NTFS.sys crash

Marius TIVADAR
marius.tivadar@gmail.com

July 19, 2017

1 Introduction

Type of issue: denial of service. One can generate blue-screen-of-death using a handcrafted NTFS image. This Denial of Service type of attack, can be driven from user mode, limited user account or Administrator. It can even crash the system if it is in locked state. In the end I will discuss possible impact of this attack and what do I think it should be changed in the way the system works in locked state, from security point of view.

Below you can find the documents:

Material: **This** document, explaining how-to.

Material: PoC videos https://photos.app.goo.gl/rCe3tZYmjgEaTpFl2

Material: PoC forged 10MB NTFS image that will crash the system
https://drive.google.com/drive/folders/0B7FTEvpR6pGTbzRkLU5QeFdCZ3c

1.1 Affected systems

1. Windows 7 Enterprise 6.1.7601 SP1, Build 7601 x64
2. Windows 10 Pro 10.0.15063, Build 15063 x64
3. Windows 10 Enterprise Evaluation Insider Preview 10.0.16215, Build 16215 x64

Note: these are the only systems I have tested.

2 Preparing the image

As a proof of concept, I attached a 10MB NTFS image.
The attack consists of modifying **root** directory name, also it's **INDEX_ALLOCATION** in three places.

1. In the PoC image, I took file record 5 (root), modified it's name: '.' to '4', offset 0x3564da in file.
2. Then, in INDEX_ALLOCATION of root, we take an arbitrary entry and overwrite it's file name with the same name as modified root, in our case '4', offset 0x02c542 in file.
3. Also, in the same INDEX_ALLOCATION entries, we take the entry that contains the root directory '.' , we modify it's name with '4', offset 0x02c4ea in file.

note: no other file record than #5 has been modified. The patches reside only in INDEX_ALLOCATION.

3 Dissection of image

Listing file record #5 will look like below. Please note the modified '.' entry.

```
|- 4                 ---> (modified root name)
   |- $AttrDef
   |- $BadClus
   |- $Bitmap
   |- $Boot
   |- $Extend
      |- $Deleted
      |- $ObjId
      |- $Quota
      |- $Reparse
      |- $RmMetadata
   |- $LogFile
   |- $MFT
   |- $MFTMirr
   |- $Secure
   |- $UpCase
   |- $Volume
   |- 4               ---> (modified root name)
   |- 4               ---> (modified arbitrary file name)
   |- bigfile.exe
   |- kuku
      |- examplefile
   |- System Volume Information
      |- IndexerVolumeGuid
      |- WPSettings.dat
```

3.1 Some info from image parsing

Here we can see file name modified for record #5

```
DEBUG:fs_ntfs:==================== [File record #5] ====================
DEBUG:fs_ntfs:Offset to update sequence: 0x30
DEBUG:fs_ntfs:Size in words of update sequence: 0x3
...
DEBUG:fs_ntfs:
DEBUG:fs_ntfs:---=== attributes ======
DEBUG:fs_ntfs:Attribute type: $STANDARD_INFORMATION
DEBUG:fs_ntfs:Length: 0x48
DEBUG:fs_ntfs:Non-resident flag: 0x0, name length: 0x0
DEBUG:fs_ntfs:Attribute is: resident, not named
DEBUG:fs_ntfs:Length of the attribute: 0x30
DEBUG:fs_ntfs:
DEBUG:fs_ntfs:Attribute type: $FILE_NAME
```

```
DEBUG:fs_ntfs:Length: 0x60
DEBUG:fs_ntfs:Non-resident flag: 0x0, name length: 0x0
DEBUG:fs_ntfs:Attribute is: resident, not named
DEBUG:fs_ntfs:Length of the attribute: 0x44
DEBUG:fs_ntfs:Allocated size of file: 0x0
DEBUG:fs_ntfs:Real size of file: 0x0
DEBUG:fs_ntfs:Flags: 0x10000006
DEBUG:fs_ntfs:Filename namespace: 3
DEBUG:fs_ntfs:File name: 4                    ---> '.' was modified into '4'
```

We can see modifications from INDEX_ALLOCATION below.

```
DEBUG:fs_ntfs:Attribute type: $INDEX_ALLOCATION
DEBUG:fs_ntfs:Length: 0x50
DEBUG:fs_ntfs:Non-resident flag: 0x1, name length: 0x4
DEBUG:fs_ntfs:Attribute is: non resident, named
DEBUG:fs_ntfs:Starting VCN: 0x0, last VCN: 0x0
DEBUG:fs_ntfs:Attribute name: $I30
DEBUG:fs_ntfs:Real size of the attribute: 0x1000
DEBUG:fs_ntfs.mft:LCN relative 0x0000002c, length_size: 0x1,
offset_size: 0x1, n_clusters: 0x0001, LCN start: 0x002c
DEBUG:fs_ntfs.mft:
DEBUG:fs_ntfs:0x0001 clusters @ LCN 0x0000002c, @ f_offset 0x2c000,
size_in_bytes 4,096
...
DEBUG:fs_ntfs:Iterating $I30 index...
DEBUG:fs_ntfs:
DEBUG:fs_ntfs:-= index entry =-
DEBUG:fs_ntfs:Index flags: 0x0
DEBUG:fs_ntfs:Real size of file: 0
DEBUG:fs_ntfs:Filename namespace: 3
DEBUG:fs_ntfs:Length of the filename: 0x8
DEBUG:fs_ntfs:Offset to filename: 0x52
DEBUG:fs_ntfs:Filename: $AttrDef
DEBUG:fs_ntfs:
DEBUG:fs_ntfs:-= index entry =-
DEBUG:fs_ntfs:Index flags: 0x0
DEBUG:fs_ntfs:Real size of file: 0
DEBUG:fs_ntfs:Filename namespace: 3
DEBUG:fs_ntfs:Length of the filename: 0x8
DEBUG:fs_ntfs:Offset to filename: 0x52
DEBUG:fs_ntfs:Filename: $BadClus
DEBUG:fs_ntfs:
DEBUG:fs_ntfs:-= index entry =-
DEBUG:fs_ntfs:Index flags: 0x0
DEBUG:fs_ntfs:Real size of file: 0
DEBUG:fs_ntfs:Filename namespace: 3
DEBUG:fs_ntfs:Length of the filename: 0x7
DEBUG:fs_ntfs:Offset to filename: 0x50
DEBUG:fs_ntfs:Filename: $Bitmap
```

```
DEBUG:fs_ntfs:
DEBUG:fs_ntfs:-= index entry =-
DEBUG:fs_ntfs:Index flags: 0x0
DEBUG:fs_ntfs:Real size of file: 0
DEBUG:fs_ntfs:Filename namespace: 3
DEBUG:fs_ntfs:Length of the filename: 0x5
DEBUG:fs_ntfs:Offset to filename: 0x4C
DEBUG:fs_ntfs:Filename: $Boot
DEBUG:fs_ntfs:
DEBUG:fs_ntfs:-= index entry =-
DEBUG:fs_ntfs:Index flags: 0x0
DEBUG:fs_ntfs:Real size of file: 0
DEBUG:fs_ntfs:Filename namespace: 3
DEBUG:fs_ntfs:Length of the filename: 0x7
DEBUG:fs_ntfs:Offset to filename: 0x50
DEBUG:fs_ntfs:Filename: $Extend
DEBUG:fs_ntfs:
DEBUG:fs_ntfs:-= index entry =-
DEBUG:fs_ntfs:Index flags: 0x0
DEBUG:fs_ntfs:Real size of file: 0
DEBUG:fs_ntfs:Filename namespace: 3
DEBUG:fs_ntfs:Length of the filename: 0x8
DEBUG:fs_ntfs:Offset to filename: 0x52
DEBUG:fs_ntfs:Filename: $LogFile
DEBUG:fs_ntfs:
DEBUG:fs_ntfs:-= index entry =-
DEBUG:fs_ntfs:Index flags: 0x0
DEBUG:fs_ntfs:Real size of file: 16,384
DEBUG:fs_ntfs:Filename namespace: 3
DEBUG:fs_ntfs:Length of the filename: 0x4
DEBUG:fs_ntfs:Offset to filename: 0x4A
DEBUG:fs_ntfs:Filename: $MFT
DEBUG:fs_ntfs:
DEBUG:fs_ntfs:-= index entry =-
DEBUG:fs_ntfs:Index flags: 0x0
DEBUG:fs_ntfs:Real size of file: 0
DEBUG:fs_ntfs:Filename namespace: 3
DEBUG:fs_ntfs:Length of the filename: 0x8
DEBUG:fs_ntfs:Offset to filename: 0x52
DEBUG:fs_ntfs:Filename: $MFTMirr
DEBUG:fs_ntfs:
DEBUG:fs_ntfs:-= index entry =-
DEBUG:fs_ntfs:Index flags: 0x0
DEBUG:fs_ntfs:Real size of file: 0
DEBUG:fs_ntfs:Filename namespace: 3
DEBUG:fs_ntfs:Length of the filename: 0x7
DEBUG:fs_ntfs:Offset to filename: 0x50
DEBUG:fs_ntfs:Filename: $Secure
DEBUG:fs_ntfs:
DEBUG:fs_ntfs:-= index entry =-
```

```
DEBUG:fs_ntfs:Index flags: 0x0
DEBUG:fs_ntfs:Real size of file: 0
DEBUG:fs_ntfs:Filename namespace: 3
DEBUG:fs_ntfs:Length of the filename: 0x7
DEBUG:fs_ntfs:Offset to filename: 0x50
DEBUG:fs_ntfs:Filename: $UpCase
DEBUG:fs_ntfs:
DEBUG:fs_ntfs:-= index entry =-
DEBUG:fs_ntfs:Index flags: 0x0
DEBUG:fs_ntfs:Real size of file: 0
DEBUG:fs_ntfs:Filename namespace: 3
DEBUG:fs_ntfs:Length of the filename: 0x7
DEBUG:fs_ntfs:Offset to filename: 0x50
DEBUG:fs_ntfs:Filename: $Volume
DEBUG:fs_ntfs:
DEBUG:fs_ntfs:-= index entry =-
DEBUG:fs_ntfs:Index flags: 0x0
DEBUG:fs_ntfs:Real size of file: 0
DEBUG:fs_ntfs:Filename namespace: 3
DEBUG:fs_ntfs:Length of the filename: 0x1
DEBUG:fs_ntfs:Offset to filename: 0x44
DEBUG:fs_ntfs:Filename: 4     ---> this was the root directory '.'
DEBUG:fs_ntfs:
DEBUG:fs_ntfs:-= index entry =-
DEBUG:fs_ntfs:Index flags: 0x0
DEBUG:fs_ntfs:Real size of file: 26,453
DEBUG:fs_ntfs:Filename namespace: 0
DEBUG:fs_ntfs:Length of the filename: 0xE
DEBUG:fs_ntfs:Offset to filename: 0x5E
DEBUG:fs_ntfs:Filename: 4     ---> this was a file name we modified.
                                   length of filename remained the
                                   original one.
```

4 Post mortem

4.1 Context

```
EXCEPTION_RECORD:  fffff88004437ad8 -- (.exr 0xfffff88004437ad8)
ExceptionAddress: fffff8800123040e (Ntfs!NtfsFindExistingLcb+0x0000000000000062)
   ExceptionCode: c0000005 (Access violation)
  ExceptionFlags: 00000000
NumberParameters: 2
   Parameter[0]: 0000000000000000
   Parameter[1]: 0000000000000070
Attempt to read from address 0000000000000070

CONTEXT:  fffff88004437330 -- (.cxr 0xfffff88004437330)
rax=0000000000000000 rbx=fffff8a00165d450 rcx=fffffa8002290010
```

```
rdx=fffff8a00165d010 rsi=fffff88004438058 rdi=fffff8a00165d030
rip=fffff8800123040e rsp=fffff88004437d10 rbp=fffff8a00165d010
 r8=fffff8a00165d010  r9=fffff88004438058 r10=0000000000000001
r11=fffff88004437da0 r12=fffff88004438003 r13=fffff88004438058
r14=fffff8a00165d140 r15=fffffa8002290010
```

4.2 Instruction that fails

```
FOLLOWUP_IP:
Ntfs!NtfsFindExistingLcb+62
fffff880`0123040e 483b6870        cmp     rbp,qword ptr [rax+70h]
```

rax is **NULL**, and we get access violation on read.

4.3 Stack trace

```
Ntfs!NtfsFindExistingLcb+0x62
Ntfs!NtfsCreateLcb+0x7c
Ntfs!NtfsOpenFile+0x3b9
Ntfs!NtfsCommonCreate+0xc49
Ntfs!NtfsCommonCreateCallout+0x1d
nt!KeExpandKernelStackAndCalloutEx+0xd8
Ntfs!NtfsCommonCreateOnNewStack+0x4f
Ntfs!NtfsFsdCreate+0x1ac
fltmgr!FltpLegacyProcessingAfterPreCallbacksCompleted+0x24f
fltmgr!FltpCreate+0x2a9
nt!IopParseDevice+0x5a5
nt!ObpLookupObjectName+0x588
nt!ObOpenObjectByName+0x306
nt!IopCreateFile+0x2bc
nt!NtCreateFile+0x78
nt!KiSystemServiceCopyEnd+0x13
ntdll!ZwCreateFile+0xa
```

4.4 Analysis

The bug is in `NtfsFindExistingLcb()` function that doesn't check null pointer taken from FCB structure.

```
.text:000000000002340A                mov     rax, [rbx+18h]  ; rax is null
.text:000000000002340E                cmp     rbp, [rax+70h]
```

`NtfsOpenFile()` calls `NtfsCreateFcb()` that will return the structure containing the null pointer. This gets passed down to `NtfsCreateLcb()` and then to `NtfsFindExistingLcb()` where the crash takes place.

`NtfsCreateFcb()` takes the entry from the AVL tree, my investigation stops here. No other kind of exploitation was possible for the moment.

5 Impact of the issue

Auto-play is activated by default, this leads to automatically crashing the system when usb stick is inserted.
Even with auto-play disabled, system will crash when the file is accessed. This can be done for eg. when Windows Defender scans the usb stick, or any other tool opening it.
If none the above, finally, if the user clicks on the file, system will crash.

other possibilities: an even scarier behavior is when the user locks his/hers computer, and an attacker could insert the usb stick, system will also crash because auto-play still takes action. I **strongly believe** that this behavior should be changed, while no usb stick/volume should be mounted when the system is locked. Generally speaking, no driver should be loaded, no code should get executed when the system is locked and external peripherals are inserted into the machine. I may think this as, code gets executed without user consent. If this kind of crash was exploitable, and attacker could load malware even if the system is locked, this could open thousands of possible scenarios.
Of course, it is not necessary to have an usb stick. A malware for example could drop a tiny ntfs image and mount it somehow, thus triggering the crash.