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

```c
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

- Additional features
  - Quota
  - Alignment
Pool Allocator

Pool

PoolType
- Paged
- NonPaged
- NonPagedNX
- Session

DynamicLookaside

Segment Heap
- LFH
- VS
- Segment
- Large

Alignment

Quota
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

<table>
<thead>
<tr>
<th>alloc_size</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;= 0x200</td>
</tr>
<tr>
<td>&lt;= 0x20000</td>
</tr>
<tr>
<td>&lt;= 0x7f000</td>
</tr>
<tr>
<td>&lt;= 0x7f0000</td>
</tr>
<tr>
<td>&gt; 0x7f0000</td>
</tr>
</tbody>
</table>

- RtlHpLfhContextAllocate
- RtlHpVsContextAllocateInternal
- RtlHpSegAlloc
- RtlHpSegAlloc
- RtlHpLargeAlloc
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

- 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

```c
struct POOL_HEADER {
    char PreviousSize;
    char PoolIndex;
    char BlockSize;
    char PoolType;
    int PoolTag;
    Ptr64 ProcessBilled;
};
```
Pool Allocator
DynamicLookaside

- $512 \text{ B} < \text{allocation} \leq 4064 \text{ B}$
- Dedicated linked list of free chunk
- Prevent usage of backend’s free mechanism
- Grouped by size
- Size recovered from \texttt{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
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
  \[ \text{ProcessBilled} = \text{addr}(\text{EPROCESS}) \oplus \text{addr}(\text{Chunk}) \oplus \text{ExpPoolQuotaCookie} \]
- 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 end up 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.

\[ \text{OriginalHdr} = \text{AlignedHdr} - (\text{AlignedHdr->PreviousSize} \times 0x10) \]

- 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;
    v60 = 0x10164 * (unsigned __int8*)*(WORD*)chunk_addr->previous_size;
    corrected_chunk_addr = chunk_addr + v60 / 0xFFFFF6F00164;
    if (!(*(chunk_addr[v60 / 0xFFFFF6F00164]).pool_type & NonPagedPoolMustSucceed))
        _XeBugCheckEx(
            0xC2u,
            0x58164,
            (ULONG_PTR)&chunk_addr[v60 / 0xFFFFF6F00164],
            *(unsigned int *)&corrected_chunk_addr->previous_size,
            (ULONG_PTR)&corrected_chunk_addr);
    v60 = (ExpCacheLineSize - 1) & (0xFFFFF00 - (DWORD)corrected_chunk_addr);
    if ( !v60
    {(_POOL_HEADER *)&(char *)corrected_chunk_addr + v60) != chunk_addr
    { (LONGLONG)v0 = (unsigned __int8*)(WORD*)correction_chunck_addr->block_size,
        v60 = (unsigned __int8)*(_WORD*)chunk_addr->block_size + (unsigned __int8)previous_block_size,
        v112 = 97,
        *(DWORD)v0 != v60 } }
        _XeBugCheckEx(
            0xC2u,
            0x10164,
            (ULONG_PTR)corrected_chunk_addr,
            *(unsigned int *)&corrected_chunk_addr->previous_size,
            (ULONG_PTR)corrected_chunk_addr + v60);
    }
    if ( (unsigned __int8)previous_block_size > 1
    {(_unsigned __int64)chunk_addr = _ExpPoolQuotaCookie) != *(DWORD *)&corrected_chunk_addr[1].previous_size }
        _XeBugCheckEx(
            0xC2u,
            0x10164,
            (ULONG_PTR)corrected_chunk_addr,
            *(unsigned int *)&corrected_chunk_addr->previous_size,
            *(unsigned __int64)chunk_addr = _ExpPoolQuotaCookie);
    }
    chunk_addr = (_POOL_HEADER *)&(char *)chunk_addr - v60;
    P = &corrected_chunk_addr[1];
}
Aligned Chunk Freeing Mechanism

Since Segment Heap introduction, all checks are gone

```c
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

```c
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.
Creating the ghost chunk
Creating the ghost chunk

Pool Header  PIPE_ATTR  Fake Header  OVERWRITE  Overwritten chunk
Creating the ghost chunk

Pool Header | PIPEATTR | Attribute Name and Value | OVERWRITE HEADER | Overwritten chunk (free)
Getting a leak

1. NtfsControlFile()
   Set new attribute
Getting a leak

1. `NtfsControlFile()` Set new attribute
2. `NtfsControlFile()` Read attribute value

Pipe Attribute Value

Overwritten chunk

Attribute Name and Value

Leaked Pool Header

Leaked PIPE_ATTR

Pool Header

PIPE_ATTR
Getting an arbitrary read
Getting an arbitrary read

Allocate a new Pipe Attribute
Overwrite the ghost chunk content with controlled data
Getting an arbitrary read

Inject a PipeAttribute in the attribute list in userland

Kernel-Land

User-Land

Pool Header | PIPE_ATTR | Attribute Name and Value

Pipe Attribute Value

Overwritten chunk
Getting an arbitrary read

Point to arbitrary location

List Entry: pName, Value Size, pValue, Written Data

User-Land

Kernel-Land

Read attribute value

Pool Header, PIPE_ATTR, Attribute Name and Value

Pipe Attribute Value

Overwritten chunk
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

- Pool Header
  - Freed
  - Overwritten chunk
- Leaked Pool Header
- Leaked PIPE_ATTR
- Attribute Name and Value

Free the Pipe Attribute
Getting an arbitrary decrement

Allocate a new Pipe Attribute
Overwrite the POOL_HEADER of the ghost chunk content with controlled data
Getting an arbitrary decrement

Controlled _EPROCESS
Arbitrary pointer

Arbitrary Decrement

TOKEN
Privileges.Enabled

Pool Header
PIPE_ATTR
Attribute Name and Value

Free the Ghost chunk
Trigger an arbitrary decrement

Pipe Attribute Value
Overwritten chunk

Trigger 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
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

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
MERCI DE VOTRE ATTENTION

AVEZ-VOUS DES QUESTIONS?