Scoop the Windows 10 Pool!

05 Juin 2020

Paul Fariello (@paulfariello)
Corentin Bayet (@OnlyTheDuck)

Who are we?

- Corentin ”@OnlyTheDuck” Bayet
  - Previous work on Windows kernel heap exploitation.
- Paul Fariello ”@paulfariello”
  - Previous work on VM escape and exploiting Linux stuff.
- Both employees @Synacktiv
  - Offensive security company created in 2012.
  - Soon 74 ninjas!
  - pentest, reverse engineering, development.
  - Paris, Toulouse, Lyon, Rennes

Windows Pool

- Windows Pool is the Windows Kernel Heap allocator
- Used since Windows 7
- Segment Heap allocator introduced in Windows 10 kernel - 19H1

Goals of the research

- Discover what changed
- What is the impact on specific pool materials?
- What is the impact on an exploitation point of view?

Plan

1 Windows Pool 101

2 Exploiting a Heap Overflow

3 Exploitation

4 Conclusion

Plan

1 Windows Pool 101

2 Exploiting a Heap Overflow

3 Exploitation

4 Conclusion

Pool Allocator - API

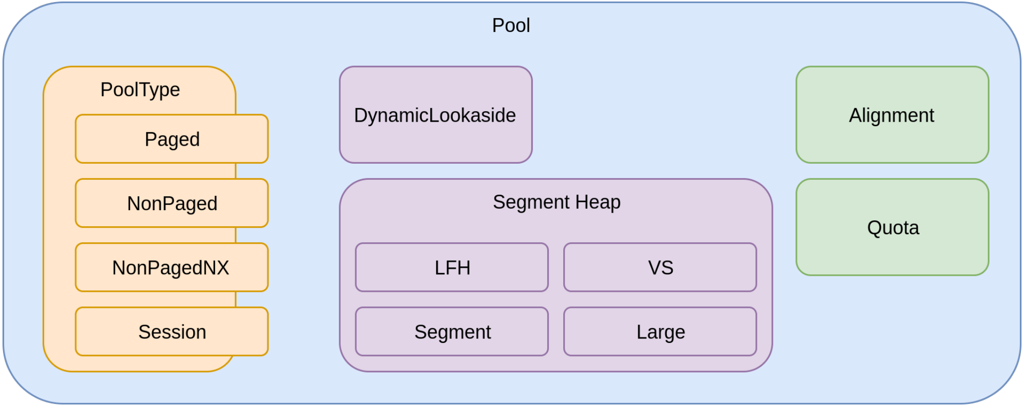
```
void * ExAllocatePoolWithTag(POOL_TYPE       PoolType,
                             size_t          NumberOfBytes,
                             unsigned int    Tag);

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

Pool Allocator

- Allocation associated with a tag
  - 32-bit value, usually printable
  - Mostly used for debug
- Allocation of different memory types
  - NonPagedPool (NonPagedPoolNx since Windows 8)
  - PagedPool
  - SessionPool
- Additionnal features
  - Quota
  - Alignment

Pool Allocator



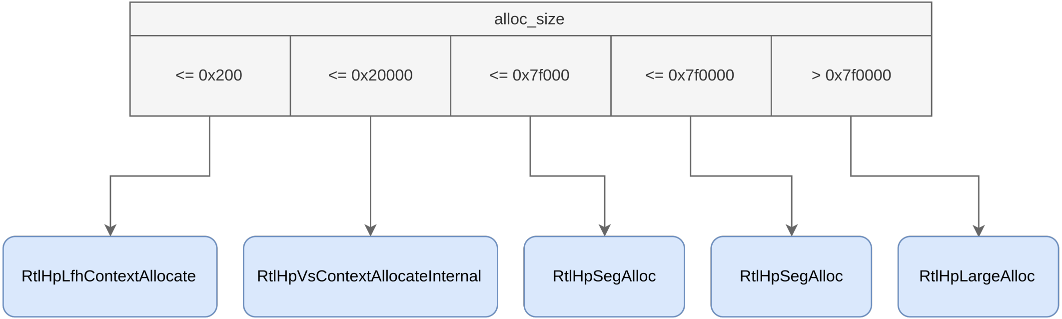
Segment Heap

- Introduced in userland with Windows 10
- Used in kernel since Windows 10 - 19H1
- Aims at providing different features depending on allocation size

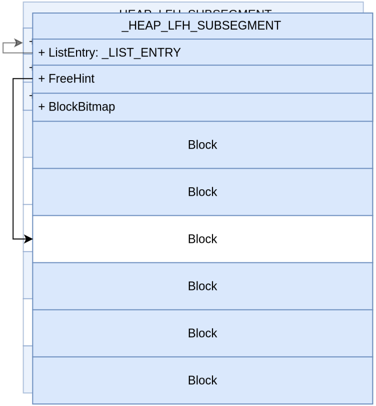
Segment Heap – Backends

- Allocation delegated to different backend
- Depends on requested size
- Each backend has its own allocation/free mechanism
  - Low Fragmentation Heap
  - Variable Size
  - Segment
  - Large

Segment Heap – Backends



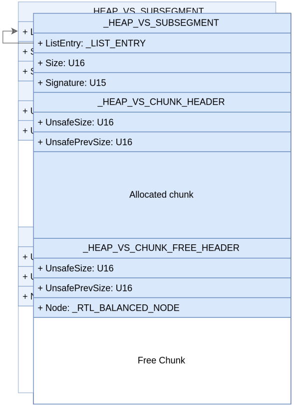
Segment Heap – LFH



LFH

- allocation <= 512 B
- backed by multiple `SubSegments`
- chunk grouped by size in `buckets`
- affinity slots if contention detected
- bitmap of free/used blocks
- (kind of) randomize access

Segment Heap – VS



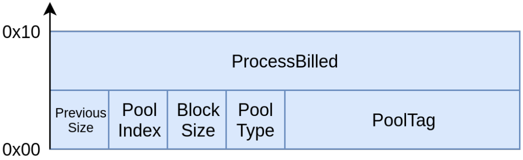
VS

- 512 B < allocation <= 128 KiB
- backed by multiple `SubSegment`
- RB tree maintaining list of free chunks
- Chunk are prefixed with a dedicated struct `_HEAP_VS_CHUNK_HEADER`
- Contiguous free chunks are coalesced

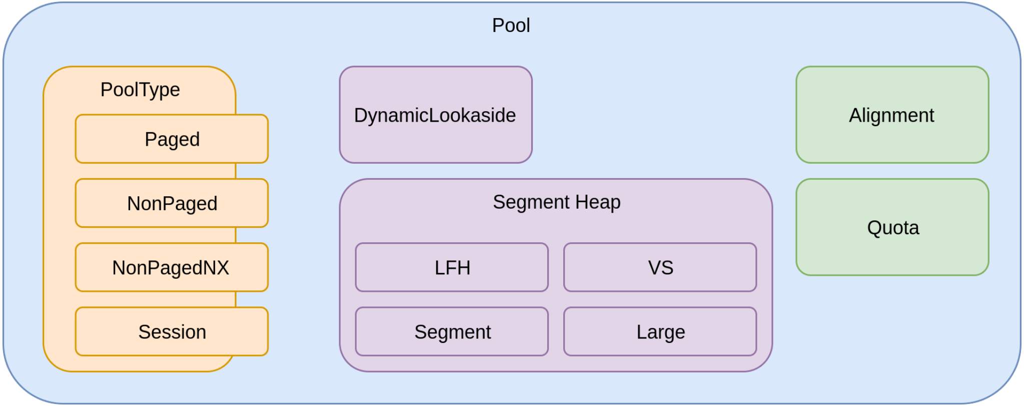
Pool Allocator - POOL_HEADER

- Present before each allocated chunk
- Was used by the previous allocator
- Not needed by the Segment Heap, but still present

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



Pool Allocator



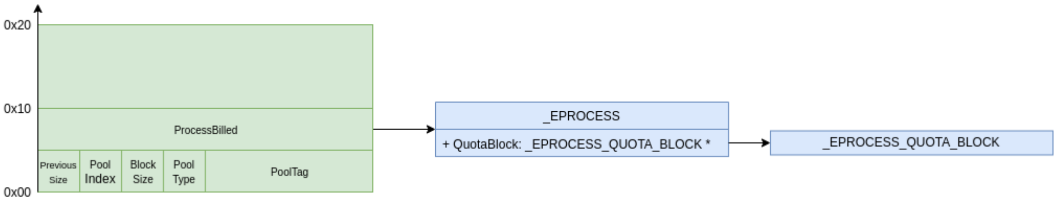
DynamicLookaside

- 512 B < allocation <= 4064 B
- Dedicated linked list of free chunk
- Prevent usage of backend's free mechanism
- Grouped by size
- Size recovered from `POOL_HEADER`
- Enabled only if size is heavily used (Balance Set Manager)

Pool Allocator - PoolQuota

- Mechanism to monitor heap usage
- Enabled with `PoolQuota` bit in PoolType (bit 3)
- Pointer to `EPROCESS` stored in `ProcessBilled` of `POOL_HEADER`
  - A counter is incremented when an allocation occurs
  - ... and decremented when a free occurs

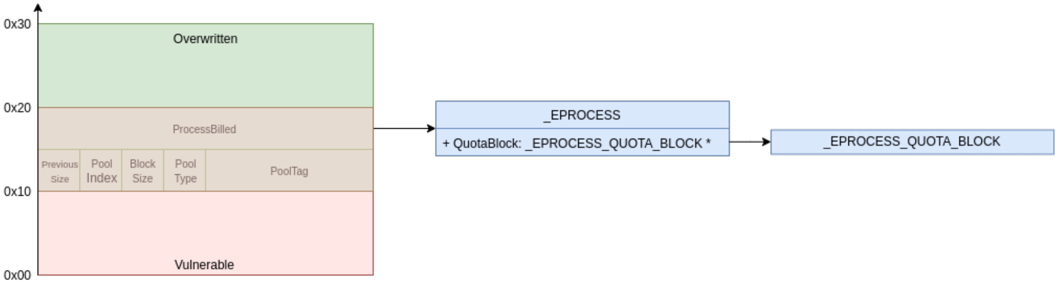
Pool Allocator - PoolQuota



Quota Process Pointer Overwrite attack

- `Quota Process Pointer Overwrite` is an attack leveraging a heap overflow
- Described by @kernelpool in 2011
- Overwrite the `POOL_HEADER` of the next allocation
  - Make `ProcessBilled` point to a fake `EPROCESS`
  - Provides arbitrary decrement primitive
  - Might be enough to elevate privileges to `SYSTEM`

Quota Process Pointer Overwrite attack



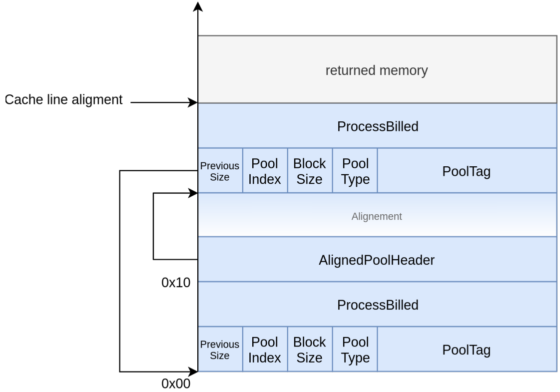
Quota Process Pointer Overwrite Mitigation

- Mitigated since Windows 8
- ProcessBilled pointer xored with a randomly generated Cookie
- $\texttt{ProcessBilled} = \texttt{addrof(EPROCESS)} \oplus \texttt{addrof(Chunk)} \oplus \texttt{ExpPoolQuotaCookie}$
- Cannot be forged without a strong leak / read primitive
- Previous work on this at Nuit du Hack XV.

Alignment mechanism

- Request memory aligned on CPU cache line
  - Set `CacheAligned` bit in `POOL_TYPE` (bit 2)
- But chunk must respect two conditions:
  - `POOL_HEADER` present at the very start of the chunk
  - `POOL_HEADER` present 0x10 bytes before the returned pointer
- Might endup with two `POOL_HEADER` in the chunk
- `PreviousSize` of second `POOL_HEADER` = offset to first `POOL_HEADER`

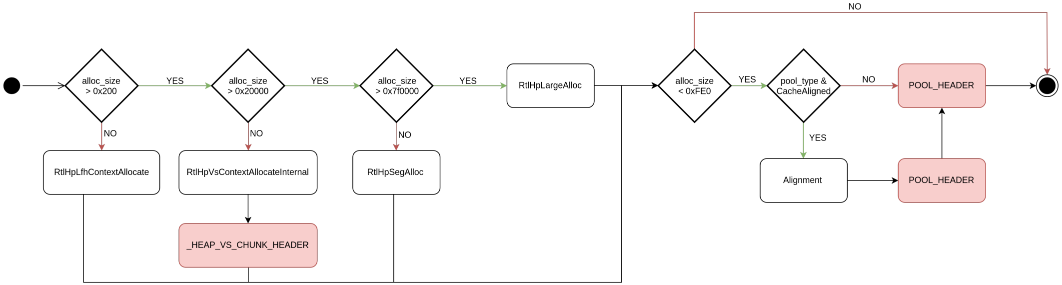
Alignment mechanism



Returned memory

- A chunk can be shaped with various headers
- Depends on
  - used backend
  - requested `POOL_TYPE`

Returned memory



Plan

1 Windows Pool 101

2 Exploiting a Heap Overflow

3 Exploitation

4 Conclusion

Exploiting a Pool Overflow after Windows 10 19H1

- When having a big and controlled heap overflow primitive, probably better to do a full data attack
  - Overwrite the `POOL_HEADER` with values that won't make crash
  - Ensure `PoolQuota` bit is not set in `PoolType`
  - Target next chunk content
  - Fix VS header
- But overflow of 4 bytes on `POOL_HEADER` of the next chunk is enough
  - Aligned Chunk Confusion

Aligned Chunk Freeing Mechanism

- When freeing an aligned chunk, the allocator needs to find the real address of the start of the chunk.
- Uses the `PreviousSize` field of the second `POOL_HEADER` to retrieve the start of the chunk

```
OriginalHdr = AlignedHdr - (AlignedHdr->PreviousSize * 0x10)
```

- The values stored in the `OriginalHeader` are then used to free the chunk

Aligned Chunk Freeing Mechanism

- Mechanism exists since introduction of Pool allocator
- But before introduction of Segment Heap, there were multiple checks when freeing an aligned chunk :
  - The offset between the two headers were recomputed and checked
  - The BlockSize of the second header was recomputed and checked
  - The AlignedPoolHeader pointer was checked to match the address of the aligned header

Aligned Chunk Freeing Mechanism

```
if ( pool_type & NonPagedPoolCacheAligned ) // // is chunk cache aligned
{
  previous_block_size = *(_WORD *)&chunk_addr->previous_size;
  v66 = 0x10i64 * (unsigned __int8)*(_WORD *)&chunk_addr->previous_size;
  corrected_chunk_addr = &chunk_addr[v66 / 0xFFFFFFFFFFFFFFF0ui64];
  if ( !(chunk_addr[v66 / 0xFFFFFFFFFFFFFFF0ui64].pool_type & NonPagedPoolMustSucceed) )
    KeBugCheckEx(
      0xC2u,
      0xBui64,
      (ULONG_PTR)&chunk_addr[v66 / 0xFFFFFFFFFFFFFFF0ui64],
      *(unsigned int *)&corrected_chunk_addr->previous_size,
      (ULONG_PTR)P);
  v68 = (ExpCacheLineSize - 1) & (0xFFFFFFF0 - (_DWORD)corrected_chunk_addr);
  if ( !v68
    || (MY_POOL_HEADER *)((char *)corrected_chunk_addr + v68) != chunk_addr
    || (LODWORD(v7) = (unsigned __int8)*(_WORD *)&corrected_chunk_addr->block_size,
        v69 = (unsigned __int8)*(_WORD *)&chunk_addr->block_size + (unsigned __int8)previous_block_size,
        v112 = v7,
        (_DWORD)v7 != v69) )
  {
    KeBugCheckEx(
      0xC2u,
      0x10ui64,
      (ULONG_PTR)corrected_chunk_addr,
      *(unsigned int *)&corrected_chunk_addr->previous_size,
      (ULONG_PTR)corrected_chunk_addr + v68);
  }
  if ( (unsigned __int8)previous_block_size > 1u
    && ((unsigned __int64)chunk_addr ^ ExpPoolQuotaCookie) != *(_QWORD *)&corrected_chunk_addr[1].previous_size )
  {
    KeBugCheckEx(
      0xC2u,
      0x11ui64,
      (ULONG_PTR)corrected_chunk_addr,
      *(unsigned int *)&corrected_chunk_addr->previous_size,
      (unsigned __int64)chunk_addr ^ ExpPoolQuotaCookie);
  }
  chunk_addr = (MY_POOL_HEADER *)((char *)chunk_addr - v66);
  P = &corrected_chunk_addr[1];
}
```

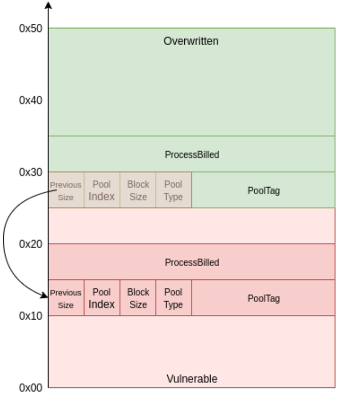
Aligned Chunk Freeing Mechanism

- Since Segment Heap introduction, all checks are gone

```
if ( *(_BYTE *)(user_addr - 0xD) & NonPagedPoolCacheAligned )// is chunk cache aligned
{
  chunk_addr -= (unsigned __int8)*(_WORD *)&chunk_addr->previous_size;
  chunk_addr->pool_type |= NonPagedPoolCacheAligned;
}
```

Aligned Chunk Confusion

- Overwrite PreviousSize and PoolType of next chunk to change it into a CacheAligned chunk
- Trigger free of overwritten chunk, but actually frees controlled `POOL_HEADER`
- Leverage DynamicLookaside to reuse the created chunk



Plan

1 Windows Pool 101

2 Exploiting a Heap Overflow

3 Exploitation

4 Conclusion

Notice

Goals

- Demonstrate exploitation technique
- Not vulnerability

Setup

- Demo driver with dedicated fake vulnerability

Aligned Chunk Confusion Exploitation

Goals

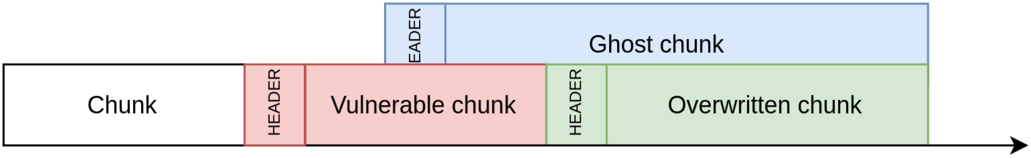
- Leverage Aligned Chunk Confusion to elevate privilege
- Develop a generic exploitation technique
  - That can work in PagedPool or NonPagedPoolNx
  - That is independent of the size of the vulnerable chunk

Overflow primitive constraints

- Overflow 1st and 4th byte of following `POOL_HEADER`
- Control allocation and free of vulnerable chunk

Exploitation strategy

1. Leverage Aligned Chunk Confusion
2. Create a ghost chunk
3. Allocate an object in the ghost chunk
4. Overwrite this object to obtain read/write primitives



Finding an object – Requirements

Need objects that can be sprayed and that are interesting to control.

Object properties

- Controlled allocation and free, to be sprayable
- Provides arbitrary read or write if fully user controlled
- Variable size, to be generic to any heap overflow

Object residence

- One in PagedPool
- One in NonPagedPoolNx

Targeted object – PagedPool

PipeAttribute

- Linked to a `Pipe`
- User controlled insertion and deletion
- Controlled size
- Provide arbitrary read
- Easy way to write data in kernel

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

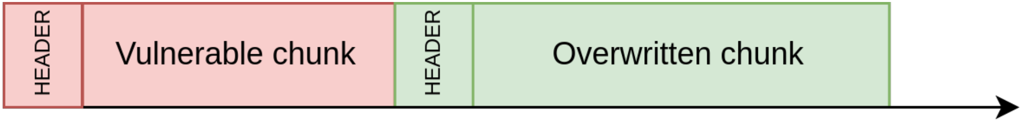
Exploitation strategy - updated

1. Overwrite next `POOL_HEADER`
2. Create a ghost chunk
3. Use `PipeAttribute` to get a leak and an arbitrary read
4. Use `Quota Process Pointer Overwrite` to get `SYSTEM` privileges

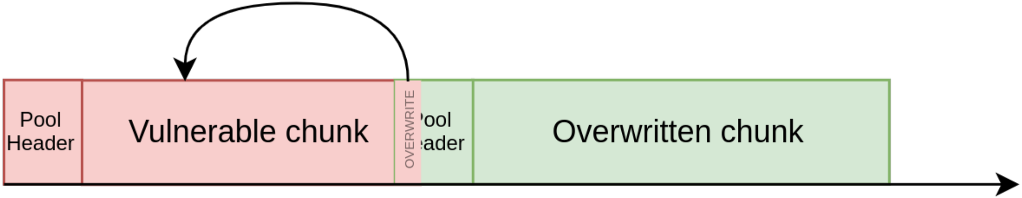
Note

Following example is only about PagedPool. But the same applies to NonPagedPoolNx with a different object.

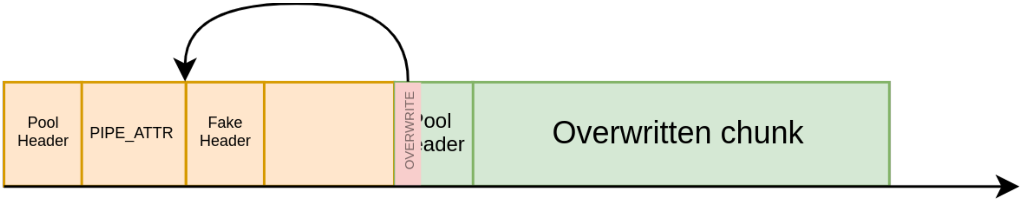
Shaping



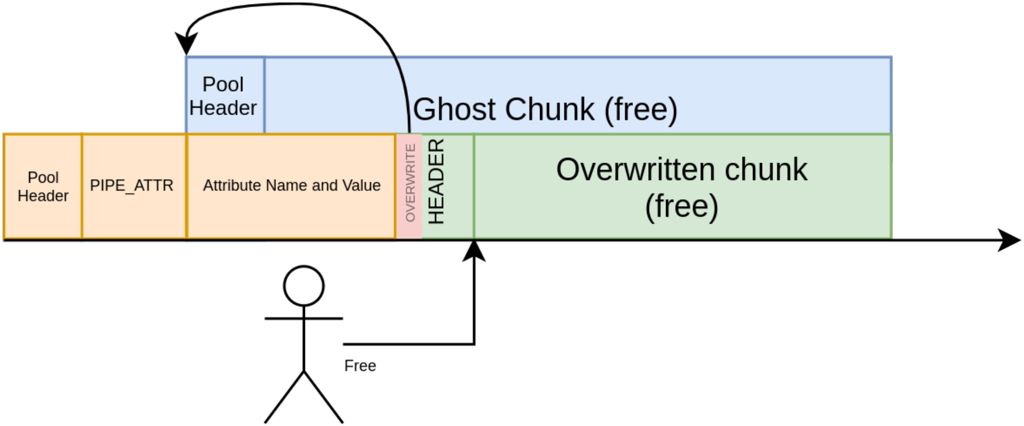
Creating the ghost chunk



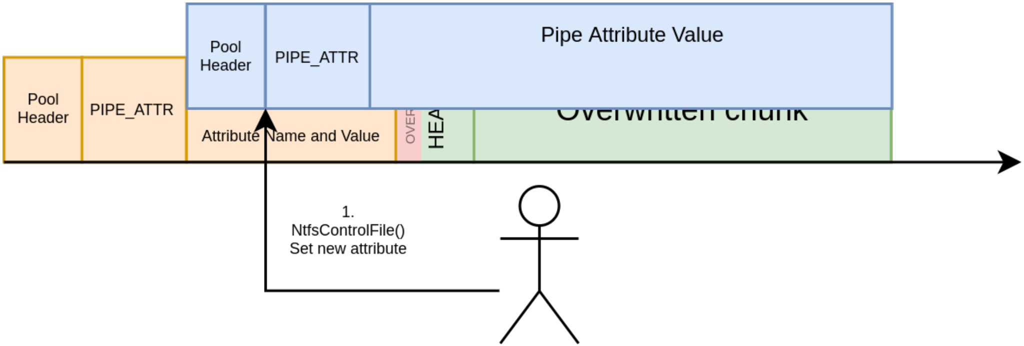
Creating the ghost chunk



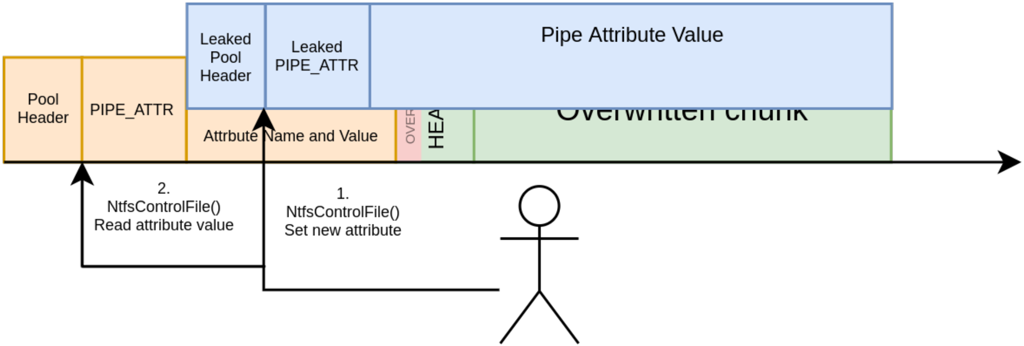
Creating the ghost chunk



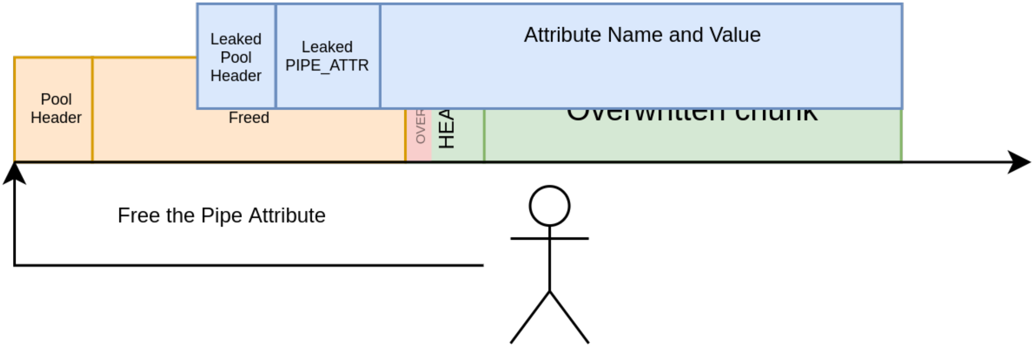
Getting a leak



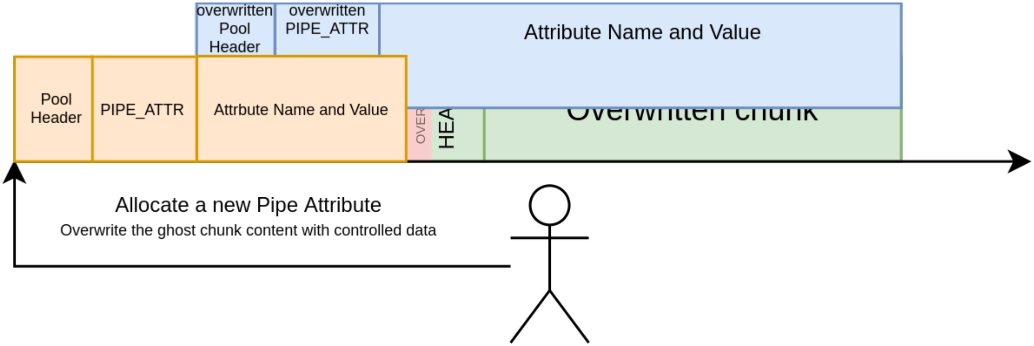
Getting a leak



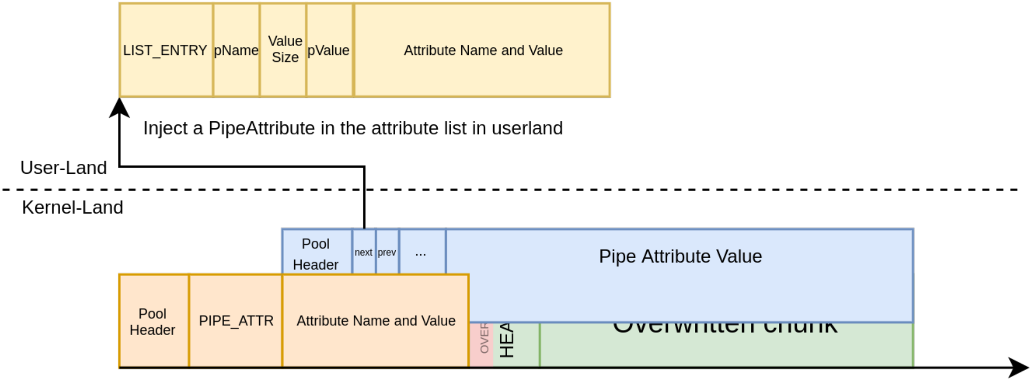
Getting an arbitrary read



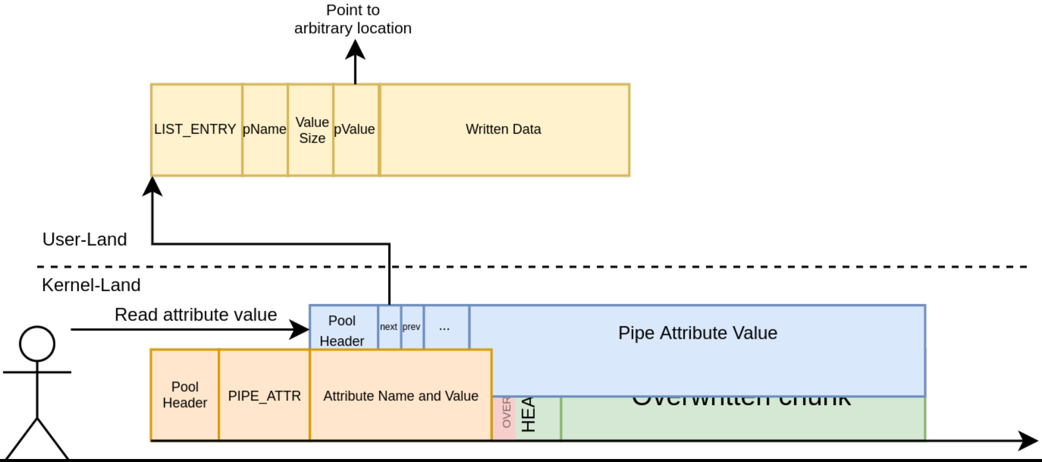
Getting an arbitrary read



Getting an arbitrary read



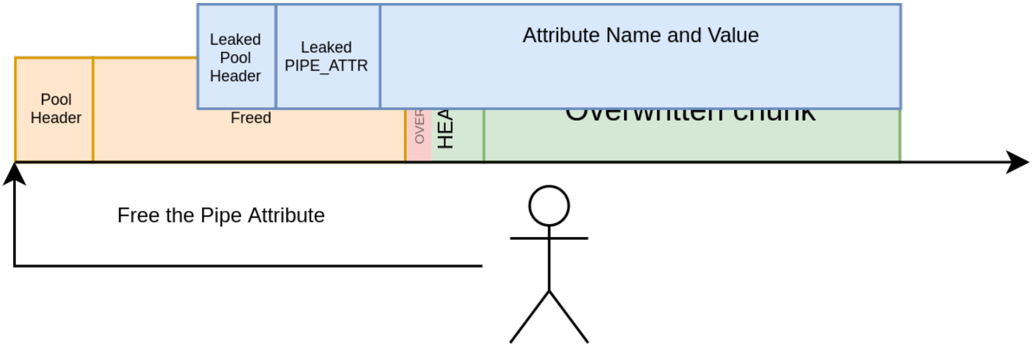
Getting an arbitrary read



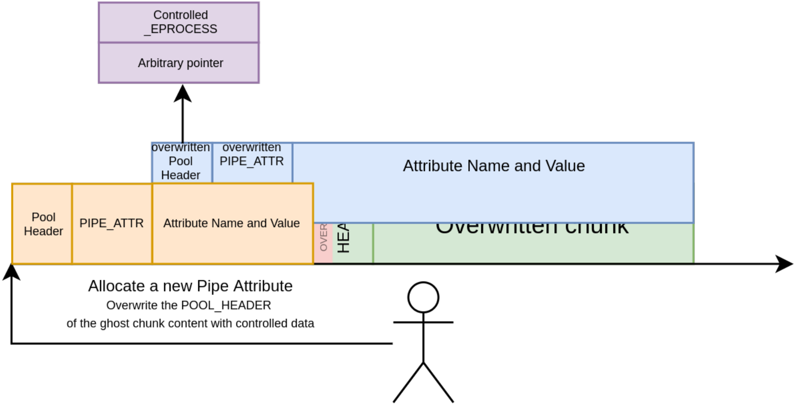
Exploitation - Arbitrary read

- We got an arbitrary read and a heap leak
- We can use it this to retrieve some values
  - Value of `ExpPoolQuotaCookie`
  - Address of self `EPROCESS`
  - Address of self `TOKEN`
- And use a `Quota Process Pointer Overwrite` to get an arbitrary decrement !

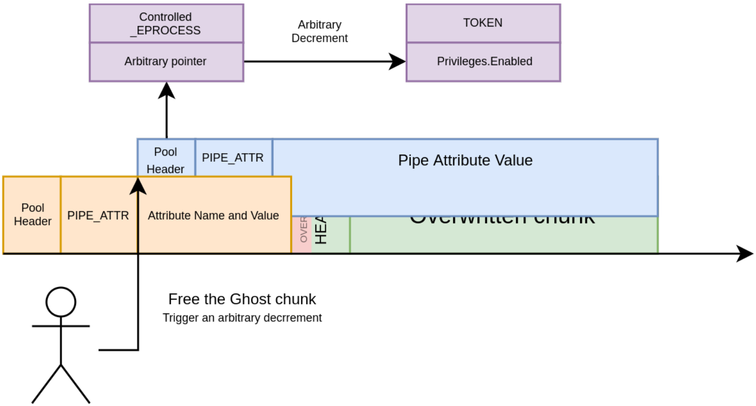
Getting an arbitrary decrement



Getting an arbitrary decrement



Getting an arbitrary decrement



Exploitation - Use the arbitrary decrement

- Use the arbitrary decrement twice by reallocating an refreeing the ghost chunk
  - Decrement `TOKEN.Prileges.Enabled`
  - Decrement `TOKEN.Prileges.Present`
- Provides `SeDebugPrivilege` to our process
- Debug a `SYSTEM` process and inject a shellcode

DEMO

Exploitation - Discussion

- Could use the same exploitation technique to achieve different outcomes (code execution, etc.)
- Not perfectly stable, spraying could be improved
- Tested on one version of Windows 10 only
- Offsets of ntoskrnl hardcoded, that can be easily fixed using the arbitrary read

https://github.com/synacktiv/Windows-kernel-SegmentHeap-Aligned-Chunk-Confusion

Plan

1 Windows Pool 101

2 Exploiting a Heap Overflow

3 Exploitation

4 Conclusion

Conclusion

- Segment Heap brings lots of changes to the Pool
- Some mitigations have been removed allowing for a novel exploitation technique
- Our exploitation technique works with any heap overflow providing:
  - overwrite first and fourth bytes of `POOL_HEADER`
  - control allocation and deallocation of the vulnerable chunk
- The exploit we developed is generic:
  - Works in both PagedPool and NonPagedPoolNx
  - Works for any vulnerable size

AVEZ-VOUS DES QUESTIONS?

MERCI DE VOTRE ATTENTION

SYNACKTIV
DIGITAL SECURITY