation

Of course, as you're about to see, the allocation must have been marked as updateable in the first place.

These arguments are sent to secure kernel through a secure call, where they reach **`SkSecurePoolUpdate`**. This function passes the arguments to **`SecurePoolUpdate`**, with the allocation address adjusted to point to the correct secure kernel address.

`SecurePoolUpdate` first validates the pool handle by XORing it with the Signature field of the `SEGMENT_HEAP` and making sure the result is the address of the `SEGMENT_HEAP` itself and then forwards the arguments to **`SecurePoolUpdateInternal`**. First this function calls **`SecurePoolAllocTrackerIsAlloc`** to check the secure pool bitmap and make sure the supplied address is allocated. Then it does some more internal validations of the allocation by calling **`SecurePoolValidate`** – an internal function which validates the input arguments by making sure that the signature field for the allocation matches `Cookie ^ SecurePoolHandle ^ Tag`:



```
void __fastcall SecurePoolUpdateInternal(HANDLE SecurePoolHandle, _SK_POOL *SkPool, ULONG Tag, PVOID BaseAddress, ULONG64 Cookie, ULONG64 Offset, SIZE_T Size, PVOID SourceBuffer)
{
    size_t size; // eax
    ULONG64 dataSize; // rax

    if ( !(unsigned int)SecurePoolAllocTrackerIsAlloc(SkPool->SpBitmap, (SK_SECURE_POOL_HEADER *)BaseAddress - 1)
      || !(unsigned int)SecurePoolValidate(SecurePoolHandle, Tag, BaseAddress, Cookie)
      || (*((_DWORD *)BaseAddress - 2) & SECURE_POOL_FLAGS_MODIFIABLE) == 0
      || !Size
      || (*(_QWORD *)&size = RtlpHpSizeHeap(SkPool->SegmentHeap, (char *)BaseAddress - 16, 0i64),
          *(_QWORD *)&size == -1i64)
      || (dataSize = *(_QWORD *)&size - 16i64, Offset >= dataSize)
      || Size > dataSize - Offset )
    {
        SecurePoolFatalFailure();
    }
    memmove((char *)BaseAddress + Offset, SourceBuffer, Size);
}
```

This check is meant to make sure that the driver that is trying to modify the allocation is the one that made it, since no other driver should have the right cookie and tag that were used when allocating it.

Then **`SecurePoolUpdateInternal`** makes a few more checks:

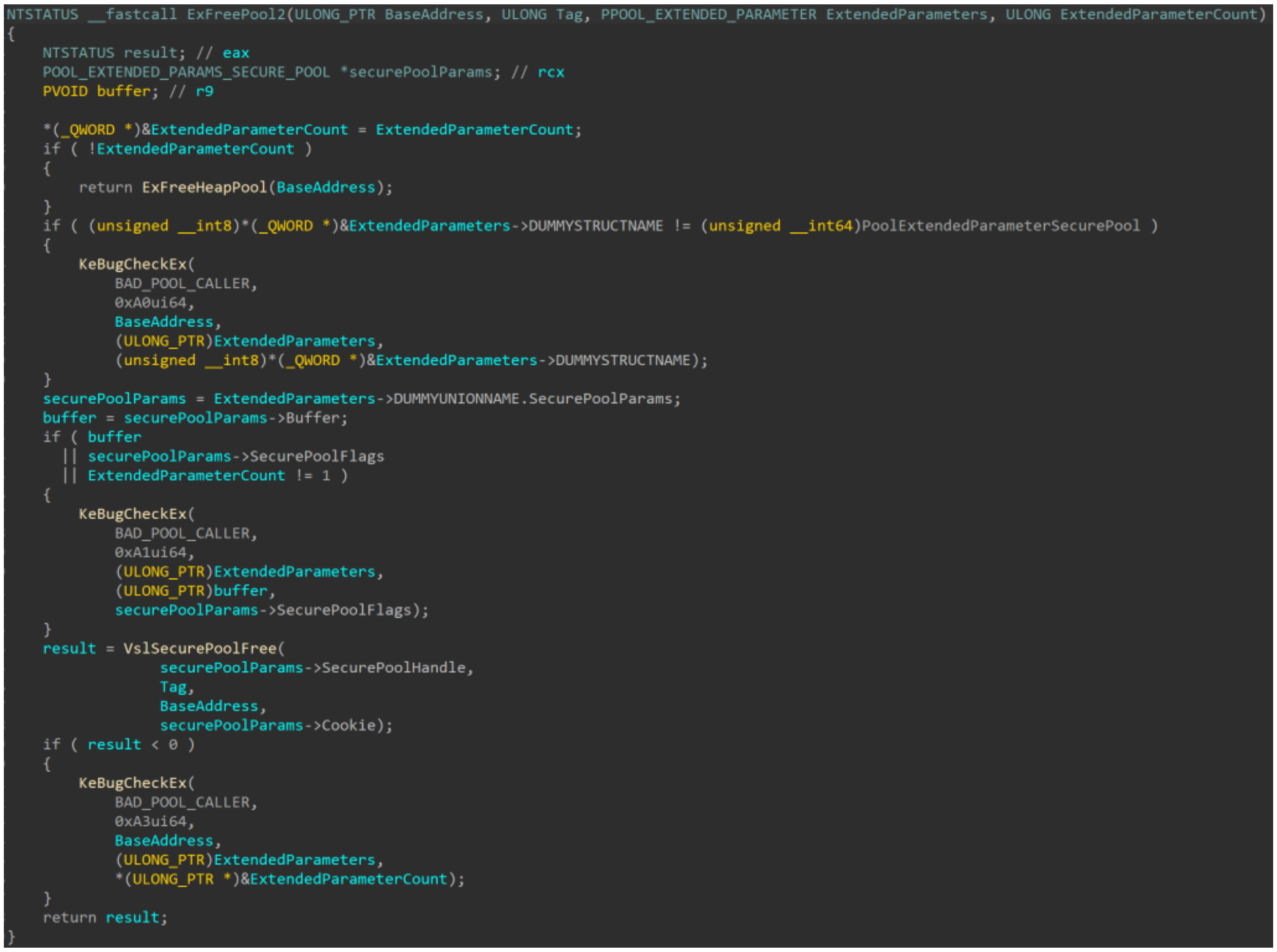
- `Flags` field of the header has to have the `SECURE_POOL_FLAGS_MODIFIABLE` bit set. If this flag was not set when allocating this block, the memory cannot be modified.
- `Size` cannot be zero
- `Offset` cannot be bigger than the size of the allocation
- `Offset` + `Size` cannot be larger than the size of the allocation (since that would create an overflow that would write over the next allocation)

If any of these checks fail, the function would bugcheck with code `0x13A` (`KERNEL_MODE_HEAP_CORRUPTION`).

Only if all the validations pass, the function will write the data in the supplied buffer into the allocation, with the requested offset and size.

Freeing Secure Pool Allocation

The last thing a driver can do with a pool allocation is free it, through `ExFreePool2`. This function, like `ExAllocatePool2/3`
receives `ExtendedParameters` and `ExtendedParametersCount`. If `ExtendedParametersCount` is zero, The function will call `ExFreeHeapPool` to free an allocation done in the normal kernel pool. Otherwise the only valid value for the `ExtendedParameters` Type field is `PoolExtendedParametersSecurePool` (`2`). If the type is correct, the function will read the secure pool parameters and validate that the Flags field is zero and that other fields are not empty. Then the requested address and tag are sent through a secure call, together with the `Cookie` and `SecurePoolHandle` that were read from `ExtendedParameters`:



```
NTSTATUS __fastcall ExFreePool2(ULONG_PTR BaseAddress, ULONG Tag, PPOOL_EXTENDED_PARAMETER ExtendedParameters, ULONG ExtendedParameterCount)
{
    NTSTATUS result; // eax
    POOL_EXTENDED_PARAMS_SECURE_POOL *securePoolParams; // rcx
    PVOID buffer; // r9

    *(_QWORD *)&ExtendedParameterCount = ExtendedParameterCount;
    if ( !ExtendedParameterCount )
    {
        return ExFreeHeapPool(BaseAddress);
    }
    if ( (unsigned __int8)*(_QWORD *)&ExtendedParameters->DUMMYSTRUCTNAME != (unsigned __int64)PoolExtendedParameterSecurePool )
    {
        KeBugCheckEx(
            BAD_POOL_CALLER,
            0xA0ui64,
            BaseAddress,
            (ULONG_PTR)ExtendedParameters,
            (unsigned __int8)*(_QWORD *)&ExtendedParameters->DUMMYSTRUCTNAME);
    }
    securePoolParams = ExtendedParameters->DUMMYUNIONNAME.SecurePoolParams;
    buffer = securePoolParams->Buffer;
    if ( buffer
      || securePoolParams->SecurePoolFlags
      || ExtendedParameterCount != 1 )
    {
        KeBugCheckEx(
            BAD_POOL_CALLER,
            0xA1ui64,
            (ULONG_PTR)ExtendedParameters,
            (ULONG_PTR)buffer,
            securePoolParams->SecurePoolFlags);
    }
    result = VslSecurePoolFree(
                 securePoolParams->SecurePoolHandle,
                 Tag,
                 BaseAddress,
                 securePoolParams->Cookie);
    if ( result < 0 )
    {
        KeBugCheckEx(
            BAD_POOL_CALLER,
            0xA3ui64,
            BaseAddress,
            (ULONG_PTR)ExtendedParameters,
            *(ULONG_PTR *)&ExtendedParameterCount);
    }
    return result;
}
```

The secure kernel functions `SecurePoolFree` and `SecurePoolFreeInternal` validate the supplied address, pool handle and the header of the pool allocation that the caller wants to free, and also make sure it was allocated with the `SECURE_POOL_FLAGS_FREEABLE` flag. If all validations pass, the memory inside the allocation is zeroed and the allocation is freed through `RtlpHpFreeHeap`. Then the `PoolAllocs` field in the `SK_POOL` structure belonging to this handle is decreased and there is another check to see that the value is not below zero.

Code Sample

We wrote a simple example for allocating, modifying and freeing secure pool memory:

```
#include <wdm.h>

DRIVER_INITIALIZE DriverEntry;
DRIVER_UNLOAD DriverUnload;

HANDLE g_SecurePoolHandle;
PVOID g_Allocation;

VOID
DriverUnload (
    _In_ PDRIVER_OBJECT DriverObject
    )
{
    POOL_EXTENDED_PARAMETER extendedParams[1] = { 0 };
    POOL_EXTENDED_PARAMS_SECURE_POOL securePoolParams = { 0 };
    UNREFERENCED_PARAMETER(DriverObject);

    if (g_SecurePoolHandle != nullptr)
    {
        if (g_Allocation != nullptr)
        {
            extendedParams[0].Type = PoolExtendedParameterSecurePool;
            extendedParams[0].SecurePoolParams = &securePoolParams;
            securePoolParams.Cookie = 0x1234;
            securePoolParams.Buffer = nullptr;
            securePoolParams.SecurePoolFlags = 0;
            securePoolParams.SecurePoolHandle = g_SecurePoolHandle;
            ExFreePool2(g_Allocation, 'mySP', extendedParams,
RTL_NUMBER_OF(extendedParams));
        }
        ExDestroyPool(g_SecurePoolHandle);
    }
    return;
}

NTSTATUS
DriverEntry (
    __In__PDRIVER_OBJECT DriverObject,
    __In_ PUNICODE_STRING RegistryPath
    )
{
    NTSTATUS status;
    POOL_EXTENDED_PARAMETER extendedParams[1] = { 0 };
    POOL_EXTENDED_PARAMS_SECURE_POOL securePoolParams = { 0 };
    ULONG64 buffer = 0x41414141;
    ULONG64 updateBuffer = 0x42424242;
     UNREFERENCED_PARAMETER(RegistryPath);

    DriverObject->DriverUnload = DriverUnload;
    //
    // Create a secure pool handle
    //
    status = ExCreatePool( POOL_CREATE_FLG_SECURE_POOL |
                           POOL_CREATE_FLG_USE_GLOBAL_POOL,
                          'mySP',
                           NULL,
                           &g_SecurePoolHandle);
    if (!NT_SUCCESS(status))
    {
        DbgPrintEx(DPFLTR_IHVDRIVER_ID,
                   DPFLTR_ERROR_LEVEL,
                   "Failed creating secure pool with status %lx\n",
                   status);
        goto Exit;
    }
    DbgPrintEx(DPFLTR_IHVDRIVER_ID,
              DPFLTR_ERROR_LEVEL,
               "Pool: 0x%p\n",
               g_SecurePoolHandle);

    //
    // Make an allocation in the secure pool
    //
    extendedParams[0].Type = PoolExtendedParameterSecurePool;
    extendedParams[0].SecurePoolParams = &securePoolParams;
    securePoolParams.Cookie = 0x1234;
    securePoolParams.SecurePoolFlags = SECURE_POOL_FLAGS_FREEABLE |
SECURE_POOL_FLAGS_MODIFIABLE;
    securePoolParams.SecurePoolHandle = g_SecurePoolHandle;
    securePoolParams.Buffer = &buffer;
    g_Allocation = ExAllocatePool3(POOL_FLAG_NON_PAGED,
                                    sizeof(buffer),
                                   'mySP',
                                   extendedParams,
                                   RTL_NUMBER_OF(extendedParams));
    if (g_Allocation == nullptr)
    {
        DbgPrintEx(DPFLTR_IHVDRIVER_ID,
DPFLTR_ERROR_LEVEL.
                   "Failed allocating memory in secure pool\n");
        status = STATUS_UNSUCCESSFUL;
        goto Exit;
    }

    DbgPrintEx(DPFLTR_IHVDRIVER_ID,
                DPFLTR_ERROR_LEVEL,
               "Allocated: 0x%p\n",
               g_Allocation);
    //
    // Update the allocation
    //
    status = ExSecurePoolUpdate(g_SecurePoolHandle,
                                'mySP',
                                g_Allocation,
                                securePoolParams.Cookie,
                                0,
                                sizeof(updateBuffer),
                                &updateBuffer);
    if (!NT_SUCCESS(status))
    {
        DbgPrintEx(DPFLTR_IHVDRIVER_ID,
                  DPFLTR_ERROR_LEVEL,
                   "Failed updating allocation with status %lx\n",
                   status);
        goto Exit;
    }

    DbgPrintEx(DPFLTR_IHVDRIVER_ID,
               DPFLTR_ERROR_LEVEL,
               "Successfully updated allocation\n");

    status = STATUS_SUCCESS;

Exit:
    return status;
}
```

Conclusion

The secure pool can be a powerful feature to help drivers protect sensitive information from other code running in kernel mode. It allows us to store memory in a way that can't be modified, and possibly not even freed, by anyone, including the driver that allocated the memory! It has the new benefit of allowing any kernel code to make use of some of the benefits of VTL `1` protection, not limiting them to Windows code only.

Like any new feature, this implementation is not perfect and might still have issues, but this is definitely a new and exciting addition that is worth keeping an eye on in upcoming Windows releases.

Read our other blog posts: