University of Amsterdam

MSc Security and Network Engineering

Research Project II

Low-level writing to NTFS file systems

Rick van Gorp
rick.vangorp@os3.nl

July 8, 2018

Supervisor:
Cedric van Bockhaven
Deloitte

Abstract

Red teams sometimes have to deal with files on NTFS drives that are non-writable or endpoint security solutions that have on-access scanning and application control features [12] [9]. These features could block writes of data to the NTFS file system that may be required for red teams to compromise and control a system. An example of a case where non-writable files are encountered is where an infrastructure has to be infiltrated silently by creating a user account in a domain. The active directory database that holds this information is locked and not accessible with administrative privileges. Using write experiments, we have attempted to write to sections of an NTFS volume. We found that writing to the NTFS boot sector is possible with opening a handle to the volume using a user-mode application. Raw write access to the disk is possible with a kernel-mode driver by directly communicating with the storage class driver. Only one tested endpoint security solution was able to detect a program, that is used in the procedure of loading the driver to disable driver signature enforcement, based on heuristics. The writing method leads to limited bypassing of NTFS access lists and bypassing real-time write protection features of endpoint security.

Contents

1 Introduction

In a penetration testing assignment, red teams sometimes have to deal with files on NTFS drives that are non-writable or endpoint security solutions that have features such as on-access scanning and application control [12] [9]. The on-access scanning feature enables security solutions to scan a file for malicious content when it is written to the disk. The application control feature could limit applications to write to specific regions of the disk. These features could block writes of (malicious) data to the NTFS file system that may be required for red teams to compromise and control a system. An example of a case where non-writable files are encountered, would be where an infrastructure has to be infiltrated silently by modifying the Active Directory Database on a domain controller, adding a new domain administrator or user. This file is an example of a file that is locked and is not accessible with administrator privileges.

Windows uses built-in Application Programming Interfaces (API) to allow a program to interact with an NTFS file system [4][5]. `ReadFile` and `WriteFile` are examples of Windows API calls that allow a program to read a file or write a file respectively [6][7]. Those APIs call lower-level functions, eventually accessing the hard drive through the kernel, file system driver and storage driver [27].

Endpoint security monitors activity on an endpoint to detect, block or report malicious activity. It could for example hook into read and write API calls to monitor or block input/output (I/O) operations. Another example is to use a file system filter driver, which could be used for monitoring I/O operations and to log and prevent execution of the calls [2][3].

In our research we will focus on methods to write data to NTFS drives on a low level, bypassing hooks in read and write operations. Low level reading of data from NTFS drives has been done by Joseph Bialek in his script `Invoke-NinjaCopy` [1]. Writing data to NTFS on a low level could result in evasion of software hooks in windows read and write operations, evasion of the file system filter driver, evasion of the Windows access lists or bypassing a file lock [1][8]. Evasion of software hooks or evasion of the file system filter driver could lead to the possibility of writing (malicious) data to an NTFS drive, without endpoint security reporting this activity. Evasion of Windows access lists or bypassing a file lock could lead to overwriting data that is currently in use or overwriting data the user has no write permissions for.

This results in our main research question: *In what way can data be written to an NTFS file system, such that hooks in write operations within Windows are bypassed?*

2 Related work

Joseph Bialek describes in his script `Invoke-NinjaCopy` that by opening a read handle to an NTFS volume and parsing the NTFS structure, reading files with a file lock and bypassing Access List (ACL) read permissions is possible [1]. He states that he does not use the Windows API, which causes the Windows ACLs to be bypassed.

At the US Blackhat conference in 2016, Udi Yavo and Tomer Bitton described different hooking engines and also identified security issues on the Windows hooking methods, such as predictable R-X code stubs, which can be used to bypass hooks [30].

Kuba Gretzky discusses how the AV-hook of Bitdefender worked in 2016 on Breakdev.org [11]. When hooking into an API call the AV solution replaces the first few bytes of the function with a jump instruction that redirects the execution flow to its own hook handler. After handling the instruction, the original code will be continued. Kuba Gretzky also describes an attack where the AV-hook can be bypassed by calling the write code in ntdll.dll directly, by copying the code in his own application and directly call the syscall related to the write instruction. This only bypasses user-mode AV-protection.

In a blogpost of Cloudburst Security is described how shellcode in malware actively bypasses system-wide AV hooks [26]. This is done by overwriting the five byte function prolog of an API-call and replacing the return address with a push instruction followed by a return instruction. This bypasses AV hooking code that might reside in the first five bytes of the API-call. According to Kuba Gretzky from Breakdev.org, malware sandboxes such as Cuckoo can detect this [11].

Virus Bulletin posted an article in 2008 about the Rustock.C botnet describing the process of Rustock operating in user-mode to operating in kernel-mode by using system drivers [23]. A driver could bypass file system filter drivers and directly write to the NTFS volume using the `IRP_MJ_WRITE` IRP-request.

3 Methods

During this research we have set up a research environment, consisting of a physical machine for development and two virtual machines for testing. In order to gather information regarding NTFS, hooking and the Windows API, desk research and static analysis were performed. This is followed by write experiments to test whether it is possible to write to specific regions of an NTFS volume. In the last phase we tested what endpoint security solution does not detect the proposed methods.

3.1 Research environment

The research was conducted on a machine running Windows 10 Home Edition x64. A 64-bits edition was chosen as it also performs x86-x64 CPU instruction translation if an x86 application is running. In order to perform driver-related tests, two virtual machines were created running Windows 7 x64 in VMWare Workstation and Virtualbox. The specifications of both machines are shown in tables 1 and 2 respectively.

Operating system	Windows 10 Home Edition - x64
CPU	Intel Core i7-6700HQ @ 2.60GHz
Memory	16GB

Table 1: Specifications of Windows 10 machine

Operating system	Windows 7 Home Edition - x64
CPU	Intel Core i5-3570S @ 3.10GHz
Memory	3.5GB

Table 2: Specifications of Windows 7 machines

We have chosen to use Windows 7 and Windows 10, because, according to Statcounter [28], those two Operating Systems are used the most from all available Windows distributions. Windows 7's market share was 39.44%, while Windows 10's market share was 47.21% in May 2018 according to Statcounter. By choosing the most used Windows distributions, the chances are higher the research will be applicable to more systems worldwide.

3.2 Desk Research

Desk research was performed in several stages. In the first stage we gathered information about writing files to NTFS drives in Windows and the Windows API in general. Based on this information, we looked into ways to intercept, modify and forward the write file functions in user-mode and kernel-mode. In the second stage, more specific information was gathered based on the outcome of the static analysis described in section 3.3 about writing to raw NTFS drives, such as to what functions can be used to write to the disk. In the last stage `Invoke-Ninjacopy` by Joe Bialek was analysed to determine how he parses and approaches the NTFS drives. To define attack methods, more research was put into the NTFS file system, the Windows driver stack and the File system filter driver system.

3.3 Static analysis

In order to gather information about the read and write operations in Windows, static analysis was performed on DLL-files and executables that are related to Windows. This includes using Hex-Rays IDA v7.0 Free edition [13] to view the DLL-files and executables in disassembly and resolve function calls. The application is mainly used to discover what calls a write operation makes and gather information about calling lower-level functions that write data to the hard disk.

3.4 Write experiments

Based on the information we found about the Windows API and `Invoke-Ninjacopy`, we performed write experiments to the disk. These experiments included testing in user- and kernel-mode to determine if it is possible to open a volume or a physical drive and write directly to it. The experiments were performed on both VirtualBox and VMWare virtual machines to validate the results are not influenced by a vendor specific driver. It could be the case that VMWare or VirtualBox provides different drivers to access the storage. Since we are looking for a generic working writing methods, we wanted to verify whether it works for both types of storage drivers. In order to perform these experiments, we have created a user-mode application and a kernel-mode driver compatible with our Windows 7 machine that attempt different write operations to the disk and volume. The locations where we attempted to write to were the NTFS boot sector, NTFS file system space and the NTFS Master File Table. Based on the results we will determine whether it is possible or not to write directly to an NTFS volume and what software hooks are bypassed. The source codes used for the experiments can be found on GitHub[1].

3.4.1 Experiments

In experiment 1 we have built a user-mode driver and attempted to write to the NTFS volume. This experiment is based on the approach Joseph Bialek has taken with his script `Invoke-Ninjacopy`. Instead, we open a handle to the volume and the hard disk using `CreateFile` from the Windows API and write to the opened file handle using the `WriteFile` function.

In experiment 2 we have built a kernel-mode driver and attempted to write to the NTFS volume. This experiment performs the same Windows API operations as performed in experiment 1, but uses calls that require kernel-permissions: `NtCreateFile` and `NtWriteFile`.

In experiment 3 we have built a kernel-mode driver that is based on sectorio[2] and a user-mode application that communicates with this kernel-mode driver to write data directly to a sector on the disk. The kernel mode driver communicates directly to the storage class driver using an IRP to perform the `IRP_MJ_WRITE`-operation. Additionally the `SL_FORCE_DIRECT_WRITE` flag was set to force direct write access to the disk.

3.5 Verification

All found methods were checked whether they would work while an endpoint security solution is running. During this phase we explicitly checked whether endpoint security solutions were able to detect the methods and flag them as malicious. If the methods are flagged as malicious and blocked, the endpoint security solution is successful in preventing the methods from being executed.

[1] https://github.com/rickvg/low-level-ntfs

[2] https://github.com/jschicht/SectorIo

4 Desk Research

Desk research is divided into sections describing the results of desk research and static analysis as described in section 3. This section mainly includes background information that is required for interpreting the results of the experiments.

4.1 Windows API

In order to read or write files to the hard disk, Windows provides an API. A read operation can be performed by calling `ReadFile` and a write operation can be performed by calling `WriteFile`. In appendix A we show that if an application in user-mode calls `WriteFile` from the Windows API, multiple calls are made in user-mode and kernel-mode before the data is written to the hard disk. The user- and kernel-mode calls are displayed in figure 1.

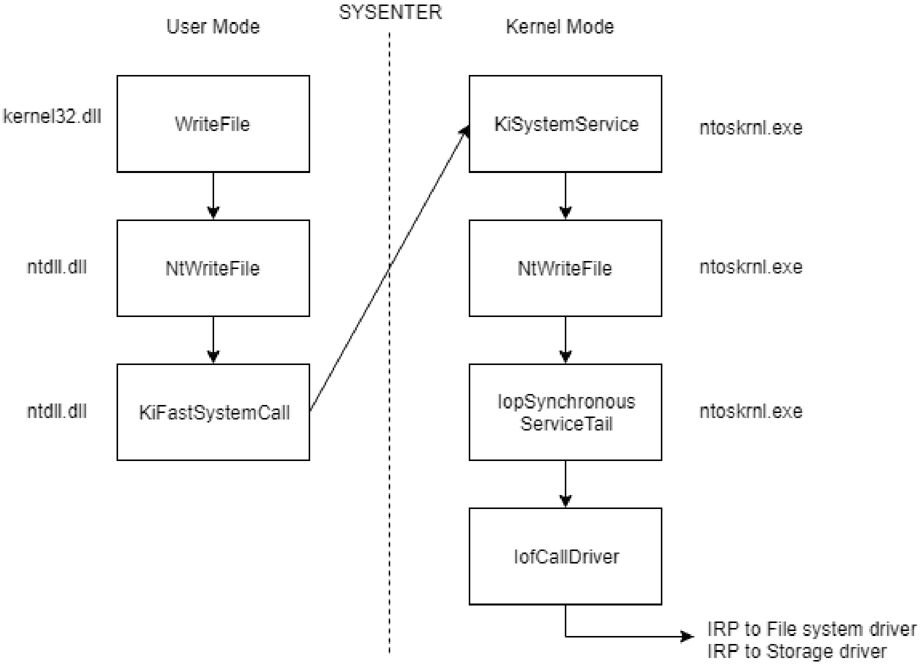


Figure 1: WriteFile related calls in user-mode and kernel-mode to the Hard Drive per DLL-file or Executable on Windows 10 x64

If an x86 application calls `WriteFile` on an x64-system, `KiFastSystemCall` in figure 1 is replaced with `Wow64SystemServiceCall` in order to translate the x86-instructions to x64-instructions. Another requirement is that `WriteFile` uses an open file handle, which could be created with API-call `CreateFile`.

A driver in kernel-mode is able to communicate directly with the file system driver or storage driver by sending an Input/Output Request Packet (IRP) through `IofCallDriver`. However, the Windows API follows the schedule depicted in figure 1 and an IRP is first forwarded to the file system driver. After processing the IRP, the file system driver sends an IRP with function `IRP_MJ_WRITE` to the storage class driver. This is an instruction for a write operation. The storage class driver is for example responsible for processing storage related IRPs, translating them to SCSI Request Blocks (SRBs) and forwarding them to the storage port driver corresponding to the given device object. Those SRBs are used to communicate with the storage port drivers. In figure 2 we show the storage driver stack, including the storage port drivers that communicate with specific types of storage, such as SCSI, SAS, IDE or SATA.

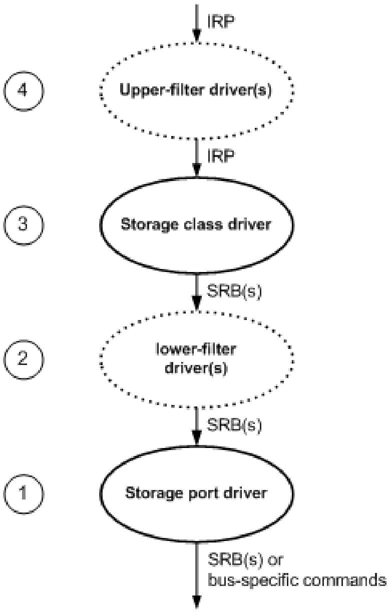


Figure 2: Windows Storage Driver Architecture (Microsoft 2017[3])

In figure 2 is shown that the storage class driver sends SRBs to the storage port drivers. This requires that the lower-filter drivers and storage port drivers must be able to interpret SCSI-commands and, if necessary, translate the command to a command compatible with IDE or SATA disks. In the figure, the upper-filter drivers are attached to the storage class driver and are able to interpret IRPs, while the lower-filter drivers must be able to interpret SCSI-commands.

4.2 Hooking

Hooking is a technique that is used to listen to or modify the behaviour of a program by intercepting an instruction of a program. According to Matt Hilman (2015), there are three hooking techniques that can be used in user-mode in Windows [14]. This is earlier described by Sameer Patil (2014), in an article where he describes user-mode and kernel-mode hooking techniques [22]. These techniques are described in sections 4.2.1 and 4.2.2.

4.2.1 User-mode hooking

Three techniques for user-mode hooking are Dynamic-Link Library injection (DLL), Import Address Table (IAT) hooking and inline hooking.

DLL-injection is used to provide hooking into function calls of another process [15]. By injecting the DLL, code is injected into the address space of the target process and therefore it is possible to manipulate the behaviour of the program. DLL-injection can be performed in four stages. The first stage includes attaching a process with access to the DLL to inject to a target process, using the `OpenProcess` function from the Windows API. In the second stage memory will be requested within the target process, which has the size of the DLL. The DLL

[3] https://docs.microsoft.com/en-us/windows-hardware/drivers/storage/storage-driver-architecture

is then copied to the created process memory space and executed by the Windows API-call `CreateRemoteThread`.

Executables in Windows could use an IAT, which contains the memory address of functions that are imported from libraries. By altering the memory address of a targeted function by the memory address of a hooking function, the call can be intercepted, altered or forwarded to the original target function. The memory address could be altered in the executable or via DLL-injection by altering the address in the process' memory. A downside of this technique is that it only works for load-time dynamic linking and not for run-time dynamic linking, because when run-time dynamic linking is used, the real memory addresses of DLL-functions are retrieved. In appendix B we show snippet of the import address table of `kernel32.dll` attached to a notepad process. The second memory address in the lines is the memory address of the function that is imported from `ntdll.dll`.

With inline hooking, the first few bytes of the function are overwritten by a jump instruction to a detour-function. The detour-function calls, after executing its instructions, a trampoline function, which is used to call the target function. The target function returns to the detour-function, which returns to the source function. Downside of using this technique is that the hooking function must perform the instructions that were overwritten in the source function. However, to allow inline hooking, Microsoft places a dummy-instruction at the start of the function (`MOV EDI, EDI`). This allows space for a short jump instruction to, for example, five padding bytes inserted by Microsoft before the instruction. These padding bytes could be used to perform a long jump to the hooking function.

4.2.2 Kernel-mode hooking

Four techniques for kernel-mode hooking are System Service Dispatch Table (SSDT) hooking, IRP hooking, Interrupt descriptor table (IDR) hooking and Sysenter hooking.

The SSDT contains mappings between function pointers and kernel routines. In order to hook into this function, the address of the call to the kernel routine should be changed to the address of the hooking function. This, however, does not work on 64-bit Windows versions as it runs Kernel Patch Protection, which prevents the kernel from being patched.

IRPs are used to communicate with device drivers and for intercommunication between drivers. Every drive must contain the `DriverEntry`-function, which creates a driver object for every device that handles I/O requests. The I/O manager, which is responsible for managing the I/O operations in Windows, creates a driver object at the same time. This driver object will be populated with addresses to the actual functions in the driver by the DriverEntry routine. Hooking can be performed by modifying the address of the driver's functions in the driver object.

The IDT contains pointers to Interrupt Service Routines (ISR) and is used to map interrupts and exceptions with responses. Replacing the address to the response could be replaced by the address of the hooking function, which allows the hook to be executed every time an interrupt was received. This, however, does not work on 64-bit Windows versions due to Kernel Patch Protection.

Sysenter allows user-mode programs to request services from the kernel. Sysenter is called in `ntdll.dll` and jumps to the address set in `SYSENTER_EIP`. By changing the address in register `SYSENTER_EIP`, the sysenter command could be forwarded to a detour-function.

4.3 File system minifilter drivers

The minifilter drivers are used to either log, monitor, modify or prevent I/O operations that are related to the file system. Microsoft Windows provides an I/O manager, which forwards I/O requests to the file system [3]. Windows also has a built-in filter manager that operates as a kernel-mode driver and attaches to the file system stack if filter drivers are active. This allows the filter manager to intercept I/O requests forwarded to the file system driver and pass them through the minifilter drivers. The process is shown in figure 3.

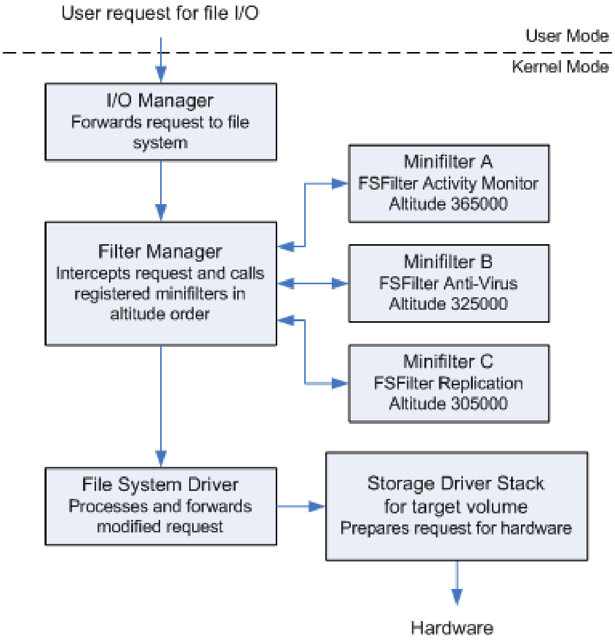


Figure 3: Simplified I/O Stack with filter manager en three minifilter drivers (Microsoft, 2017[4])

In figure 3 minifilters are shown in different categories. We observe that filter category Anti-Virus (AV) has a higher altitude than category Replication. The AV minifilter has a higher priority than the Replication filter, which means that drivers in the AV category will process the I/O request before drivers in the Replication category process it. This mechanism is for example used to prevent malicious data from being replicated as the AV minifilter could block the I/O operation when it detects malicious activity.

4.3.1 Altitudes

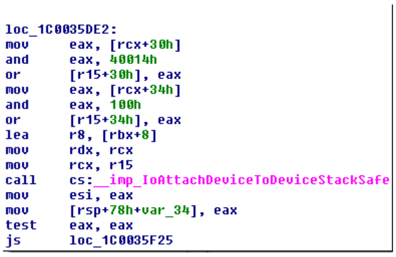
Microsoft provides a list with allocated altitudes to existing file system minifilter drivers [16]. Under column company, we observed that multiple companies that create endpoint security or AV software have registered an altitude for minifilter drivers. Those minifilter drivers are registered in the higher range of altitudes, allowing the minifilter driver to be processed earlier than lower altitudes. As of now the highest altitude not registered by Microsoft is registered by Raytheon Cyber Solutions, an American defence and technology company that includes endpoint security in their services.

[4] https://docs.microsoft.com/en-us/windows-hardware/drivers/ifs/filter-manager-concepts

4.3.2 Filter manager

The filter manager maintains device objects, which are in order attached to the file system. A device object is also known as a frame and represents a range of altitudes. If a minifilter registers to the filter manager, the altitude specified in the minifilters' INF-file is retrieved. If the specified altitude is higher than the frame altitude the frame was extended to, the frame will be extended to the defined altitude. If a legacy filter is active, the altitude of the legacy filter is determined using `FltpCalculateLegacyFilterAltitude` and the frame altitude can be extended until the legacy filter's altitude. If the specified altitude is higher than the extended frame a new set of device objects is added to each stack.

A device object is attached by the filter manager through function `FltpAttachDeviceObject`. In figure 4 we show a part of this function that was disassembled from `fltmgr.sys` in IDA Free edition. In this function `IoAttachDeviceToDeviceStackSafe` is called, which attaches an input device object to a given target device object [19].



```
loc_1C0035DE2:
mov     eax, [rcx+30h]
and     eax, 40014h
or      [r15+30h], eax
mov     eax, [rcx+34h]
and     eax, 100h
or      [r15+34h], eax
lea     r8, [rbx+8]
mov     rdx, rcx
mov     rcx, r15
call    cs:__imp_IoAttachDeviceToDeviceStackSafe
mov     esi, eax
mov     [rsp+78h+var_34], eax
test    eax, eax
js      loc_1C0035F25
```

Figure 4: Call from filter manager to attachdevice object to attach the input device object to a target device object

4.4 File caching

Windows has a method for caching files when they are in use or have recently been used [21]. The files will be stored in the system memory, also known as the system file cache. If a file is in the cache, which is the case when a handle is opened to the file and memory availability equals the size of the file, all write operations and read operations will be performed to and from the system file cache. A locked file, such as the active directory database on the domain controller, is in use by the domain controller and is being actively read and written to. Therefore there is a chance that the active directory database resides in the cache, when the the domain controller is turned on. Because of this, the data the attacker wrote to the file on the NTFS volume could be overwritten by the cached data. Also, the data is read only from disk if the file was removed from the system file cache.

4.4.1 Removing a file from the system file cache

A file can be removed from the system file cache using three methods. The system file cache can be flushed by using the undocumented `NtSetSystemInformation`-function from the Windows API [17]. This operation will temporarily slow down the Operating System, since for all I/O operations the disk has to be queried again, instead of the faster file cache. An implementation of this API-call is located on the GitHub page of Eric Grange[5].

The second method is to decrease the maximum size of the system file cache and enable enforcing the maximum size using function `NtSetSystemInformation`.

[5] https://github.com/EricGrange/FlushFileCache/blob/master/FlushFileCache.dpr

Since the maximum size of the system file cache decreases, older unused files will be removed from the system file cache to make space for new files. This method however, slows down the Operating System since for some I/O operations the disk has to be queried, instead of the file cache. An implementation of this method is created by Mark Russinovich and is called CacheSet[6].

The third method is to allocate empty memory to a process, until the memory is full. This results in a smaller system file cache size and may eventually remove the file from the cache. This method slows down the Operating System drastically, since 100 percent of the memory is in use. An implementation of this method is located at a blog of Chad Austin[7].

[6] https://docs.microsoft.com/en-us/sysinternals/downloads/cacheset

[7] https://chadaustin.me/2009/04/flushing-disk-cache/

5 Experiment Results

The experiment results are divided into sections describing results of the write experiments and methods to bypass NTFS Access lists and endpoint security.

5.1 Write experiments

Joseph Bialek describes in his script `Invoke-NinjaCopy` that he used a read handle to the whole NTFS volume [1]. He parsed the input with an NTFS parser, such that he is able to copy a selected file where he has no read access to. Based on this information we have performed some experiments by attempting to use `WriteFile` after opening a handle to the volume and disk to check whether the destination is writable or not.

5.1.1 User-mode

We have built a user-mode program that attempts to open a handle to the volume and disk using `CreateFile` from the Windows API. After opening the handle, the program attempts to write to specific sections of the NTFS volume or disk using `WriteFile`. The results are shown in table 3.

	Handle to volume	Handle to harddisk
Write to file system	X	X
Write to NTFS boot sector	V	X
Write to Master File Table	X	X
Alert by Endpoint security	X	V

Table 3: Results of write experiments sorted by writing to volume and harddisk

In our first experiment we opened a write handle to volume `C:` and attempted to write `RICK` to the first four bytes of the NTFS boot sector. In appendix C we show the initial 1024 bytes of the NTFS boot sector, followed by information whether the volume is mounted and locked. This is followed by the 1024 bytes of the boot sector after performing the writing operation. In this experiment, writing to the boot sector succeeded as the first four bytes have been overwritten by `RICK`.

In our second experiment we opened a write handle to volume `C:` and attempted to write `RICK` to the first four bytes of the Master File Table (MFT). Using `ntfsinfo64` by Sysinternals we determined the location of the MFT (`0xC0000000`), which contains pointers to files attached to file IDs. In appendix C we show MFT-data in the same format as our previous experiment. In this experiment, writing to the MFT-section failed as the first four bytes have not been overwritten.

In our third experiment we opened a write handle to disk `PhysicalDisk0`. When running the program, opening the read handle was detected by the AV-solution and classified as malicious activity. However, after turning off the AV-solution, we still were unable to write to the device. This is shown in appendix C, which shows the first 1024 of the NTFS boot sector on offset `0x23700000` from the start of the disk before and after performing the write operation.

Microsoft describes that since Windows Vista it is required to have exclusive access to the volume or physical disk in order to write to it. This can be obtained by locking or dismounting the volume with `DeviceIoControl` [7]. However, writing to the boot sector is allowed when opening a handle to the volume. Since we performed the experiments on an active volume it is not possible to dismount or lock the volume, thus we were unable to write to the Master File Table or file system space. To write to the volume, the application communicates through the file system driver to get access to the volume. To write to the disk, the application communicates with the storage drivers to get access to the disk and the request is not processed by the file system driver. At the file system level driver, access to the NTFS boot sector is allowed and this is passed to the storage driver. At the storage driver, writing to volume-related space is not allowed, unless the file system level driver communicates this by setting the flag `SL_FORCE_DIRECT_WRITE` in the IRP [20].

5.1.2 Kernel-mode

The same experiments were performed with a kernel mode driver. As the driver operates in a shared kernel space, the driver should have elevated permissions. The results are shown in table 4.

	Handle to volume	Handle to harddisk
Write to file system	X	X
Write to NTFS boot sector	V	X
Write to Master File Table	X	X
Alert by Endpoint security	X	X

Table 4: Results of write experiments sorted by writing to volume and harddisk

We observed in the results in table 4 that we did not get an alert of the endpoint security when opening a write handle to the harddisk. The write permissions for kernel-mode and user-mode using the Windows API appear to be the same.

Microsoft describes that since Windows Vista it is required to have exclusive access to the volume or physical disk in order to write to it. This can be obtained by locking or dismounting the volume with `DeviceIoControl` [7].

However, Microsoft also describes a flag that can be set in an IRP stack pointer to force a direct write to the volume: `SL_FORCE_DIRECT_WRITE` [20]. Therefore, we have written an additional kernel mode driver, mainly based on already existing code published on the code project [10]. An additional user-mode application communicates with this driver through Device IO control commands and sends data through this interface. The kernel mode driver communicates directly with the storage class driver, which is responsible for building SCSI Building Blocks based on the input IRP and forward it to the appropriate driver. The results of this experiment are shown in table 5. Since the kernel mode driver does not interact with the file system driver, it was only possible to create a handle to the harddisk.

	IDE	SATA	SCSI	SAS
Write to file system	X	X	V	V
Write to NTFS boot sector	X	X	V	V
Write to Master File Table	X	X	V	V
Alert by Endpoint security	X	X	X	X

Table 5: Results of write experiments while directly communicating with the storage class driver, sorted by storage technology

Our results in table 5 show that it is possible to directly write to an active NTFS volume using the kernel mode driver we have created. This, however, is limited to the SCSI and SAS technologies. This allows data on the NTFS volume to be overwritten. When attempting to write to IDE and SATA drives using this proof of concept, the response from the storage port driver to the class driver is `INVALID_SCSI_BLOCK_REQUEST`, which means that there is no proper translation from the sent IRP to a command in an SRB that the targeted storage technology understands. A possible solution is to write a kernel mode driver that communicates with the storage port drivers directly by using SRBs with the commands that match the storage technologies.

5.2 Writing to files with raw disk access

In tables 3, 4 and 5 we have shown to what locations we can write under what circumstances. From user-mode it is only possible to write to the NTFS boot sector, while from kernel mode raw write access to the disk is possible, allowing the complete NTFS volume to be overwritten. In order to write to files on an NTFS volume, the location of the NTFS volume on the raw disk must be known. Information about an NTFS volume, such as the cluster size, sector size and a pointer to the Master File Table can be found in the boot sector. The NTFS boot sector can be found by looking for bytes `EB 52 90 4E 54 46 53` on the raw disk [25].

5.2.1 Parsing the boot sector

The boot sector's global structure is shown in table 6.

Offset (hex)	Length	Value
0x0	3 bytes	JMP Instruction
0x3	8 bytes	OEM ID
0xB	25 bytes	BPB
0x24	48 bytes	Extended BPB
0x54	426 bytes	Bootstrap code
0x1FE	2 bytes	End of sector marker

Table 6: Boot sector structure

The Bios Parameter Block (BPB)-section in the boot sector contains information about the amount of sectors, clusters and the location of the Master File Table (MFT). In order to write to files and determine their location, the relevant data and their offset within the BPB are listed in table 7.

Offset (hex)	Length	Value
0x0B	2 bytes	Bytes per sector
0x0D	1 byte	Sectors per cluster
0x28	8 bytes	Total amount of sectors
0x30	8 bytes	Logical cluster number of MFT
0x38	8 bytes	Logical cluster number copy MFT
0x40	1 byte	Clusters per MFT record
0x44	1 byte	Clusters per index buffer

Table 7: Relevant sector and cluster information and MFT location information offsets within NTFS bootsector

This information can be used to calculate the byte location of the MFT from the start of the NTFS volume:

$$ByteLoc_{MFT} = S_{bytes} * C_{sectors} * MFT_{clusterloc}$$

$ByteLoc_{MFT}$ is the location of the MFT in bytes, S_{bytes} is the amount of bytes in the sector, $C_{sectors}$ is the amount of sectors in a cluster and $MFT_{clusterloc}$ is the location in logical clusters of the MFT.

5.2.2 Parsing the Master File Table

The MFT contains file-records that describe information about a file, such as attributes describing the name of the file, the creation time of the file and a pointer to the actual data of the file. This pointer could be used to locate the actual data on the volume and overwrite the content [24].

The MFT starts with a header attribute, which contains for example the file identifier, flags to determine whether the file exists or not and the physical record size of the MFT-record. These examples and their offsets within the MFT-record are shown in table 8. The contents of this table are derived from an MFT-record template for 010 editor by Andrea Barberio[8].

Offset (hex)	Length	Value
0x0	4 bytes	FILE, if invalid BAAD
0x14	2 bytes	Attribute offset
0x16	2 bytes	00 00 = Deleted 01 00 = Allocated 02 00 = Directory deleted 03 00 = Directory allocated
0x18	4 bytes	Actual record size
0x1C	4 bytes	Physical record size

Table 8: Structure of an MFT File-record header

NTFS uses attributes to store file-related data, such as the filename and the data of the file [24]. The attribute types, which are described in the first four bytes of an attribute, should match hexadecimal values `30 00 00 00` or `80 00 00 00`. Those values identify the file name and data attributes respectively. An attribute's content can either be resident or non-resident. The differences between the resident header and non-resident header are shown in tables 9 and 10.

Offset (hex)	Length	Value
0x10	4 bytes	Length of content
0x14	2 bytes	Offset to content
0x16	Defined by length of content	Content

Table 9: NTFS attribute resident header specific fields

Offset (hex)	Length	Value
0x10	8 bytes	Start of VCN Runlist
0x18	8 bytes	End of VCN Runlist
0x20	2 bytes	Offset to runlist

Table 10: NTFS attribute non-resident header specific fields

If the attribute's content is resident, the content resides within the MFT-record. The location and length of this content are defined by the offset and

[8] https://www.sweetscape.com/010editor/templates/files/MFTRecord.bt

length values shown in table 9. If the attribute's content is non-resident, the data is located outside of the MFT-record. The location is defined using a data runlist. The structure of a runlist is shown in table 11.

Offset (hex)	Length	Value
0x0	1 byte	Header
0x1	Defined by first 4 bits of header	Cluster count
Unknown	Defined by last 4 bits of header	LCN Offset

Table 11: NTFS Data run structure repeated x times

The purpose of a data runlist is to identify the location of (a fragment of) data that is related to the MFT-record. The first hexadecimal digit in the header defines the length of the cluster count field and the second hexadecimal digit in the header defines the length of the Logical Cluster Number (LCN) offset field. If the header is set to `00`, the end of a data runlist is indicated.

5.2.3 Overwriting file data

If, using the data runs, the cluster numbers on the volume are identified where the data is located. The data at that location can be overwritten. If the data is fragmented, the clusters that contain the data relevant for the file must be written to.

Using this method to overwrite file data bypasses the NTFS permissions set on a file as we do not communicate with the file system driver, which is responsible for maintaining those permissions. Next to that, since we have raw disk access, it is possible to write to files that are locked on the NTFS volume, such as NTDS.dit. This file is the Active Directory database and resides on the domain controller.

5.3 Bypassing Endpoint Security I/O hooks

In section 4.3 we have described that multiple endpoint security solutions use minifilter drivers to intercept I/O operations. These drivers generally have a high altitude as the endpoint security solution has to detect malicious operations before data is written to or read from the NTFS file system.

Based on this information we have performed some more writing tests with endpoint security protection solutions installed on the Windows 7 machines. The experiments included to test whether endpoint security solutions would detect if a malicious file is written to the disk with the regular `WriteFile` function from the Windows API and our method using a kernel mode driver. The malicious file used is Love Letter, which is a virtual basic (VBS) based worm that used to spread itself through e-mail. According to virustotal, this malicious file is detected by the majority of endpoint security solutions[9].

In order to create a baseline, we have written the VBS-file to the user's desktop in user-mode. In our results this is marked as regular write. The endpoint security solution should detect this in order to test whether writing the malicious data to the disk using our methods works, without the endpoint security solution blocking the operation or showing alerts. Then we performed all steps required to use our described method to write (malicious) data to the NTFS volume. Loading the driver is an operation that requires administrative privileges and was done using the `devcon` tool from the Windows Driver Development Kit[10].

[9] https://www.virustotal.com//file/556700ac50ffa845e5de853498242ee5abb288eb5b8ae1ae12bfdb5746e3b7b1/detection

[10] https://docs.microsoft.com/en-us/windows-hardware/drivers/download-the-wdk

In 64-bit editions of Windows, driver signature enforcement is enabled, which disallows unsigned drivers to be loaded. In order to disable this enforcement measure, we have used DSEFix[11] that uses a vulnerability in an already signed driver to get access to the kernel memory. We tested whether the endpoint security solution would detect this based on heuristics and not based on signatures, since this can be easily changed by modifying the source code slightly and rebuilding the application.

We used our user-mode application that runs with administrative permissions and attempted to communicate with our kernel-mode driver that directly communicates with the storage class driver. Through this communication the data in the VBS-file was written to a file that exists on the NTFS volume.

The results of those write experiments are shown in table 12.

	1	2	3	4	5	6
Regular write	✓	✓	✓	✓	✓	✓
Loading driver	✗	✗	✗	✗	✗	✗
Communication with driver	✗	✗	✗	✗	✗	✗
DSEFix activity	✗	✗	✗	✗	✗	✓
Write from kernel driver to disk	✗	✗	✗	✗	✗	✗

Table 12: Detection of writing malicious code to the disk by endpoint Security solutions from user-mode and kernel-mode

1 = Kaspersky Internet Security
2 = Norton Security Deluxe
3 = McAfee Total Protection
4 = Bitdefender Internet Security
5 = Avira Free edition
6 = ESET Internet security

In table 12 is shown that all endpoint security solutions detect writing the file to the disk using the regular Windows APIs. Loading the driver, communication with the driver and write the malicious code to the disk are not detected by the endpoint security solution. One endpoint security solution manages to detect DSEFix activity based on heuristics.

5.3.1 Alternative unverified approach

One approach could be to create a minifilter driver that manually attaches itself to a filter manager frame at a specified altitud by calling `FltAttachVolumeAtAltitude`. The altitude set should be lower, approximately 20.000, which is lower than all allocated altitudes according to Microsoft [16]. According to Microsoft's documentation on filter manager routines for I/O requests, a minifilter is allowed to open a file handle, read from a file and write to a file [18]. In the documentation of `FltCreateFile`, a function to create a file handle, is described that it is possible to pass a parameter containing the instance where it should start from. The I/O request will then only pass the minifilter drivers below. Functions `FltReadFile` and `FltWriteFile` allow the same parameter. A minifilter driver at an altitude lower than all other minifilter drivers, will process an I/O operation last in the stack. So, it would be possible that on every I/O operation that passes the stack to the file system, data is written to the file system without Endpoint security detecting this at filter driver level.

[11] https://github.com/hfiref0x/DSEFix

The downside of this method however, would be that the NTFS access lists would not be bypassed, since we still would communicate with the file system driver. The file system driver is responsible for managing the file access on an NTFS volume based on the permissions a user has for a file.

6 Recommendations

Based on the findings in this research we have formulated some recommendations that could mitigate the risk that is followed by writing to locked files and bypassing real-time protection enabled endpoint security.

In order to mitigate the risk of writing malicious data to the disk without endpoint security detecting this, the solution should hook into a lower level driver than the file system driver to monitor and block the I/O operations based on content and destination. This would be possible by hooking into the `IRP_MJ_WRITE`-function of the storage-related drivers as described in section 4.2. Another possibility is to actively block loading of unsigned drivers, unless a user manually has given permission to allow loading of each driver.

Microsoft could block the `SL_FORCE_DIRECT_WRITE` flag in IRPs for third party drivers at file system and storage level drivers. Only drivers approved and signed by Microsoft should be able to use this flag.

Use at least file-based encryption in the file system to encrypt files and implement this in the software that uses these files. If an attacker wants to use the in this research described methods to write to a file, it is required to know the encryption key(s) to encrypt the data to be written. In this situation an attacker needs to read the complete file, decrypt the content using the key, change the content in buffer that has to be changed and encrypt the content again. The complete file must then be overwritten, unless the attacker wants to write its own content to it. In the last case, the attacker is required to encrypt the content using the key and write it to the appropriate data location on the NTFS volume. If the level of encryption is implemented in such a way that, when a user is logged on, the files on the raw disk are still encrypted, it is likely that the attacker needs the encryption key of the disk, filesystem or files. This makes it more difficult to overwrite files with data that is formatted correctly for its related software when decrypted.

7 Discussion

Since it is possible to have write access to the raw disk, while bypassing the file system filter driver, file system related checks are bypassed. Therefore, NTFS access lists can be bypassed using a kernel mode driver that sends self-built IRPs with the `SL_FORCE_DIRECT_WRITE` flag set to the storage class driver. However, in our proof of concept this is limited to SCSI- and SAS-controllers as the storage class driver is not assigned to our ATA and IDE drivers, resulting in an incorrect translation between IRPs and SRBs. This could possibly be resolved by building a kernel-mode driver that directly communicates with the storage port driver that is responsible for handling requests regarding specific storage technologies.

Real-time write-related protection features in the tested endpoint security can be bypassed, since we do not communicate with the file system driver. This allows us to write malicious code to the disk using our kernel-mode driver that directly accesses the storage driver, without the endpoint security detecting this. This is resolvable by hooking into the storage class driver and listen for `IRP_MJ_WRITE` commands.

In our writing method we present a kernel mode driver. 64-bit editions of Windows require the kernel mode driver to be signed. In order to check this, driver signature enforcement is enabled and checks whether the kernel mode driver was signed during loading. Tools, such as DSEFix[12], use a vulnerability in an already signed virtual box driver, where the kernel memory can be accessed. The driver

[12] https://github.com/hfiref0x/DSEFix

signature enforcement can be disabled in the kernel memory. It appears in section 5.3 that the majority of endpoint security solutions do not detect DSEFix or any of its activity. Loading the kernel mode driver and the activity of the user mode application with the kernel mode driver are also not flagged as malicious activity.

Loading the driver and communication with the driver is only possible with administrative privileges. This method of writing still poses a risk as it would allow an attacker to write to locked files where a user with administrative privileges does not have read or write access to. An example named in the introduction of this research is the Active Directory database `NTDS.dit`. A user with administrative privileges could turn endpoint security off, but in this research we have shown that the methods to load the driver, disable driver signature enforcement, communication with the driver and writing malicious data to a disk are not detected by most of the tested endpoint security solutions. Using the methods described in this paper, it would be possible to write malicious data to the disk without alerting a system administrator about turning the endpoint security off for example.

The system file cache might interfere with bypassing NTFS ACLs. For example when an attacker attempts to write to the active directory database and the database is in use, the file might be stored in the system file cache. This results in all read and write operations from the file system going directly to the system file cache and not to the disk. In order to prevent the system file cache from interfering, just before the write of the data the system file cache should be flushed. Our method always writes the new data directly to the disk. This in combination with the flush, would require the Operating System to perform a new read operation from the disk, since it can not read from the system file cache. This would allow the new data to be read and implemented on a live system. This theory however, still has to be verified.

When an attacker is able to access the kernel space, basically anything related to Windows can be bypassed or modified. This also includes the retrieval of file encryption keys that could be in memory. On 64-bit editions of Windows, a feature called patchguard is in place that detects whether the kernel memory was altered with, making it more difficult to change variables in the kernel. However, there are multiple methods to bypass patchguard described on the Github page of `hfiref0x`[13]. This research presents a way to bypass endpoint security, without modifying the endpoint security solution in kernel. Also, ways to bypass the NTFS permissions are presented to for example write to locked files while the volume is in use. As the method is as of now undetected, alerts to system administrators might be prevented allow a red team to infiltrate an infrastructure silently.

8 Conclusion

Our research question is: *In what way can data be written to an NTFS filesystem, such that hooks in write operations within Windows are bypassed?*

From user-mode it is possible to change the NTFS Master File Table location pointer in the boot sector, this could result in a bypass of the software hooks in case a copy of the Master File Table has been created that is accessible from user-space and the location pointer is changed to that new location. This is possible by opening a handle to the volume and writing to the boot sector.

From kernel-mode it is possible to have raw disk write access, bypassing the file system driver, by directly communicating to the storage class driver. Since the volume is maintained by the file system driver and we communicate directly to

[13] https://github.com/hfiref0x/UPGDSED

the storage class driver, volume write access is not relevant. In order to perform a raw write to the disk, a kernel mode driver must be created that builds its own IRPs with major function `IRP_MJ_WRITE` and sets the flag `SL_FORCE_DIRECT_WRITE` to allow writing directly to disk. The targeted driver must be set to the storage class driver `disk.sys`. The device object must be assigned by directly referring to the hard disk object that is created by the storage class driver. Lower level methods of writing are possible, but specific commands of the storage technology have to be learned and included in the SCSI Request Block to the storage port driver.

Both approaches require administrative privileges to run and result in bypassing real-time write protection features of the tested endpoint security and bypassing of NTFS ACLs. The NTFS ACL bypass is limited, since in-use files could be located in the system file cache. Altering those files on disk could lead to invalid files as the data in the system file cache could overwrite the data that was written directly to the disk using our method described in section 5.1. Only one of the tested endpoint security solutions could detect the write methods described in this research based on heuristics of `DSEFix`, but not based on the user-mode and kernel-mode driver activity that perform the actual write operation.

9 Future work

Future work might include to research lower level methods such as directly adressing the storage port driver with a kernel mode driver. This requires SRBs to be created and forwarded to the storage port drivers containing the commands that match the storage technology. A reference for SRB to ATA translation is created by T10 in 2004 [29].

As the writing methods that allow raw disk access require kernel mode access, a relevant research field would be to investigate other methods to load unsigned kernel drivers or change kernel memory variables related to driver signature enforcement by exploiting vulnerabilities in already signed drivers.

Another research that might be performed is to test other endpoint security solutions. Check with different sensitivity levels whether they are able to detect the activity and test this also for the researched lower level methods.

As it is possible to write data to the disk, while bypassing the NTFS file system drivers, it would be interesting to check whether it would be possible to bypass Windows audit log triggers, such as: On modifying data on location X, a log event is triggered and a log is sent to a central server.

The system file cache might interfere with the method we have described in our paper to write data to the raw NTFS volume. It requires research whether the methods described in section 4.4 would work to allow a modification of an in-use locked file on the disk.

References

[1] Joseph Bialek. *Invoke-NinjaCopy*. Oct. 2013. URL: `https://github.com/clymb3r/PowerShell/blob/master/Invoke-NinjaCopy/Invoke-NinjaCopy.ps1` (visited on 06/04/2018).

[2] Microsoft Hardware Dev Center. *Allocated Altitudes*. Apr. 2017. URL: `https://docs.microsoft.com/en-us/windows-hardware/drivers/ifs/allocated-altitudes` (visited on 06/05/2018).

[3] Microsoft Hardware Dev Center. *What Is a File System Filter Driver?* Apr. 2017. URL: `https://docs.microsoft.com/en-us/windows-hardware/drivers/ifs/what-is-a-file-system-filter-driver-` (visited on 06/04/2018).

[4] Windows Dev Center. *About Transactional NTFS*. May 2018. URL: `https://msdn.microsoft.com/en-us/library/windows/desktop/aa363764(v=vs.85).aspx` (visited on 06/04/2018).

[5] Windows Dev Center. *File Management Functions*. May 2018. URL: `https://msdn.microsoft.com/en-us/library/windows/desktop/aa364232(v=vs.85).aspx` (visited on 06/04/2018).

[6] Windows Dev Center. *ReadFile Function*. May 2018. URL: `https://msdn.microsoft.com/en-us/library/windows/desktop/aa365467(v=vs.85).aspx` (visited on 06/04/2018).

[7] Windows Dev Center. *WriteFile Function*. May 2018. URL: `https://msdn.microsoft.com/en-us/library/windows/desktop/aa365747(v=vs.85).aspx` (visited on 06/04/2018).

[8] clymb3r. *Using PowerShell to Copy NTDS.dit / Registry Hives, Bypass SACL's / DACL's / File Locks*. June 2013. URL: `https://clymb3r.wordpress.com/2013/06/13/using-powershell-to-copy-ntds-dit-registry-hives-bypass-sacls-dacls-file-locks/` (visited on 06/04/2018).

[9] Cylance. "Feature Focus CylancePROTECT". In: (Nov. 2017). URL: `https://www.cylance.com/content/dam/cylance/pdfs/feature-focus/Feature_Focus_PROTECT_App_Control.pdf` (visited on 06/29/2018).

[10] dkg0414. "Reading and Writing to Raw Disk Sectors". In: (Aug. 2008). URL: `https://www.codeproject.com/Articles/28314/Reading-and-Writing-to-Raw-Disk-Sectors` (visited on 06/24/2018).

[11] Kuba Gretzky. *Defeating Antivirus Real-time Protection From The Inside*. July 2016. URL: `https://breakdev.org/defeating-antivirus-real-time-protection-from-the-inside/` (visited on 06/05/2018).

[12] Symantec Bill Hayes. "Who Goes There? An Introduction to On-Access Virus Scanning, Part One". In: (Sept. 2002). URL: `https://www.symantec.com/connect/articles/who-goes-there-introduction-access-virus-scanning-part-one` (visited on 06/29/2018).

[13] Hex-Rays. *IDA 7.0 Freeware Download page*. 2017. URL: `https://www.hex-rays.com/products/ida/support/download_freeware.shtml` (visited on 06/08/2018).

[14] Matt Hillman. *Dynamic Hooking Techniques: User Mode*. Aug. 2015. URL: `https://www.mwrinfosecurity.com/our-thinking/dynamic-hooking-techniques-user-mode/` (visited on 06/11/2018).

[15] J.Berdajs and Z. Bosić. *Extending applications using an advanced approach to DLL injection and API hooking.* Software - Practice and Experience 40:567-584. Jan. 2010. URL: `http://blog.opensecurityresearch.com/2013/01/windows-dll-injection-basics.html` (visited on 06/24/2018).

[16] Microsoft. *Allocated Altitudes.* Apr. 2017. URL: `https://docs.microsoft.com/en-us/windows-hardware/drivers/ifs/allocated-altitudes` (visited on 06/14/2018).

[17] Microsoft. *Command line utility for purging Window's standby list.* 2018. URL: `https://gist.github.com/bitshifter/c87aa396446bbebeab29` (visited on 07/08/2018).

[18] Microsoft. *Filter Manager Routines for I/O Requests Generated by the Minifilter Driver.* Apr. 2017. URL: `https://docs.microsoft.com/en-us/windows-hardware/drivers/ifs/i-o-requests-generated-by-the-minifilter-driver` (visited on 06/19/2018).

[19] Microsoft. *IoAttachDeviceToDeviceStackSafe function.* Apr. 2018. URL: `https://docs.microsoft.com/en-us/windows-hardware/drivers/ddi/content/ntddk/nf-ntddk-ioattachdevicetodevicestacksafe` (visited on 06/19/2018).

[20] Microsoft. *IRP_MJ_WRITE.* Nov. 2017. URL: `https://docs.microsoft.com/en-us/windows-hardware/drivers/ifs/irp-mj-write` (visited on 06/24/2018).

[21] Microsoft. *Performance Tuning Cache and Memory Manager.* Oct. 2017. URL: `https://docs.microsoft.com/en-us/windows-server/administration/performance-tuning/subsystem/cache-memory-management/` (visited on 07/08/2018).

[22] Sameer Patil. *Code Injection and API Hooking Techniques.* Mar. 2014. URL: `http://nagareshwar.securityxploded.com/2014/03/20/code-injection-and-api-hooking-techniques/` (visited on 06/11/2018).

[23] Chandra Prakash. *Your filters are bypassed: Rustock.C in the kernel.* Sunbelt Software - USA. Nov. 2008. URL: `https://www.virusbulletin.com/virusbulletin/2008/11/your-filters-are-bypassed-rustock-c-kernel` (visited on 06/06/2018).

[24] Richard Russon and Yuval Fledel. "NTFS Documentation". In: (2018). URL: `http://inform.pucp.edu.pe/~inf232/Ntfs/ntfsdoc.pdf` (visited on 07/08/2018).

[25] Thomas Schwarz. *COEN 252 Computer Forensics - NTFS.* 2007. URL: `http://www.cse.scu.edu/~tschwarz/coen252_07Fall/Lectures/NTFS.html` (visited on 06/25/2018).

[26] Cloudburst Security. *Reverse Engineering for Malware: Shellcodes and AV/API Hook Evasion.* June 2016. URL: `https://www.cloudburstsecurity.com/2016/06/10/reverse-engineering-for-malware-shellcodes-and-avapi-hook-evasion/` (visited on 06/05/2018).

[27] Isso (a Stackoverflow user). *(windows) raw write to file without involving win32api.* June 2012. URL: `https://stackoverflow.com/questions/11229612/windows-raw-write-to-file-without-involving-win32api/11252104#11252104` (visited on 06/04/2018).

[28] Statcounter. *Desktop Windows Version Market Share Worldwide.* 2018. URL: `http://gs.statcounter.com/windows-version-market-share/desktop/worldwide/` (visited on 06/25/2018).

[29] T10. “SCSI to ATA Command Translations”. In: (2004). 04-136r0. T10 SCSI to ATA Translations Study Group. URL: http://t10.org/ftp/t10/document.04/04-136r0.pdf (visited on 06/29/2018).

[30] EnSilo Research Team. “Captain Hook - Pirating AVs to Bypass Exploit Mitigations”. In: (2016). Blackhat Conference USA 2016 - Presentation. URL: https://www.blackhat.com/docs/us-16/materials/us-16-Yavo-Captain-Hook-Pirating-AVs-To-Bypass-Exploit-Mitigations-wp.pdf (visited on 06/04/2018).

A IDA v7.0 Windows API analysis

Kernel32.dll: WriteFile function. Contains a reference to import from ntdll.dll.

```
.text:6B86E270 ; BOOL __stdcall WriteFile(HANDLE hFile, LPCVOID lpBuffer, DWORD nNumberOfBytesToWrite, LPDWORD lpNumberOfBytesWritten, LPOVERLAPPED lpOverlapped)
.text:6B86E270                public _WriteFile@20
.text:6B86E270 _WriteFile@20   proc near               ; DATA XREF: .rdata:off_6B880768^^Yo
.text:6B86E270
.text:6B86E270 hFile           = dword ptr  4
.text:6B86E270 lpBuffer        = dword ptr  8
.text:6B86E270 nNumberOfBytesToWrite= dword ptr  0Ch
.text:6B86E270 lpNumberOfBytesWritten= dword ptr  10h
.text:6B86E270 lpOverlapped    = dword ptr  14h
.text:6B86E270
.text:6B86E270                jmp     ds:__imp__WriteFile@20 ; WriteFile(x,x,x,x,x)
.text:6B86E270 _WriteFile@20   endp
```

Ntdll.dll: NtWriteFile function. Contains a Wow64SystemServiceCall to enter kernel-mode with a `sysenter`.

```
.text:4B2EE8B0 ; __stdcall NtWriteFile(x, x, x, x, x, x, x, x, x)
.text:4B2EE8B0                public _NtWriteFile@36
.text:4B2EE8B0 _NtWriteFile@36 proc near               ; CODE XREF: EtwpFlushBuffer(x,x,x)+12C^^Xp
.text:4B2EE8B0                                         ; EtwpFinalizeLogFileHeader(x,x)+1DD^^Xp ...
.text:4B2EE8B0                mov     eax, 1A0008h    ; NtWriteFile
.text:4B2EE8B5                mov     edx, offset _Wow64SystemServiceCall@0 ; Wow64SystemServiceCall()
.text:4B2EE8BA                call    edx ; Wow64SystemServiceCall() ; Wow64SystemServiceCall()
.text:4B2EE8BC                retn    24h
.text:4B2EE8BC _NtWriteFile@36 endp
```

Ntoskrnl.exe: KiSystemService will be the call handler and call NtWritefile. NtWriteFile then calls for IopSynchronousServiceTail.

```
PAGE:00000001404CCCF2                mov     r9b, 1
PAGE:00000001404CCCF5                mov     r8, rsi
PAGE:00000001404CCCF8                mov     rdx, rdi        ; Irp
PAGE:00000001404CCCFB                call    IopSynchronousServiceTail
PAGE:00000001404CCD00                jmp     loc_1404CCB8F
```

Ntoskrnl.exe: IopSynchronousServiceTail calls IofCallDriver.

```
PAGE:00000001404CB5B3 loc_1404CB5B3:                          ; CODE XREF: IopSynchronousServiceTail+181^^Xj
PAGE:00000001404CB5B3                                         ; IopSynchronousServiceTail+347^^Yj
```

```
        PAGE:00000001404CB5B3                mov     rdx, rbx          ; Irp
        PAGE:00000001404CB5B6                mov     rcx, r13          ;
         ↪  DeviceObject
        PAGE:00000001404CB5B9                call    IofCallDriver
        PAGE:00000001404CB5BE                mov     ebp, eax
```

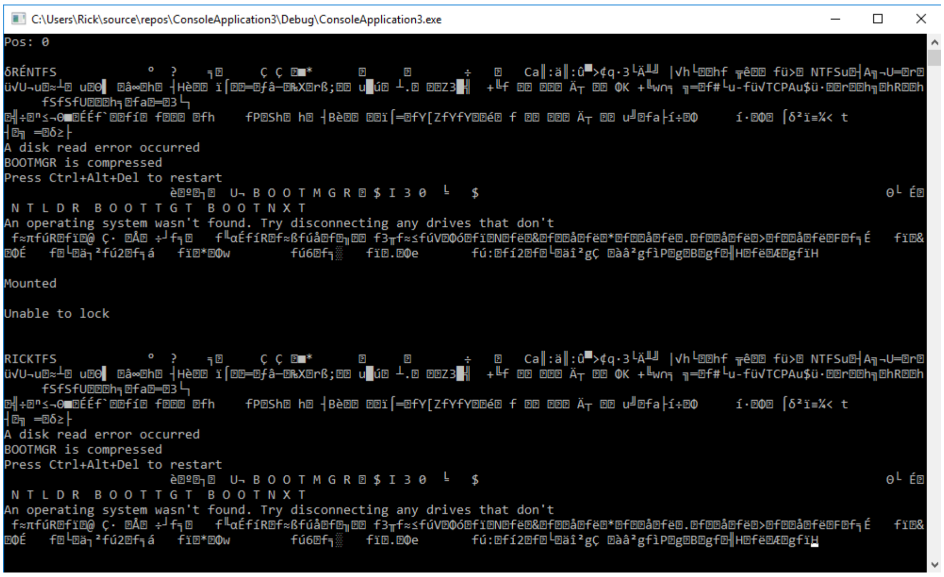
Ntoskrnl.exe: IofCallDriver sends IRP_MJ_WRITE input/output request packets to the drivers.

```
        PAGEVRFY:0000000140795600                dq offset aIrp_mj_write ;
         ↪  "IRP_MJ_WRITE"
```

B Import Address Table of kernel32.dll attached to process Notepad

```
        00007ffc`e1099700  00007ffc`e392bff0 ntdll!NtLockFile
        00007ffc`e1099708  00007ffc`e392d7b0 ntdll!NtUnlockFile
        00007ffc`e1099710  00007ffc`e392a280 ntdll!NtAllocateVirtualMemory
        00007ffc`e1099718  00007ffc`e392a040 ntdll!NtReadFile
        00007ffc`e1099720  00007ffc`e38e2560 ntdll!RtlIsTextUnicode
        00007ffc`e1099728  00007ffc`e392a080 ntdll!NtWriteFile
```

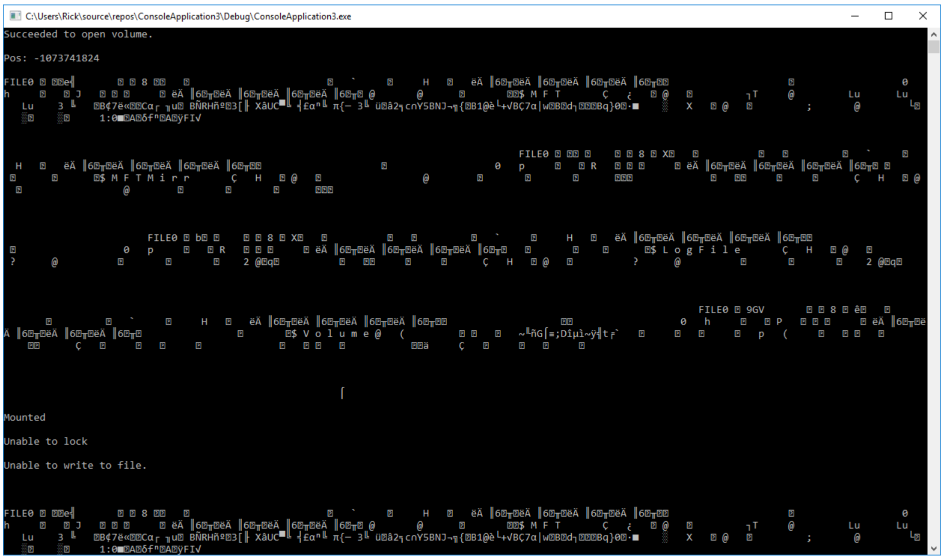
C NTFS write test results



C:\Users\Rick\source\repos\ConsoleApplication3\Debug\ConsoleApplication3.exe

```
Pos: 0

δRÉNTFS            °  ?    ¬      Ç Ç  *        ÷     Ca║:ä║:û▀>¢q·3└Ä╨╝ |√h└  hf ╥ê   fü>  NTFSu ┤A╗¬U=  r
ü√U¬u ≈┴  u Θ█   â∞  h  ┤Hè   ï⌠  =  ƒâ─ ‰X rß;   u█ú  ┴.   Z3█╢  +╙f       Ä┬    ΦK +╙wn╕ ╗=  f#└u-fü√TCPAu$ü·  r  h╕ hR  h
     fSfSfU   h╕ fa =  3└╕
 ╣÷ ⁿ≤¬Θ▄ ÉÉf`  fí  f    fh    fP Sh  h  ┤Bè     ï⌠= fY[ZfYfY  é  f         Ä┬    u╝ fa├í÷ Φ     í· Φ  ⌠δ²ï≡¼< t
┤ ╗ = δ≥├
A disk read error occurred
BOOTMGR is compressed
Press Ctrl+Alt+Del to restart
                      è º ╕   U¬ B O O T M G R   $ I 3 0  ╘   $                                                     Θ└ É
 N T L D R  B O O T T G T  B O O T N X T
An operating system wasn't found. Try disconnecting any drives that don't
 f≈πfúR fï @ Ç·  Å  ÷┘f╕     f╙αÉfíR f≈ßfúå f ╗   f3╥f≈≤fúV Φó fï N fë &  f  å fë *  f  å fë . f  å fë > f  å fë F f╕É    fï &
 ΦÉ   f └ ä╕²fú2 f╕á   fï * Φw        fú6 f╕░   fï . Φe       fú: fí2 f └ äî²gÇ  àâ²gfìP g B gf ║H fë Æ gfïH

Mounted

Unable to lock


RICKTFS            °  ?    ¬      Ç Ç  *        ÷     Ca║:ä║:û▀>¢q·3└Ä╨╝ |√h└  hf ╥ê   fü>  NTFSu ┤A╗¬U=  r
ü√U¬u ≈┴  u Θ█   â∞  h  ┤Hè   ï⌠  =  ƒâ─ ‰X rß;   u█ú  ┴.   Z3█╢  +╙f       Ä┬    ΦK +╙wn╕ ╗=  f#└u-fü√TCPAu$ü·  r  h╕ hR  h
     fSfSfU   h╕ fa =  3└╕
 ╣÷ ⁿ≤¬Θ▄ ÉÉf`  fí  f    fh    fP Sh  h  ┤Bè     ï⌠= fY[ZfYfY  é  f         Ä┬    u╝ fa├í÷ Φ     í· Φ  ⌠δ²ï≡¼< t
┤ ╗ = δ≥├
A disk read error occurred
BOOTMGR is compressed
Press Ctrl+Alt+Del to restart
                      è º ╕   U¬ B O O T M G R   $ I 3 0  ╘   $                                                     Θ└ É
 N T L D R  B O O T T G T  B O O T N X T
An operating system wasn't found. Try disconnecting any drives that don't
 f≈πfúR fï @ Ç·  Å  ÷┘f╕     f╙αÉfíR f≈ßfúå f ╗   f3╥f≈≤fúV Φó fï N fë &  f  å fë *  f  å fë . f  å fë > f  å fë F f╕É    fï &
 ΦÉ   f └ ä╕²fú2 f╕á   fï * Φw        fú6 f╕░   fï . Φe       fú: fí2 f └ äî²gÇ  àâ²gfìP g B gf ║H fë Æ gfïH
```

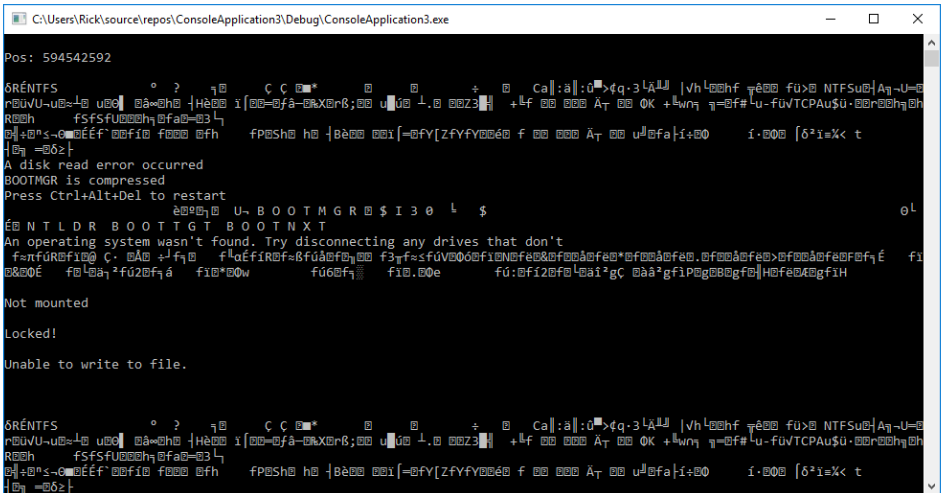
Figure A1: Succesful write to NTFS boot sector with WriteFile



C:\Users\Rick\source\repos\ConsoleApplication3\Debug\ConsoleApplication3.exe

```
Succeeded to open volume.

Pos: -1073741824

FILE0 ⍰ ⍰⍰e╡       ⍰ ⍰ 8 ⍰⍰   ⍰                   ⍰   `     ⍰     H   ⍰   ëÄ ║6⍰╥⍰ëÄ ║6⍰╥⍰ëÄ ║6⍰╥⍰ëÄ ║6⍰╥⍰⍰                 ⍰            0
h     ⍰   ⍰ J   ⍰ ⍰ ⍰      ⍰ ëÄ ║6⍰╥⍰ëÄ ║6⍰╥⍰ëÄ ║6⍰╥⍰ëÄ ║6⍰╥⍰ @       @       ⍰       ⍰⍰$ M F T       Ç   ¿   ⍰ @   ⍰         ┐T     @         Lu      Lu
   Lu    3 ╙   ⍰B¢7ë«⍰⍰Cα┌ ╗u⍰ BÑRHñº⍰3[╟ XâUC▀╙ ╡£αⁿ╙ π{- 3╙ ü⍰â2┐c∩Y5BNJ¬╗{⍰B1@è└+√BÇ7α|w⍰B⍰d┐⍰⍰⍰Bq}0⍰·■     ░    X   ⍰ @   ⍰          ;       @        └⍰
   ░⍰    ░⍰     1:0■⍰A⍰δfⁿ⍰A⍰ÿFI√



 H   ⍰   ëÄ ║6⍰╥⍰ëÄ ║6⍰╥⍰ëÄ ║6⍰╥⍰ëÄ ║6⍰╥⍰⍰                   ⍰                  0   p     ⍰   ⍰ R   ⍰ ⍰ ⍰      ⍰ ëÄ ║6⍰╥⍰ëÄ ║6⍰╥⍰ëÄ ║6⍰╥⍰ëÄ ║6⍰╥⍰ ⍰
 ⍰      ⍰      ⍰$ M F T M i r r       Ç   H   ⍰ @   ⍰              @        ⍰       ⍰       ⍰       ⍰⍰⍰              ⍰   ⍰⍰     ⍰      ⍰      Ç   H   ⍰ @
 ⍰                @        ⍰       ⍰       ⍰      ⍰⍰⍰

                             FILE0 ⍰ ⍰⍰ ⍰    ⍰ ⍰ 8 ⍰ X⍰   ⍰                   ⍰   `     ⍰     H   ⍰   ëÄ ║6⍰╥⍰ëÄ ║6⍰╥⍰ëÄ ║6⍰╥⍰ëÄ ║6⍰╥⍰⍰
 ⍰              0   p     ⍰   ⍰ R   ⍰ ⍰ ⍰      ⍰ ëÄ ║6⍰╥⍰ëÄ ║6⍰╥⍰ëÄ ║6⍰╥⍰ëÄ ║6⍰╥⍰   ⍰       ⍰     ⍰      ⍰$ L o g F i l e      Ç   H   ⍰ @   ⍰
 ?      @        ⍰       ⍰       ⍰   2 @⍰q⍰        ⍰   ⍰⍰     ⍰      ⍰      Ç   H   ⍰ @   ⍰          ?       @           ⍰        ⍰        ⍰    2 @⍰q⍰



     ⍰          ⍰   `     ⍰     H   ⍰   ëÄ ║6⍰╥⍰ëÄ ║6⍰╥⍰ëÄ ║6⍰╥⍰ëÄ ║6⍰╥⍰⍰                                     FILE0 ⍰ 9GV      ⍰ ⍰ 8 ⍰ ê⍰   ⍰
Ä ║6⍰╥⍰ëÄ ║6⍰╥⍰ëÄ ║6⍰╥⍰              ⍰        ⍰$ V o l u m e @   (          ⍰⍰                         0   h     ⍰   ⍰ P   ⍰ ⍰ ⍰      ⍰ ëÄ ║6⍰╥⍰ë
   ⍰⍰      Ç   ⍰     ⍰   ⍰      ⍰             ⍰   ⍰ ⍰   ⍰          ⍰⍰ä     Ç   ⍰     ⍰ ⍰   ⍰   ~╙ñG⌠≡;Dîµì~ÿ╡t╒`   ⍰      ⍰    ⍰      ⍰   p   (     ⍰   ⍰ ⍰   ⍰
                                                                                Ç   ⍰      ⍰   ⍰      ⍰

                                                         ⌠

Mounted

Unable to lock

Unable to write to file.


FILE0 ⍰ ⍰⍰e╡       ⍰ ⍰ 8 ⍰⍰   ⍰                   ⍰   `     ⍰     H   ⍰   ëÄ ║6⍰╥⍰ëÄ ║6⍰╥⍰ëÄ ║6⍰╥⍰ëÄ ║6⍰╥⍰⍰                 ⍰            0
h     ⍰   ⍰ J   ⍰ ⍰ ⍰      ⍰ ëÄ ║6⍰╥⍰ëÄ ║6⍰╥⍰ëÄ ║6⍰╥⍰ëÄ ║6⍰╥⍰ @       @       ⍰       ⍰⍰$ M F T       Ç   ¿   ⍰ @   ⍰         ┐T     @         Lu      Lu
   Lu    3 ╙   ⍰B¢7ë«⍰⍰Cα┌ ╗u⍰ BÑRHñº⍰3[╟ XâUC▀╙ ╡£αⁿ╙ π{- 3╙ ü⍰â2┐c∩Y5BNJ¬╗{⍰B1@è└+√BÇ7α|w⍰B⍰d┐⍰⍰⍰Bq}0⍰·■     ░    X   ⍰ @   ⍰          ;       @        └⍰
   ░⍰    ░⍰     1:0■⍰A⍰δfⁿ⍰A⍰ÿFI√
```

Figure A2: Failed write attempt to NTFS Master File Table with WriteFile



C:\Users\Rick\source\repos\ConsoleApplication3\Debug\ConsoleApplication3.exe

```
Pos: 594542592

δRÉNTFS           °  ?    ┐⍰     Ç Ç ⍰■*      ⍰     ⍰        ÷   ⍰   Ca║:ä║:û▀>¢q·3└Ä╨╛ |√h└⍰⍰hf ╤ê⍰⍰ fü>⍰ NTFSu⍰┤A╗¬U=⍰
r⍰ü√U¬u⍰≈┴⍰ u⍰Θ▌ ⍰â∞⍰h⍰ ┤Hè⍰⍰ ï⌠⍰⍰=⍰ƒâ─⍰‰X⍰rß;⍰⍰ u█ú⍰ ┴.⍰ ⍰⍰Z3█╢  +╙f ⍰⍰ ⍰⍰⍰ Ä┬ ⍰⍰ ΦK +╙w∩┐ ╕=⍰f#└u-fü√TCPAu$ü·⍰⍰r⍰⍰h╕⍰h
R⍰⍰h     fSfSfU⍰⍰⍰h┐⍰fa⍰=⍰3└┐
⍰╢÷⍰ⁿ≤¬Θ■⍰ÉÉf`⍰⍰fí⍰ f⍰⍰⍰ ⍰fh    fP⍰Sh⍰ h⍰ ┤Bè⍰⍰ ⍰⍰ï⌠=⍰fY[ZfYfY⍰⍰é⍰ f ⍰⍰ ⍰⍰⍰ Ä┬ ⍰⍰ u╛⍰fa├í÷⍰Φ     í·⍰Φ⍰ ⌠δ²ï≡¼< t
┤⍰╗ =⍰δ≥├
A disk read error occurred
BOOTMGR is compressed
Press Ctrl+Alt+Del to restart
                          è⍰º⍰┐⍰  U¬ B O O T M G R ⍰ $ I 3 0  ╘   $                                                                      Θ└
É⍰ N T L D R  B O O T T G T  B O O T N X T
An operating system wasn't found. Try disconnecting any drives that don't
 f≈πfúR⍰fï⍰@ Ç· ⍰Å⍰ ÷┘f┐⍰   f╙αÉfíR⍰f≈ßfúå⍰f⍰╗⍰⍰ f3╥f≈≤fúV⍰Φó⍰fï⍰N⍰fë⍰&⍰f⍰⍰å⍰fë⍰*⍰f⍰⍰å⍰fë⍰.⍰f⍰⍰å⍰fë⍰>⍰f⍰⍰å⍰fë⍰F⍰f┐É   fï
⍰&⍰ΦÉ   f⍰└⍰ä┐²fú2⍰f┐á   fï⍰*⍰Φw        fú6⍰f┐░   fï⍰.⍰Φe       fú:⍰fí2⍰f⍰└⍰äî²gÇ ⍰àâ²gfìP⍰g⍰B⍰gf⍰╢H⍰fë⍰Æ⍰gfïH

Not mounted

Locked!

Unable to write to file.


δRÉNTFS           °  ?    ┐⍰     Ç Ç ⍰■*      ⍰     ⍰        ÷   ⍰   Ca║:ä║:û▀>¢q·3└Ä╨╛ |√h└⍰⍰hf ╤ê⍰⍰ fü>⍰ NTFSu⍰┤A╗¬U=⍰
r⍰ü√U¬u⍰≈┴⍰ u⍰Θ▌ ⍰â∞⍰h⍰ ┤Hè⍰⍰ ï⌠⍰⍰=⍰ƒâ─⍰‰X⍰rß;⍰⍰ u█ú⍰ ┴.⍰ ⍰⍰Z3█╢  +╙f ⍰⍰ ⍰⍰⍰ Ä┬ ⍰⍰ ΦK +╙w∩┐ ╕=⍰f#└u-fü√TCPAu$ü·⍰⍰r⍰⍰h╕⍰h
R⍰⍰h     fSfSfU⍰⍰⍰h┐⍰fa⍰=⍰3└┐
⍰╢÷⍰ⁿ≤¬Θ■⍰ÉÉf`⍰⍰fí⍰ f⍰⍰⍰ ⍰fh    fP⍰Sh⍰ h⍰ ┤Bè⍰⍰ ⍰⍰ï⌠=⍰fY[ZfYfY⍰⍰é⍰ f ⍰⍰ ⍰⍰⍰ Ä┬ ⍰⍰ u╛⍰fa├í÷⍰Φ     í·⍰Φ⍰ ⌠δ²ï≡¼< t
┤⍰╗ =⍰δ≥├
```

Figure A3: Failed write attempt to NTFS physical disk with WriteFile