# DATA ONLY EXPLOITS FOR WINDOWS KERNEL BUGS

Arav Garg
@AravGarg3

# ABOUT THIS RESEARCH

Over the past 7 months, I have written reliable PE exploits for 7 CVEs in Windows kernel drivers. These are:

-> 5 in clfs.sys: CVE-2022-22000, CVE-2021-40443, CVE-2021-36955, CVE-2021-40466, CVE-2021-31954

-> 2 in ntfs.sys: CVE-2021-43229, CVE-2021-31956

# BUG CLASSES

-> Paged Pool Overflow
-> Integer Overflow
-> Use-after-free

# GOOD TO KNOW

SEGMENT HEAP IN THE WINDOWS KERNEL AFTER 19H1 UPDATE

http://web.archive.org/web/20211113145025/https://www.sstic.org/media/SSTIC2020/SSTIC-actes/pool_overflow_exploitation_since_windows_10_19h1/SSTIC2020-Article-pool_overflow_exploitation_since_windows_10_19h1-bayet_fariello.pdf

# TARGETED OBJECTS

-> WINDOWS NOTIFICATION FACILITY (WNF)
-> PIPE ATTRIBUTES

# WINDOWS NOTIFICATION FACILITY (WNF)

- http://web.archive.org/web/20210822230409/https://docplayer.net/145030841-The-windows-notification-facility.html
- http://web.archive.org/web/20210822230419/https://blog.quarkslab.com/playing-with-the-windows-notification-facility-wnf.html
- http://web.archive.org/web/20211128145401/https://research.nccgroup.com/2021/07/15/cve-2021-31956-exploiting-the-windows-kernel-ntfs-with-wnf-part-1/
- http://web.archive.org/web/20220114081845/https://research.nccgroup.com/2021/08/17/cve-2021-31956-exploiting-the-windows-kernel-ntfs-with-wnf-part-2/

Objects from WNF are used to groom the heap and get PE

# CREATE WNF STATE NAME

-> WNF State Name is created via the syscall - NtCreateWnfStateName
-> Kernel internally calls ExpWnfCreateNameInstance()

```
ExpWnfCreateNameInstance
          (_WNF_SCOPE_INSTANCE *ScopeInstance,_WNF_STATE_NAME *StateName,undefined4 *param_3,_KPROCESS *param_4,
          _EX_RUNDOWN_REF **param_5)

{

[Truncated]

    uVar23 = (uint)((ulonglong)StateName >> 4) & 3;
    if ((PsInitialSystemProcess == Process) || (uVar23 != 3)) {
        SVar20 = 0xb8;
        if (*(longlong *)(param_3 + 2) == 0) {
            SVar20 = 0xa8;
        }
        NameInstance = (_WNF_NAME_INSTANCE *)ExAllocatePoolWithTag(PagedPool,SVar20,0x20666e57);
    }
    else {
        SVar20 = 0xb8;
        if (*(longlong *)(param_3 + 2) == 0) {

[1]

            SVar20 = 0xa8;
        }

[2]

        NameInstance = (_WNF_NAME_INSTANCE *)ExAllocatePoolWithQuotaTag(9,SVar20,0x20666e57);
    }
```

Chunk of size 0xa8 (0xc0) [1] is allocated from the PagedPool [2] as a structure of type _WNF_NAME_INSTANCE.

# _WNF_NAME_INSTANCE

Relevant fields:

-> StateName: Uniquely identifies the name instance [3]

-> StateData: Stores the data associated with the instance [4]

-> CreatorProcess: Stores the _EPROCESS structure of the process that created the name instance [5]

```
Offset     Length(bytes)     Field
-------    --------------    ---------------------------
0x0        0x4               Header
0x8        0x8               RunRef
0x10       0x18              TreeLinks

[3]

0x28       0x8               StateName
0x30       0x8               ScopeInstance
0x38       0x18              StateNameInfo
0x50       0x8               StateDataLock

[4]

0x58       0x8               StateData
0x60       0x4               CurrentChangeStamp
0x68       0x8               PermanentDataStore
0x70       0x8               StateSubscriptionListLock
0x78       0x10              StateSubscriptionListHead
0x88       0x10              TemporaryNameListEntry

[5]

0x98       0x8               CreatorProcess
0xa0       0x4               DataSubscribersCount
0xa4       0x4               CurrentDeliveryCount
```

# _WNF_STATE_DATA

-> Variable Size!!!!!!  Great for heap spraying.

-> Referred to when WNF State Data is updated or queried.

```
Offset      Length(bytes)       Field
-------     --------------      --------------

[6]

0x0         0x4                 Header

[7]

0x4         0x4                 AllocatedSize
0x8         0x4                 DataSize
0xc         0x4                 ChangeStamp
0x10        Variable            Data
```

# UPDATE WNF STATE DATA

-> WNF State Data is updated via the syscall - NtUpdateWnfStateData

-> Kernel internally calls ExpWnfWriteStateData()

```
void ExpWnfWriteStateData
               (_WNF_NAME_INSTANCE *NameInstance,void *InputBuffer,ulonglong Length,int MatchingChangeStamp,
               int CheckStamp)

{

[Truncated]

    if (NameInstance->StateData != (_WNF_STATE_DATA *)0x1) {

[8]

        StateData = NameInstance->StateData;
    }
    LengtH = (uint)(Length & 0xffffffff);

[9]

    if (((StateData == NULL) && ((NameInstance->PermanentDataStore != NULL || (LengtH != 0)))) ||

[10]

       ((StateData != NULL && (StateData->AllocatedSize < LengtH)))) {

[Truncated]

[11]

            StateData = (_WNF_STATE_DATA *)ExAllocatePoolWithQuotaTag(9,(ulonglong)(LengtH + 0x10),0x20666e57);

[Truncated]

[12]

        StateData->Header = (_WNF_NODE_HEADER)0x100904;
        StateData->AllocatedSize = LengtH;

[Truncated]

[13]

        RtlCopyMemory(StateData + 1,InputBuffer,Length & 0xffffffff);
        StateData->DataSize = LengtH;
        StateData->ChangeStamp = uVar5;

[Truncated]

    __security_check_cookie(local_30 ^ (ulonglong)&stack0xfffffffffffff08);
    return;
}
```

# UPDATE WNF STATE DATA

-> InputBuffer and Length parameters to the function contain the contents and size of the data (user controlled).

-> StateData is first retrieved from the related name instance [8].

-> If StateData is NULL (as is the case initially) at [9], or if the current size is lesser than the size of the new data [10], memory is allocated from the PagedPool for the new StateData pointer at [11]. It important to note that the size of allocation is the size of the new data (Length) plus 0x10, to account for the _WNF_STATE_DATA header. The Header and AllocateSize fields shown at [6] and [7] of the _WNF_STATE_DATA header are then initialized at [12].

-> If the current StateData is large enough for the new data, code execution from [8] jumps directly to [13]. Length bytes from InputBuffer are then copied into StateData at [13]. The DataSize field in the _WNF_STATE_DATA header is also filled at [13].

# DELETE WNF STATE NAME

-> A WNF State Name can be deleted via the syscall - NtDeleteWnfStateName

-> Frees the associated name instance (_WNF_NAME_INSTANCE) and StateData (_WNF_STATE_DATA) buffers described above.

# QUERY WNF STATE DATA

-> WNF State Data is queried via the syscall - NtQueryWnfStateData
-> Kernel internally calls ExpWnfReadStateData()

```
undefined4
ExpWnfReadStateData(_WNF_NAME_INSTANCE *NameInstance,undefined4 *param_2,void *OutBuf,uint OutBufSize,undefined4 *param_5)

{

[Truncated]

[14]

    StateData = NameInstance->StateData;
    if (StateData == NULL) {
        *param_2 = 0;
    }
    else {
        if (StateData != (_WNF_STATE_DATA *)0x1) {
            *param_2 = StateData->ChangeStamp;
            *param_5 = StateData->DataSize;

[15]

            if (OutBufSize < StateData->DataSize) {
                local_48 = 0xc0000023;
            }
            else {

[16]

                RtlCopyMemory(OutBuf,StateData + 1,(ulonglong)StateData->DataSize);
                local_48 = 0;
            }
            goto LAB_fffff8054ce2383f;
        }
        *param_2 = NameInstance->CurrentChangeStamp;
    }
```

-> OutBuf and OutBufSize are user controlled.

-> StateData is first retrieved from the related name instance [14].

-> If the output buffer is large enough to store the data (which is checked at [15]), StateData->DataSize bytes starting right after the StateData header are copied into the output buffer at [16].

# PIPE ATTRIBUTES

-> After the creation of a pipe, a user has the ability to add attributes to the pipe.
-> The attributes are a key-value pair, and are stored into a Linked List. The PipeAttributes object is allocated in the PagedPool.

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

-> Size and data PipeAttributes is user controlled.
-> AttributeName and AttributeValue are pointers to different offsets of the data field.

-> A PipeAttribute can be created on a Pipe using the NtfsControlFile syscall, and the 0x11003c control code.

-> The attribute's value can then be read using the 0x110038 control code. AttributeValue and AttributeValueSize will be used to read the attribute value and return it to the user.

# Common Log File System (CLFS)

-> File parsing in the kernel…... what could possibly go wrong???!!!

CLFS INTERNALS:

[GitHub - ionescu007/clfs-docs: Unofficial Common Log File System (CLFS) Documentation]

# CVE-2021-36955

RecordParamsPtr[iFlushBlockDup].pbImage
Is freed (L 820)

RecordParamsPtr[iFlushBlockDup].pbImage
Is set to
RecordParamsPtr[ShadowIndex].pbImage
(L 827)

BUT .........
RecordParamsPtr[iFlushBlockDup].pbImage
AND
RecordParamsPtr[ShadowIndex].pbImage

WERE POINTING TO THE SAME MEMORY!!!!

=> USE AFTER FREE!!!!!!!!!

```
787 long __thiscall
788 CClfsBaseFilePersisted::ExtendMetadataBlockDescriptor
789           (CClfsBaseFilePersisted *this,_CLFS_METADATA_BLOCK_TYPE iFlushBlock,unsigned_long cExtendSectors>>1)
790
791 {
792
793 [Truncated]
794
795     iFlushBlockDup = (ulonglong)iFlushBlock;
796     NewMetadataBlock = NULL;
797     RecordHeader = NULL;
798     iVar13 = 0;
799     uVar3 = this->m_cbRawSectorSize;
800     if (uVar3 == 0) {
801         NewSize = 0;
802     }
803     else {
804         NewSize = (uVar3 - 1) + this->m_rgBlocks[iFlushBlockDup].cbImage + cExtendSectors>>1 * 0x200 & -uVar3;
805     }
806     RecordsParamsPtr = this->m_rgBlocks;
807     pCVar1 = RecordsParamsPtr + iFlushBlockDup;
808     uVar4 = *(undefined4 *)&pCVar1->pbImage;
809     uVar5 = *(undefined4 *)((longlong)&pCVar1->pbImage + 4);
810     uVar3 = pCVar1->cbImage;
811     uVar6 = pCVar1->cbOffset;
812     CVar2 = RecordsParamsPtr[iFlushBlockDup].eBlockType;
813     ShadowIndex._0_4_ = iFlushBlock + ClfsMetaBlockControlShadow;
814     ShadowIndex = (ulonglong)(uint)ShadowIndex;
815     uVar7 = IsShadowBlock((CClfsBaseFilePersisted *)ShadowIndex,iFlushBlock,(uint)ShadowIndex);
816     if ((uVar7 == (unsigned_char)0x0) &&
817        (uVar7 = IsShadowBlock((CClfsBaseFilePersisted *)ShadowIndex,(unsigned_long)ShadowIndex,iFlushBlock),
818        uVar7 != (unsigned_char)0x0)) {
819         if (RecordsParamsPtr[iFlushBlockDup].pbImage != NULL) {
820             ExFreePoolWithTag(RecordsParamsPtr[iFlushBlockDup].pbImage,0);
821             this->m_rgBlocks[iFlushBlockDup].pbImage = NULL;
822             RecordsParamsPtr = this->m_rgBlocks;
823             ShadowIndex = (ulonglong)(iFlushBlock + ClfsMetaBlockControlShadow);
824         }
825         RecordsParamsPtr[iFlushBlockDup].cbImage = RecordsParamsPtr[ShadowIndex].cbImage;
826         m_rgBlocksDup = this->m_rgBlocks;
827         m_rgBlocksDup[iFlushBlockDup].pbImage = m_rgBlocks[ShadowIndex].pbImage;
828
829 [Truncated]
```

# EXPLOIT STEP 1

-> Spray large number of PipeAttributes of size 0x7a00 to use up all fragmented chunks in VSBackend and allocate new ones.
-> The last few will each be allocated on separate segments of size 0x11000, with the last (0x11000-0x7a00) bytes of each segment unused.

| | | | | | |
|---|---|---|---|---|---|
| PIPE ATTRIBUTE (0x7A00) | FREE (0x11000-0x7a00) | PIPE ATTRIBUTE (0x7A00) | FREE (0x11000-0x7a00) | PIPE ATTRIBUTE (0x7A00) | FREE (0x11000-0x7a00) |

# EXPLOIT STEP 2

-> Delete one of the later PipeAttributes.
-> This will consolidate the first 0x7a00 bytes with the remaining bytes in the rest of the segment, and put the entire segment back in the VS Backend.

# EXPLOIT STEP 3

-> Allocate the vulnerable chunk of size 0x7a00 by opening the malicious Base Log File.
-> This will get allocated from the freed segment in Step2. Similar to Step1, the last (0x11000-0x7a00) bytes will be unused.

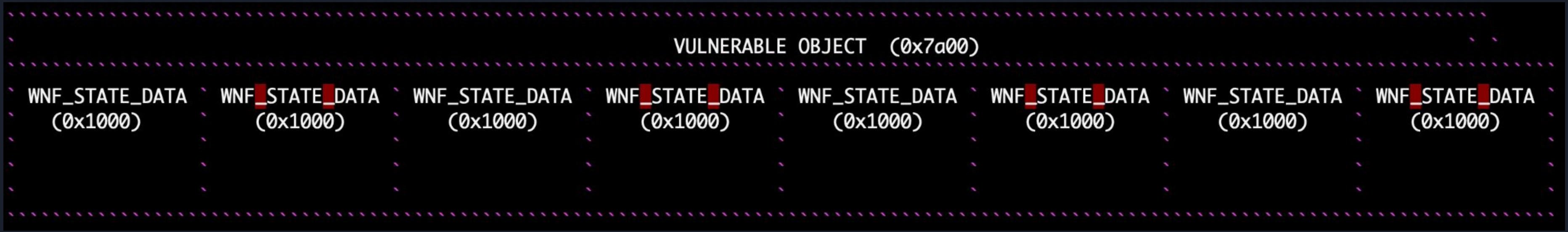
| | | | | | |
|---|---|---|---|---|---|
| PIPE ATTRIBUTE (0x7A00) | FREE (0x11000-0x7a00) | VULNERABLE OBJECT (0x7A00) | FREE (0x11000-0x7a00) | PIPE ATTRIBUTE (0x7A00) | FREE (0x11000-0x7a00) |

-> The vulnerable chunk will be freed for the first time shortly afterwards.
-> Similar to Step2, the entire segment will be back in the VS Backend.

| | | | | |
|---|---|---|---|---|
| PIPE ATTRIBUTE (0x7A00) | FREE (0x11000-0x7a00) | FREE (0x11000) | PIPE ATTRIBUTE (0x7A00) | FREE (0x11000-0x7a00) |

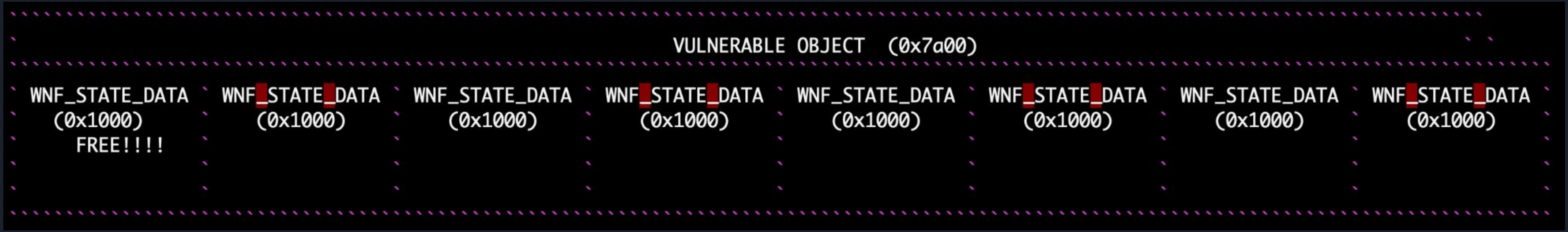
# EXPLOIT STEP 4

-> Spray large number of WNF_STATE_DATA objects of size 0x1000. This will first use up fragmented chunks in VS Backend and then the entire freed segment in Step3.
-> Note that no size lesser than 0x1000 (and maximum is 0x1000 for WNF_STATE_DATA objects) can be used because that will have an additional header that will corrupt the header in the vulnerable chunk, blocking a double free.



# EXPLOIT STEP 5

Free the vulnerable chunk for the second time. This will end up freeing the memory of one of the WNF_STATE_DATA objects allocated in Step4, without actually releasing the object.

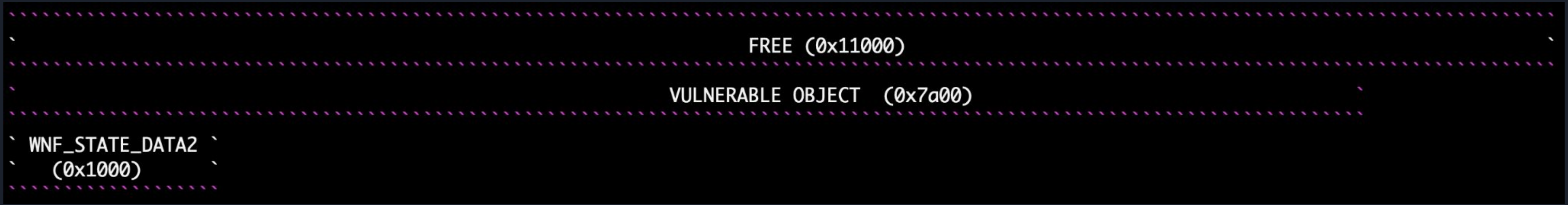


# EXPLOIT STEP 6

-> Allocate a WNF_STATE_DATA object of size 0x1000 over the freed chunk in Step5.
-> This will create 2 entirely overlapping WNF_STATE_DATA objects of size 0x1000.

# EXPLOIT STEP 7

-> Free all the WNF_STATE_DATA objects allocated in Step4. This will once again put the entire vulnerable segment (of size 0x11000) back in the VS Backend.



# EXPLOIT STEP 8

-> Spray large number of WNF_STATE_DATA objects of size 0x700, each with unique data.
-> This will first use up fragmented chunks in VS Backend and then the entire freed segment in Step7.
-> Note, here size 0x700 can be used because the rest of the exploit doesn't require any more freeing of the vulnerable chunk.
-> This creates 2 overlapping WNF_STATE_DATA objects, one of size 0x1000 (allocated in Step6) and other of size 0x700 (allocated here). Size 0x700 is specifically chosen for 2 reasons:

1. The additional chunk header (of size 0x10) in the 0x700-sized object means that the StateData header of the 0x1000-sized object is 0x10 bytes before the StateData header of the 0x700-sized object. Thus, the StateData header of the 0x700-sized object overlaps with the StateData data of the 0x1000-sized object.
2. Mentioned later

# EXPLOIT STEP 9

-> Update the StateData of the 0x1000-sized object to corrupt the StateData header of the 0x700-sized object such that the AllocatedSize and DataSize fields of the 0x700-sized object is increased from 0x6c0 (0x700-0x40) to 0x7000 each.
-> Now, querying or updating the 0x700-sized object will result in an out-of-bounds read/write into adjacent 0x700-sized WNF_STATE_DATA objects allocated in Step8.

```
VULNERABLE OBJECT  (0x7a00)

WNF_STATE_DATA2
   (0x1000)

CHUNK HEADER | WNF_STATE_DATA           | WNF_STATE_DATA | WNF_STATE_DATA | WNF_STATE_DATA | WNF_STATE_DATA | WNF_STATE_DATA
   (0x10)    |  (0x700)                 |  (0x700)       |   (0x700)      |   (0x700)      |   (0x700)      |   (0x700)
             |  Corrupted StateData Header
             |  (0x7000)
```

# EXPLOIT STEP 10

-> Identify the corrupted 0x700-sized WNF_STATE_DATA object by querying all of them with a Buffer size of 0x700.
-> All will return successfully except for the corrupted one, which will return with an error indicating that the buffer size is too small. This is because the DataSize field was increased (Step9).

StateData->DataSize is 0x7000 for the corrupted chunk and 0x6c0 for all others.

OutBufSize is 0x700

The check at [15] will only fail for the corrupted object

```
undefined4
ExpWnfReadStateData(_WNF_NAME_INSTANCE *NameInstance,undefined4 *param_2,void *OutBuf,uint OutBufSize,undefined4 *param_5)

{

[Truncated]

[14]

    StateData = NameInstance->StateData;
    if (StateData == NULL) {
        *param_2 = 0;
    }
    else {
        if (StateData != (_WNF_STATE_DATA *)0x1) {
            *param_2 = StateData->ChangeStamp;
            *param_5 = StateData->DataSize;

[15]

            if (OutBufSize < StateData->DataSize) {
                local_48 = 0xc0000023;
            }
            else {

[16]

                RtlCopyMemory(OutBuf,StateData + 1,(ulonglong)StateData->DataSize);
                local_48 = 0;
            }
            goto LAB_fffff8054ce2383f;
        }
        *param_2 = NameInstance->CurrentChangeStamp;
    }
```

```
VULNERABLE OBJECT (0x7a00)

                WNF_STATE_DATA2
                  (0x1000)

CHUNK HEADER | WNF_STATE_DATA          | WNF_STATE_DATA | WNF_STATE_DATA | WNF_STATE_DATA | WNF_STATE_DATA | WNF_STATE_DATA
   (0x10)    |  (0x700)                |  (0x700)       |  (0x700)       |  (0x700)       |  (0x700)       |  (0x700)
             |  Corrupted StateData Header
             |  (0x7000)
```

# EXPLOIT STEP 11

-> Query the corrupted 0x700-sized WNF_STATE_DATA object (identified in Step10) to further identify the next *2* adjacent WNF_STATE_DATA objects using the OOB read.

StateData->DataSize is 0x7000 for the corrupted chunk despite chunk size being 0x700.

OutBufSize is 0x7000

RtlCopyMemory() triggers OOB Read [16] => info leaks

```
undefined4
ExpWnfReadStateData(_WNF_NAME_INSTANCE *NameInstance,undefined4 *param_2,void *OutBuf,uint OutBufSize,undefined4 *param_5)

{

[Truncated]

[14]

    StateData = NameInstance->StateData;
    if (StateData == NULL) {
        *param_2 = 0;
    }
    else {
        if (StateData != (_WNF_STATE_DATA *)0x1) {
            *param_2 = StateData->ChangeStamp;
            *param_5 = StateData->DataSize;

[15]

            if (OutBufSize < StateData->DataSize) {
                local_48 = 0xc0000023;
            }
            else {

[16]

                RtlCopyMemory(OutBuf,StateData + 1,(ulonglong)StateData->DataSize);
                local_48 = 0;
            }
            goto LAB_fffff8054ce2383f;
        }
        *param_2 = NameInstance->CurrentChangeStamp;
    }
```

```
                                   VULNERABLE OBJECT  (0x7a00)

               WNF_STATE_DATA2
                  (0x1000)

 CHUNK HEADER  WNF_STATE_DATA               WNF_STATE_DATA  WNF_STATE_DATA  WNF_STATE_DATA  WNF_STATE_DATA  WNF_STATE_DATA
    (0x10)      (0x700)                      (0x700)          (0x700)         (0x700)         (0x700)         (0x700)
                Corrupted StateData Header
                (0x7000)
```

```
`                                      VULNERABLE OBJECT  (0x7a00)                                      `

`              WNF_STATE_DATA2                                  `
`                 (0x1000)                                      `

` CHUNK HEADER ` WNF_STATE_DATA              ` WNF_STATE_DATA       `WNF_STATE_DATA        `WNF_STATE_DATA `WNF_STATE_DATA `
`     (0x10)   `  (0x700)                    `  (0x700)             `  (0x700)            `  (0x700)      `  (0x700)      `
`              `  Corrupted StateData Header `                      `                     `               `               `
`              `  (0x7000)                   `                      `                     `               `               `
`              `  (Identified in Step10)     `(Identified in Step11) ` (Identified in Step11) `             `               `
```

# EXPLOIT STEP 12

Free the second newly identified WNF_STATE_DATA object of size 0x700.

```
` ` ` ` ` ` ` ` ` ` ` ` ` ` ` ` ` ` ` ` ` ` ` ` ` ` ` ` ` ` ` ` ` ` ` ` ` ` ` ` ` ` ` ` ` ` ` ` ` ` ` ` ` ` ` ` ` ` ` ` ` ` ` ` `
`                                     VULNERABLE OBJECT  (0x7a00)                                                   `
` ` ` ` ` ` ` ` ` ` ` ` ` ` ` ` ` ` ` ` ` ` ` ` ` ` ` ` ` ` ` ` ` ` ` ` ` ` ` ` ` ` ` ` ` ` ` ` ` ` ` ` ` ` ` ` ` ` ` ` ` ` ` ` `
`               WNF_STATE_DATA2                              `
`                  (0x1000)                                  `
` ` ` ` ` ` ` ` ` ` ` ` ` ` ` ` ` ` ` ` ` ` ` ` ` ` ` ` ` ` ` `
` CHUNK HEADER ` WNF_STATE_DATA           ` WNF_STATE_DATA      ` FREE                 `WNF_STATE_DATA `WNF_STATE_DATA `
`    (0x10)    `  (0x700)                 `  (0x700)            ` (0x700)              `   (0x700)     `   (0x700)     `
`              `  Corrupted StateData Header `                  `                      `               `               `
`              `  (0x7000)                `                     `                      `               `               `
`              `  (Identified in Step10)  `(Identified in Step11) ` (Identified in Step11) `             `               `
` ` ` ` ` ` ` ` ` ` ` ` ` ` ` ` ` ` ` ` ` ` ` ` ` ` ` ` ` ` ` ` ` ` ` ` ` ` ` ` ` ` ` ` ` ` ` ` ` ` ` ` ` ` ` ` ` ` ` ` ` ` `
```

# EXPLOIT STEP 13

-> Create a new process, which will run with the same privileges as the exploit process.

->  The token of this new process is allocated over the freed WNF_STATE_DATA object in Step12.

-> This is the second reason for choosing size 0x700, as the size of _TOKEN object is also 0x700.

```
VULNERABLE OBJECT  (0x7a00)

              WNF_STATE_DATA2
                 (0x1000)

CHUNK HEADER  WNF_STATE_DATA              WNF_STATE_DATA          _ETOKEN          WNF_STATE_DATA  WNF_STATE_DATA
   (0x10)      (0x700)                     (0x700)                (0x700)            (0x700)         (0x700)
               Corrupted StateData Header
               (0x7000)
               (Identified in Step10)     (Identified in Step11)
```

# EXPLOIT STEP 14

-> Query the corrupted 0x700-sized WNF_STATE_DATA object (identified in Step10) to identify the contents of _TOKEN.
-> Calculate the offset to the Privileges.Enabled and Privileges.Present fields in the _TOKEN object.

StateData->DataSize is 0x7000 for the corrupted chunk despite chunk size being 0x700.

OutBufSize is 0x7000

RtlCopyMemory() triggers OOB Read [16] => leak _TOKEN

```
undefined4
ExpWnfReadStateData(_WNF_NAME_INSTANCE *NameInstance,undefined4 *param_2,void *OutBuf,uint OutBufSize,undefined4 *param_5)

{

[Truncated]

[14]

    StateData = NameInstance->StateData;
    if (StateData == NULL) {
        *param_2 = 0;
    }
    else {
        if (StateData != (_WNF_STATE_DATA *)0x1) {
            *param_2 = StateData->ChangeStamp;
            *param_5 = StateData->DataSize;

[15]

            if (OutBufSize < StateData->DataSize) {
                local_48 = 0xc0000023;
            }
            else {

[16]

                RtlCopyMemory(OutBuf,StateData + 1,(ulonglong)StateData->DataSize);
                local_48 = 0;
            }
            goto LAB_fffff8054ce2383f;
        }
        *param_2 = NameInstance->CurrentChangeStamp;
    }
```

<table>
<tr><td colspan="6">VULNERABLE OBJECT (0x7a00)</td></tr>
<tr><td></td><td colspan="2">WNF_STATE_DATA2<br>(0x1000)</td><td colspan="3"></td></tr>
<tr><td>CHUNK HEADER<br>(0x10)</td><td>WNF_STATE_DATA<br>(0x700)<br>Corrupted StateData Header<br>(0x7000)<br>(Identified in Step10)</td><td>WNF_STATE_DATA<br>(0x700)<br><br><br>(Identified in Step11)</td><td>_ETOKEN<br>(0x700)</td><td>WNF_STATE_DATA<br>(0x700)</td><td>WNF_STATE_DATA<br>(0x700)</td></tr>
</table>

# EXPLOIT STEP 15

-> Update the corrupted 0x700-sized WNF_STATE_DATA object to corrupt the first adjacent object (identified in Step11) using the OOB write.
-> Increase AllocatedSize and DataSize in StateData (refer Step9) to 0x1000.

StateData->AllocatedSize is 0x7000 for the corrupted chunk despite chunk size being 0x700.

LengthH is user-controlled (>0x700 for OOB write [10]).

RtlCopyMemory() triggers OOB Write [13] => corrupt next WNF_STATE_DATA

```
void ExpWnfWriteStateData
               (_WNF_NAME_INSTANCE *NameInstance,void *InputBuffer,ulonglong Length,int MatchingChangeStamp,
               int CheckStamp)

{
[Truncated]

    if (NameInstance->StateData != (_WNF_STATE_DATA *)0x1) {

[8]

        StateData = NameInstance->StateData;
    }
    LengthH = (uint)(Length & 0xffffffff);

[9]

    if (((StateData == NULL) && ((NameInstance->PermanentDataStore != NULL || (LengthH != 0)))) ||

[10]

       ((StateData != NULL && (StateData->AllocatedSize < LengthH)))) {

[Truncated]

[11]

            StateData = (_WNF_STATE_DATA *)ExAllocatePoolWithQuotaTag(9,(ulonglong)(LengthH + 0x10),0x20666e57);

[Truncated]

[12]

        StateData->Header = (_WNF_NODE_HEADER)0x100904;
        StateData->AllocatedSize = LengthH;

[Truncated]

[13]

        RtlCopyMemory(StateData + 1,InputBuffer,Length & 0xffffffff);
        StateData->DataSize = LengthH;
        StateData->ChangeStamp = uVar5;

[Truncated]

    __security_check_cookie(local_30 ^ (ulonglong)&stack0xfffffffffffffff08);
    return;
}
```

```
................................................................................................................
`                                        VULNERABLE OBJECT  (0x7a00)                                           `
................................................................................................................
`              WNF_STATE_DATA2                                  `
`                (0x1000)                                       `
.................................................................
` CHUNK HEADER ` WNF_STATE_DATA             ` WNF_STATE_DATA     ` _ETOKEN       `WNF_STATE_DATA `WNF_STATE_DATA `
`    (0x10)    `  (0x700)                   `  (0x700)           ` (0x700)       `  (0x700)      `  (0x700)      `
`              `  Corrupted StateData Header `                   `               `               `               `
`              `  (0x7000)                  `                    `               `               `               `
`              `  (Identified in Step10)    `(Identified in Step11) `            `               `               `
..................................................................................................
```

```
`                                                VULNERABLE OBJECT  (0x7a00)                                              `

`              WNF_STATE_DATA2                                    `
`                (0x1000)                                         `

` CHUNK HEADER ` WNF_STATE_DATA            ` WNF_STATE_DATA            `  _ETOKEN        `WNF_STATE_DATA `WNF_STATE_DATA `
`    (0x10)    `  (0x700)                 `  (0x700)                  `  (0x700)        `  (0x700)      `  (0x700)      `
`              `  Corrupted StateData Header ` Corrupted StateData Header `              `               `               `
`              `  (0x7000)                `  (0x1000)                 `                 `               `               `
`              `  (Identified in Step10)  `(Identified in Step11)      `                 `               `               `
```

# EXPLOIT STEP 16

-> Update the most recent corrupted WNF_STATE_DATA object (Step 15) to corrupt the adjacent _TOKEN object using the OOB write.
-> Overwrite Privileges.Enabled and Privileges.Present in _TOKEN to 0xFFFFFFFFFFFFFFFF, thereby setting all the privileges. This completes the PE.

StateData->AllocatedSize is 0x1000 for the second corrupted chunk despite chunk size being 0x700.

LengthH is user-controlled (>0x700 for OOB write [10]).

RtlCopyMemory() triggers OOB Write [13] => corrupt _TOKEN

```
void ExpWnfWriteStateData
               (_WNF_NAME_INSTANCE *NameInstance,void *InputBuffer,ulonglong Length,int MatchingChangeStamp,
               int CheckStamp)

{
[Truncated]

    if (NameInstance->StateData != (_WNF_STATE_DATA *)0x1) {

[8]

        StateData = NameInstance->StateData;
    }
    LengthH = (uint)(Length & 0xffffffff);

[9]

    if (((StateData == NULL) && ((NameInstance->PermanentDataStore != NULL || (LengthH != 0)))) ||

[10]

       ((StateData != NULL && (StateData->AllocatedSize < LengthH)))) {

[Truncated]

[11]

            StateData = (_WNF_STATE_DATA *)ExAllocatePoolWithQuotaTag(9,(ulonglong)(LengthH + 0x10),0x20666e57);

[Truncated]

[12]

        StateData->Header = (_WNF_NODE_HEADER)0x100904;
        StateData->AllocatedSize = LengthH;

[Truncated]

[13]

        RtlCopyMemory(StateData + 1,InputBuffer,Length & 0xffffffff);
        StateData->DataSize = LengthH;
        StateData->ChangeStamp = uVar5;

[Truncated]

    __security_check_cookie(local_30 ^ (ulonglong)&stack0xfffffffffffff08);
    return;
}
```

```
`````````````````````````````````````````````````````````````````````````````````````````````````````````````````
`                                          VULNERABLE OBJECT  (0x7a00)                                          `
`````````````````````````````````````````````````````````````````````````````````````````````````````````````````
`              WNF_STATE_DATA2                                    `
`                 (0x1000)                                        `
```````````````````````````````````````````````````````````````````
` CHUNK HEADER ` WNF_STATE_DATA            ` WNF_STATE_DATA            `  _ETOKEN        `WNF_STATE_DATA `WNF_STATE_DATA `
`    (0x10)    `  (0x700)                  `  (0x700)                  `  (0x700)        `  (0x700)      `  (0x700)      `
`              `  Corrupted StateData Header ` Corrupted StateData Header ` CORRUPTED!!!!!!! `               `               `
`              `  (0x7000)                 `  (0x1000)                 `                 `               `               `
`              `  (Identified in Step10)   `(Identified in Step11)     `                 `               `               `
``````````````````````````````````````````````````````````````````````````````````````````````````````````````````
```

# RECAP

PRIMITIVE:  UAF in 0x7a00-sized chunk in PagedPool

UAF => DOUBLE FREE => OVERLAPPING OBJECTS => TYPE CONFUSION => OOB R/W => LPE!!!!!

# CHECKING EXPLOITABILITY

FIRST CONVERT YOUR PRIMITIVE(S) INTO A PAGED POOL OVERFLOW

CATEGORY 1: vulnerable chunk > 0x200

Case1: Controlled size and data of overflow => definitely exploitable

Case2: Controlled size but uncontrolled data => if data at offset corresponding to a nearby StateData->AllocatedSize is such that AllocatedSize can be increased, then exploitable (Increase StateData->AllocatedSize of adjacent chunk and proceed .......)

Case3: Uncontrolled size but controlled data =>

Subcase1: 0x200<vulnerable chunk size<0x1000 => if size > 0x25, exploitable
(partial overwrite to Increase StateData->AllocatedSize of adjacent chunk and proceed .......)

Subcase2: vulnerable chunk size>0x1000 => if size > 0x15, exploitable
(partial overwrite to Increase StateData->AllocatedSize of adjacent chunk and proceed .......)

Case4: Uncontrolled size and uncontrolled data => Intersection of Case 2,3

# RELATED CVES

EACH OF

6 of the 7 CVEs SATISFY EITHER ONE OF THE CASES => ALL EXPLOITABLE !!!!!!!!
HEAP GROOMING IS UNIQUE FOR EACH CASE THOUGH

# REMAINING QUESTIONS

-> What about the 7th CVE?

-> What about category 2, when vulnerable chunk size < 0x200?

-> Where’s my ARBITRARY READ/WRITE????!!!!!!

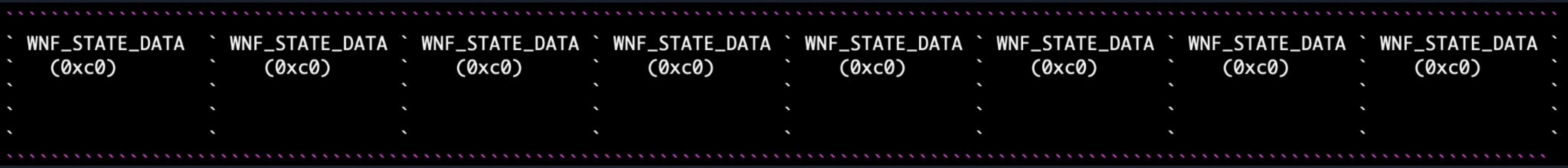
# SPECIAL CASE

VULNERABLE CHUNK SIZE == 0xC0

# EXPLOIT STEP 1

-> Spray large number of WNF_STATE_DATA objects of size 0xc0 to use up all fragmented chunks in LFH Backend and allocate new ones.

-> The last few will be contiguous.

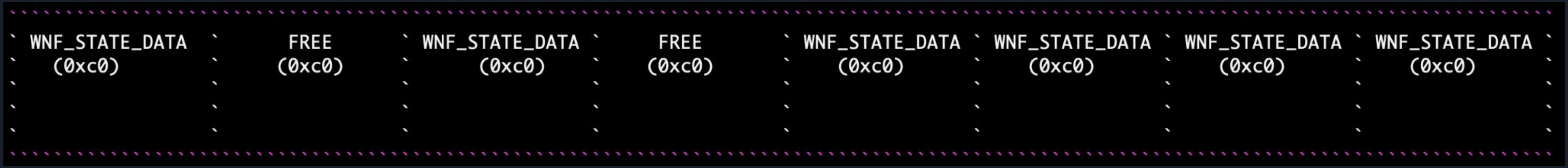


# EXPLOIT STEP 2

-> Delete 2 of the last few WNF_STATE_DATA objects to create 2 holes in the spray.

-> The freed chunks will go into the Lookaside List for size 0xc0.

NOTE: Ensure that the Lookaside List for size 0xc0 is enabled by using objects of size 0xc0 prior to the spray



# EXPLOIT STEP 3

-> Allocate the vulnerable chunk and a WNF_NAME_INSTANCE object nearby to each other, with a WNF_STATE_DATA object in between.

| Offset | Length(bytes) | Field |
|---|---|---|
| 0x0 | 0x4 | Header |
| 0x8 | 0x8 | RunRef |
| 0x10 | 0x18 | TreeLinks |
| [3] | | |
| 0x28 | 0x8 | StateName |
| 0x30 | 0x8 | ScopeInstance |
| 0x38 | 0x18 | StateNameInfo |
| 0x50 | 0x8 | StateDataLock |
| [4] | | |
| 0x58 | 0x8 | StateData |
| 0x60 | 0x4 | CurrentChangeStamp |
| 0x68 | 0x8 | PermanentDataStore |
| 0x70 | 0x8 | StateSubscriptionListLock |
| 0x78 | 0x10 | StateSubscriptionListHead |
| 0x88 | 0x10 | TemporaryNameListEntry |
| [5] | | |
| 0x98 | 0x8 | CreatorProcess |
| 0xa0 | 0x4 | DataSubscribersCount |
| 0xa4 | 0x4 | CurrentDeliveryCount |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| WNF_STATE_DATA (0xc0) | VULNERABLE (0xc0) | WNF_STATE_DATA (0xc0) | WNF_NAME_INSTANCE (0xc0) | WNF_STATE_DATA (0xc0) | WNF_STATE_DATA (0xc0) | WNF_STATE_DATA (0xc0) | WNF_STATE_DATA (0xc0) |

# EXPLOIT STEP 4-6

-> Follow Steps 9-11 (previous exploit) to leak to corrupt the adjacent WNF_STATE_DATA and leak the contents of WNF_NAME_INSTANCE

```
` WNF_STATE_DATA `   VULNERABLE   `      WNF_STATE_DATA       ` WNF_NAME_INSTANCE ` WNF_STATE_DATA ` WNF_STATE_DATA ` WNF_STATE_DATA `
`    (0xc0)      `     (0xc0)     `         (0xc0)            `      (0xc0)       `     (0xc0)     `     (0xc0)     `     (0xc0)     `
`                `                ` Corrupted StateData Header`   (Identified)    `                `                `                `
`                `                `         (0x100)           `                   `                `                `                `
`                `                `       (Identified)        `                   `                `                `                `
```

# EXPLOIT STEP 7

-> Follow Step 15 (previous exploit) to corrupt StateData

| Offset | Length(bytes) | Field |
|---|---|---|
| 0x0 | 0x4 | Header |
| 0x8 | 0x8 | RunRef |
| 0x10 | 0x18 | TreeLinks |
| [3] | | |
| 0x28 | 0x8 | StateName |
| 0x30 | 0x8 | ScopeInstance |
| 0x38 | 0x18 | StateNameInfo |
| 0x50 | 0x8 | StateDataLock |
| [4] | | |
| 0x58 | 0x8 | StateData |
| 0x60 | 0x4 | CurrentChangeStamp |
| 0x68 | 0x8 | PermanentDataStore |
| 0x70 | 0x8 | StateSubscriptionListLock |
| 0x78 | 0x10 | StateSubscriptionListHead |
| 0x88 | 0x10 | TemporaryNameListEntry |
| [5] | | |
| 0x98 | 0x8 | CreatorProcess |
| 0xa0 | 0x4 | DataSubscribersCount |
| 0xa4 | 0x4 | CurrentDeliveryCount |

| | | | | | | |
|---|---|---|---|---|---|---|
| WNF_STATE_DATA (0xc0) | VULNERABLE (0xc0) | WNF_STATE_DATA (0xc0) Corrupted StateData Header (0x100) (Identified) | WNF_NAME_INSTANCE (0xc0) (Identified) (Corrupted) | WNF_STATE_DATA (0xc0) | WNF_STATE_DATA (0xc0) | WNF_STATE_DATA (0xc0) |

# EXPLOIT STEP 8

-> Querying the corrupted StateData gives ARBITRARY READ

StateData [14] is user controlled!!!

Arbitrary read of user-controlled bytes at [16]

```
undefined4
ExpWnfReadStateData(_WNF_NAME_INSTANCE *NameInstance,undefined4 *param_2,void *OutBuf,uint OutBufSize,undefined4 *param_5)

{

[Truncated]

[14]

    StateData = NameInstance->StateData;
    if (StateData == NULL) {
        *param_2 = 0;
    }
    else {
        if (StateData != (_WNF_STATE_DATA *)0x1) {
            *param_2 = StateData->ChangeStamp;
            *param_5 = StateData->DataSize;

[15]

            if (OutBufSize < StateData->DataSize) {
                local_48 = 0xc0000023;
            }
            else {

[16]

                RtlCopyMemory(OutBuf,StateData + 1,(ulonglong)StateData->DataSize);
                local_48 = 0;
            }
            goto LAB_fffff8054ce2383f;
        }
        *param_2 = NameInstance->CurrentChangeStamp;
    }
```

# EXPLOIT STEP 9

-> Updating the corrupted StateData gives ARBITRARY WRITE

StateData [8] is user controlled!!!

Arbitrary write of user-controlled bytes at [13]

```
void ExpWnfWriteStateData
               (_WNF_NAME_INSTANCE *NameInstance,void *InputBuffer,ulonglong Length,int MatchingChangeStamp,
               int CheckStamp)

{

[Truncated]

    if (NameInstance->StateData != (_WNF_STATE_DATA *)0x1) {

[8]

        StateData = NameInstance->StateData;
    }
    LengthH = (uint)(Length & 0xffffffff);

[9]

    if (((StateData == NULL) && ((NameInstance->PermanentDataStore != NULL || (LengthH != 0)))) ||

[10]

       ((StateData != NULL && (StateData->AllocatedSize < LengthH)))) {

[Truncated]

[11]

            StateData = (_WNF_STATE_DATA *)ExAllocatePoolWithQuotaTag(9,(ulonglong)(LengthH + 0x10),0x20666e57);

[Truncated]

[12]

        StateData->Header = (_WNF_NODE_HEADER)0x100904;
        StateData->AllocatedSize = LengthH;

[Truncated]

[13]

        RtlCopyMemory(StateData + 1,InputBuffer,Length & 0xffffffff);
        StateData->DataSize = LengthH;
        StateData->ChangeStamp = uVar5;

[Truncated]

    __security_check_cookie(local_30 ^ (ulonglong)&stack0xfffffffffffff08);
    return;
}
```

# EXPLOIT STEP 10

-> We already know the _EPROCESS address of the exploit process ( Step 4-6 )
-> Use Arbitrary Read + Infoleaks to find the _TOKEN bits to be set
-> Use Arbitrary Write to set _TOKEN bits (Step 16 of previous exploit)

# WHAT ABOUT FOR SIZES < 0x200 but != 0xc0 HERE’S THE TRUTH……..

ANY OF THESE METHODS CAN BE USED REGARDLESS OF VULNERABLE CHUNK SIZE!!!!!

PLAY AROUND WITH HEAP GROOMING:

http://web.archive.org/web/20211113145025/https://www.sstic.org/media/SSTIC2020/SSTIC-actes/pool_overflow_exploitation_since_windows_10_19h1/SSTIC2020-Article-pool_overflow_exploitation_since_windows_10_19h1-bayet_fariello.pdf

# QUESTIONS ????

THANK YOU!!!

FEEL FREE TO CONTACT ME ANYTIME:

@AravGarg3
cyber8knight#1645