LUCRARE DE LICENŢĂ

Utilizarea Managerului de Filtrare Windows Pentru a Proteja Confidențialitatea Datelor Utilizatorilor

Conducător ştiinţific
Dr. CZIBULA Istvan Gergely, Profesor Universitar

Absolvent
BUZAS Laura Andrada

2019

BABEŞ-BOLYAI UNIVERSITY CLUJ-NAPOCA

FACULTY OF MATHEMATICS AND COMPUTER SCIENCE

SPECIALIZATION IN COMPUTER SCIENCE

DIPLOMA THESIS

Leveraging the Windows Filter Manager to Protect User Data Confidentiality

Supervisor
PhD. CZIBULA Istvan Gergely, Professor

Author
BUZAS Laura Andrada

2019

Contents

1. Introduction

Privacy is one of the things humanity has been keen on in the last few decades. To achieve safety software-wise, as a first thing, you would like to protect personal data.

What is important to keep in mind regarding data security is a matter of three concepts, namely availability, confidentiality and integrity, each one them having a high importance. Data availability means that it is available at any time, in case the user tries to access it.

Confidentiality is all about data that cannot be accessed without adequate authorization. A solution to take into account when it comes to confidentiality would be encryption. This is a proper solution, regarding the fact that a user's data will be available to read only for him.

Integrity means that user's data cannot be modified without authorization. In case integrity is violated, the data would be in an inconsistent state, not what the user is expecting. This would also render the data unavailable because the user does not have access to the old data anymore.

To achieve data safety you can consider having your data encrypted, but even with it, the integrity problem would still be open.

The main objective of this paper is to describe the process of design and implementation of a solution that tackles the integrity problem mentioned above. The designed solution is composed of: a minifilter driver that contains the monitoring and protection logic, a DLL that is an abstract interface over the minifilter functionality, and a Windows Desktop application built on top of WPF (Windows Presentation Foundation).

The second chapter of this paper contains four sections. First section is an high level overview of a few existing products, containing their description and functionality. Second section is divided in three main subjects: Windows Filter Manager where I'll be discussing the minifilter framework and how it can be used to achieve our proposed objective, C and .NET where I will motivate why I've chosen to use these languages. Third section is a detailed view of solution's design and implementation, especially the minifilter logic and communication. Last section focuses on how the application was tested.

Third chapter describes the further work that can be done to add more functionality to the current application.

Last chapter is a conclusion of the paper, summing up the main ideas.

1.1   Personal Contribution

Designing an SDK (Software Development Kit) that can be integrated by third-parties. My work consists of providing a documented interface for integrators in order to easily manage file protection rules that are based on Windows Kernel-level mechanisms.

The purpose of this is to provide an abstraction.

2. File Protection

2.1 Built in protection

Usually, operating systems do not have a built in mechanism that include a password protection method. However, there is a file permission policy that counts as a level of protection from other users.

2.1.1 Windows

File protection in Windows can be achieved through DACLs (discretionary access control lists). As a short explanation, a Windows object can have a DACL which contains ACEs(access control entries) that define what type of access any user or group of users can have. [4]

Usually, DACLs are used by defining the users and groups that can have access because the ones that are not included will have restricted access by default. If a DACL does not contain any ACEs the system will deny any type of access.
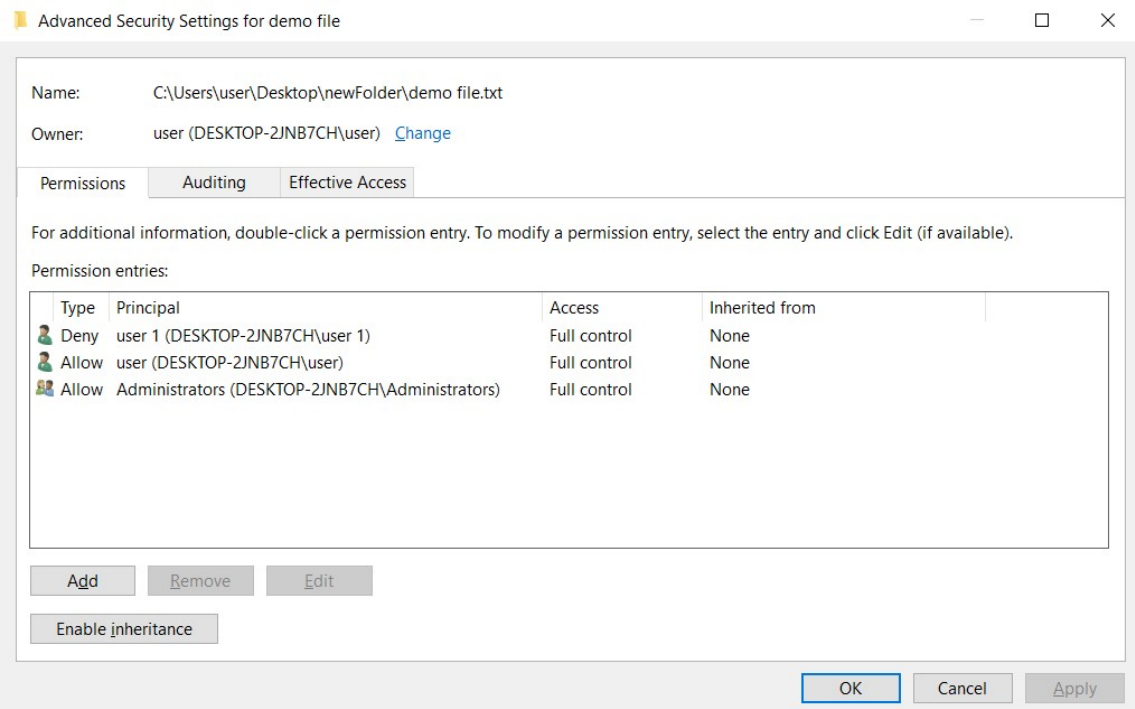

Figure 2.1: Advanced Security Settings dialog

Let us consider a new text file named "demo file.txt" on a Windows 10 OS that has 2 users, namely user and user 1, which both have administrator permissions. To restrict access for user 1 I have to go to file's properties **Security->Advanced**.
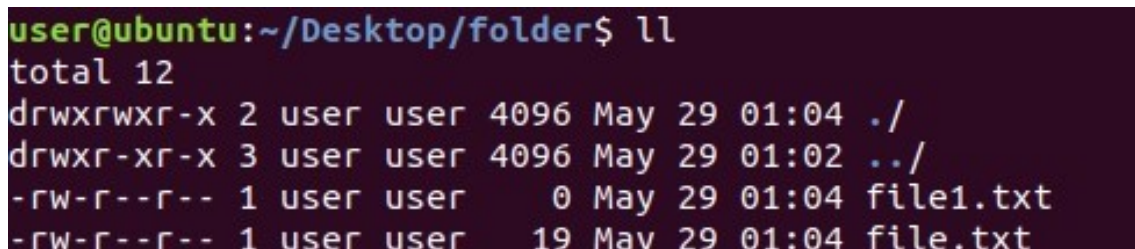
In figure 2.1 we can see the "Permission Entries" which maps to DACL we mentioned earlier whilst list's entries map to ACEs. I added the Administrators and user to have full control, and also I set user 1's permissions to deny any type of action. In this example we can notice that order in which permissions are set matters because user 1 cannot access the file, even if that user takes part from Administrator's group.

At the moment only the current user and other possible users in the Administrator group could have access to the file. Still, we are not fully protected when it comes for example to exploits. We could consider that we are logged in with user 1, which currently does not have any kind of permission for the demo file. There is the possibility of an exploit having first user's permissions and overwriting, encrypting or deleting the file which would be disastrous.

2.1.2 Linux

A similar mechanism exists in Linux, but it's more trivial. In my examples I will talk about Ubuntu, but the matter is almost the same for most of the Linux distributions.

For every file in Linux we have permissions for three main user groups: owner of the file, owner's group and all other users.

Figure 2.2: Linux file permissions

Figure 2.2 displays in the first column 10 characters, 9 of which represent the permissions for every file and directory. The first character represents the type of item, which can be a file or a directory. The next characters are split in sets of three: the first set represents user's permission, the middle set is for group permissions, and the last set represents the permissions for other users. For the described three-character groups we can have the following possible values:

- "-" - restricted access

- "r" - read permission

- "w" - write permission

- "x" - execute permission

The ownership of a file can be changed through chown command whilst the permissions of the file can be changed with chmod command, so the user can decide if the file can be accessed by others. This approach is more generic than the Windows mechanism because you cannot specify the rights for every user or group of users.

2.2 Existing products for Windows

As a similar tool, a first one that draws attention, is iBoysoft's File Protector for Windows, coming as a 7-day free trial or a payed version. As a main functionality, this tool allows the user to select the desired folders or files to be protected, along with the protection type (i.e. read/write/delete).

The application starts with a message box, demanding a password. After it was inserted correctly, the application starts, displaying on the main screen a list of files and folders the user has chosen to be protected.
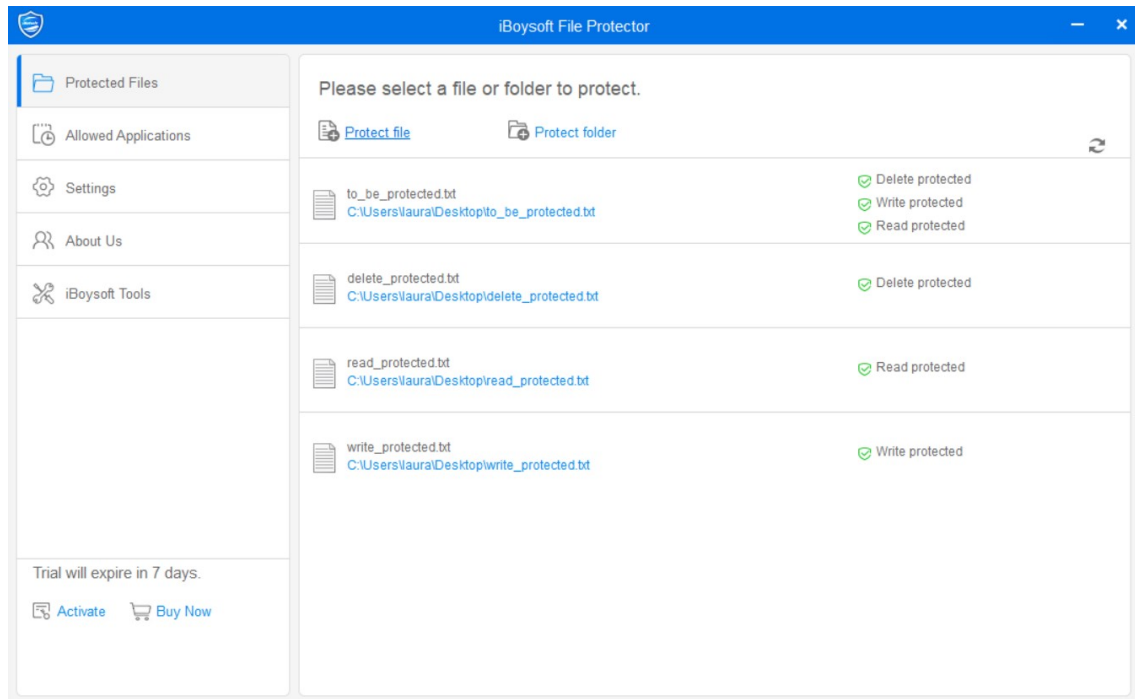
Figure 2.3: iBoysoft File Protector for Windows

In figure 2.3 we can see the main screen of the application, where we have a list of protected files, buttons for adding files or folders, and also the menu on the right.

Another feature of this tool is the allowed applications. There, the user can choose which applications that can have rights, in spite of the protection selected for the file or folder. After adding an application, the user should also add the files or folders that application can have access to.

If for example a text file was added to the list with read protection, that file cannot be opened by the user, which is a drawback for this application. There is one thing that can be done, and that is giving permissions to other applications to perform actions upon the desired files.

There is also the settings tab which contains two important sections, one where you can change the password, another one where you can choose the default protection types that are selected when adding a file or folder.

As a conclusion, the only time a password is required, is when the application is started.

2.3 Linux Proof of Concept

2.4 Windows Kernel Development

This chapter has as purpose to present an overview to main concepts related to Windows drivers and minifilter drivers as well as main functionalities.

2.4.1 Windows Drivers

A Windows Driver is a software component which usually makes the communication between the operating system and devices possible. What is important, and is not commonly known is that not all the drivers need to communicate directly with a device. For example, an I/O request can be passed through multiple drivers overlaid in a stack, like in figure 2.4. These drivers can complete, modify or pass along the request to the drivers below until the request is completed by one of the drivers. After a request is fulfilled a completion notification could be sent to interested drivers. [10]
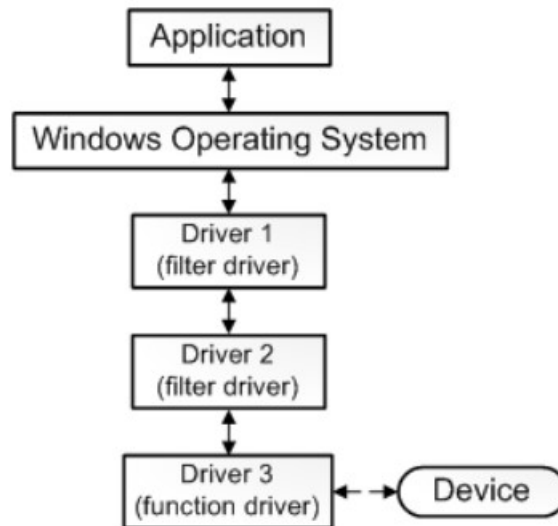
Figure 2.4: Request path through drivers[10]

I/O requests are sent to device drivers packaged in IRPs (I/O request packets). An IRP is a structure containing request's information. An important role pays the following field from the IRP structure: [7]

$$\texttt{struct \_IO\_STACK\_LOCATION *CurrentStackLocation}$$

I/O manager takes care for each driver to have an I/O stack location for every IRP. As an overview, the I/O manager creates for every IRP an array with I/O stack locations, every element corresponding to driver.

The I/O stack location of an IRP is comprised of the following information: [8]

- IRP_MJ_* - a major function code that the driver should handle

- IRP_MN_* - a minor function code which indicates a particular case of a major operation. This can be found for example in PnP (Plug and Play) drivers.

- A set of arguments related to the operation that should be performed, like the offset from where the driver starts reading and length that should be read.

- A pointer to the device object representing the target device for requested operation.

- A pointer to the file object which can be for example an open file or directory

As some basic examples of major IRPs which are also common, the following can be considered:

- `IRP_MJ_CREATE` - describes a request to open a handle to a resource

- `IRP_MJ_WRITE` - contains a buffer with data to be transferred to a device.

- `IRP_MJ_READ` - describes a buffer which the driver can fill in with data from a device.

- `IRP_MJ_CLEANUP` - this IRP is issued when all handles to a file object are closed.

In order to get information regarding the I/O performed operation in a driver IoGetCurrentIrpStackLocation has to be called . After the desired operations have been performed in the current high-level driver, the developer must set up the I/O stack location for the next driver. [8]

In terms of ways to organize a driver, there are multiple driver models such as Legacy Model, WDM (Windows Driver Model) or WDF (Windows Driver Foundation). The models consist of a set of API's that ease the development process. Depending on what type of driver a developer wants to implement a model can be chosen. In general, drivers can be of the following types: bus drivers, function drivers and filter drivers.

A bus driver is responsible with individual I/O bus devices. Bus drivers are required and there usually exist bus drivers for every type of bus on the machine. This type of driver can also report child devices connected to the bus. Microsoft and OEMs usually offers a set of common bus drivers such as USB, PCI and many others. [9]

Function drivers are the main driver for a device providing its operational interface. They usually handle writes, reads and manages device's power policy. This type of drivers can service more than one device. [9]

Finally, filter drivers are optional drivers which filter I/O requests thus making a change in device's usual behavior. Like functional drivers, a filter driver can service more than one device. Filter drivers can be of types: bus filter drivers, lower-level filter drivers and upper-level filter drivers. [9]

For a better understanding of filter drivers an example would be a driver that encrypts the disk data. It's functionality is to encrypt everything that goes to the disk and also decrypt the content requested from upper drivers. Therefore, the said filter driver could be an upper filter for the disk driver.

From this example we can easily see that the way drivers are structured is in compliance with the first two SOLID principles. The single responsibility principle is backed up by the fact that every driver should handle one particular functionality that is easy to maintain with respect to other drivers. Also, the open-closed principle is followed because if Microsoft or a third party driver vendor wants to have a new functionality it can be written as a new filter driver to extend the functionality the old ones were offering. This way the old functionality is still provided, and the new ones can work independently without modifying the old components.

Figure 2.5 displays a driver stack containing basic drivers in a stack. The bus driver is at the bottom of the stack because it is closest to hardware. Filter drivers, which are optional, add functionality to the bus an function drivers. And finally, the function driver is the one which provides the operational interface for the device.
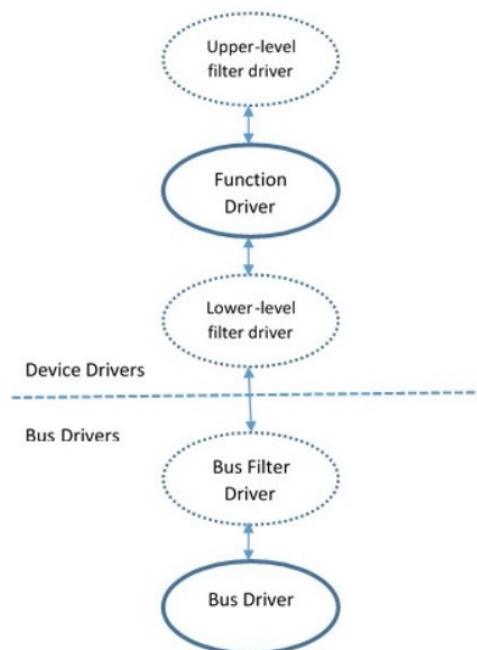

Figure 2.5: Driver stack[3]

2.4.2   Windows Filter Manager

During the years, a lot of problem with legacy filter drivers were starting to push the developer community to find solution to ease filters development. One great inconvenience is the fact that all IRP operation must be processed thus being a hard process. Due this and many other reasons, Microsoft decided to come with a solution regarding file system filters, which has been the development of Windows Filter Manager Driver. Nowadays files system filters are not supported anymore on newest Windows versions.

Filter Manager is a Windows driver containing functionality commonly required by drivers that intend to filter the file system. This set of functionalities is installed by default on Windows, but it only becomes active when a minifilter driver is loaded. [5] Some of the features from Filter Manager are loading/unloading a minifilter any time, having more control over where the minifilter is loaded in terms of altitude and also volume it attaches to and the most important feature being the possibility to filter IRPs by registering callbacks that are called before the operation is started (Pre callbacks) and after the operation is completed (Post callbacks).

Pre callbacks can be used to provide additional functionality to the file system stack, allowing developers to complete the IRP themselves. Post callbacks are useful when a decision must be made based on the IRP completion information (i.e. status). Both pre and post callbacks have two important input parameters: a PFLT_CALLBACK_DATA structure and a PCFLT_RELATED_OBJECTS structure. The first one is a view over the IRP that is presented to the minifilter driver. The second one is a structure that contains pointers to the objects that are related to the operation being processed (i.e. Volume, FileObject).

The altitude is a unique string identifier, interpreted as a number, which represents the position where a minifilter driver is loaded relative to other minifilters. The allocation of altitudes is managed by Microsoft thus the developers having to request an altitude in order to launch their driver into the market. [5]

In minifilters, and not only, there are a set of load order groups for file system filters, which are loaded at system startup. These load order groups represent the order in which drivers will be loaded in memory. [5] Load order groups are common for legacy filter drivers and minifilters, but the difference is that minifilters have the altitude therefore having an exact order in which the drivers will be loaded in spite of legacy filter drivers where filters having the same load order group will be loaded in memory in a random order.

In figure 2.6 we can see the path an I/O request can take from User Mode through Kernel Mode. First of all, the user request transforms into an IRP initialized by the I/O manager. After creation, the IRP gets passed through drives, one of them being Filter Manager. Next step is to pass the IRP through every minifilter registered with Filter Manager until last minifilter is reached or until completion. After the last minifilter is reached and the IRP has not been completed, the IRP goes through the next drivers until it reaches the lowest one in the stack.
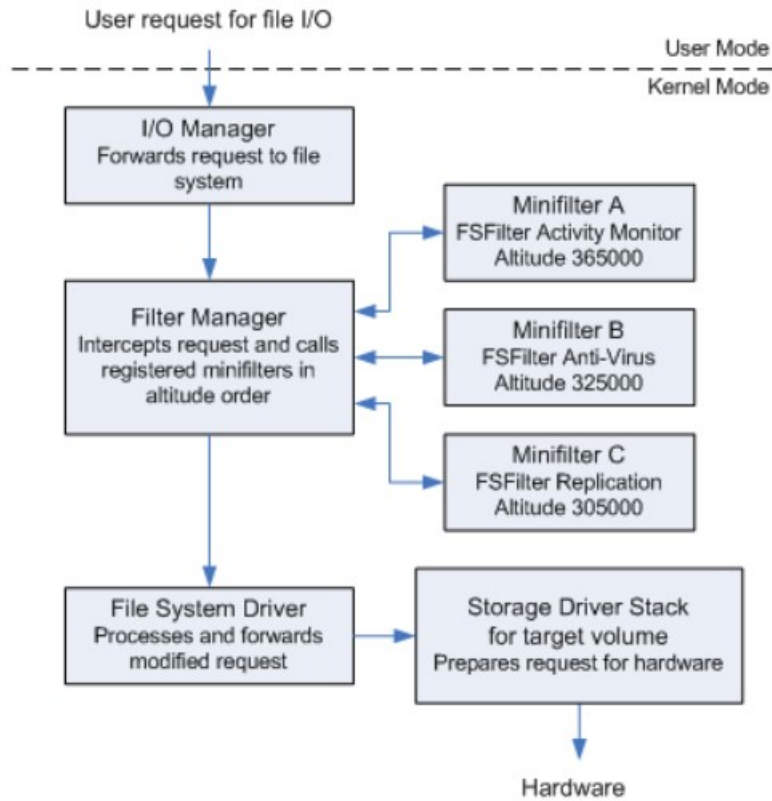
Figure 2.6: Filter Manager [5]

Setting the I/O stack location is no longer a concern for the developer because Filter Manager has this functionality already implemented.

2.5 Design and Implementation

In this chapter I will start to explain in detail the main features of the application. Following steps are to explain the logic for file blocking which is the key feature of the application. Also, for a better understanding of the whole process I will present the communication between application's components. Finally, as the development process is not easy for a new developer, I will state some of the challenges which I encountered.

2.5.1 High Level Overview of Design

Designed application is layered in 3 main components: 2.7

- fpf.sys

- FileProtectorCore.dll

- FileProtectorUI.exe

The first component, fpf.sys, is a minifilter driver built in C. It encapsulates the logic for file blocking, communicating with the dll through a filter port.

Secondly, FileProtectorCore.dll, built in C, exposes a set of APIs which can be used to communicate with the driver. This dll can be used for future integration in another applications.
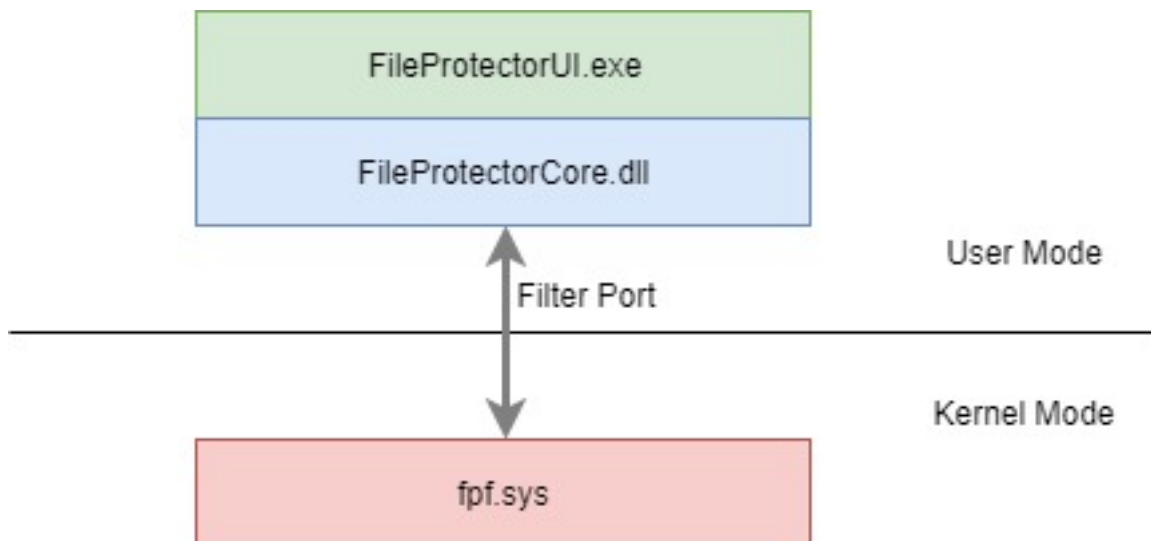

Figure 2.7: File Protector main components

Last component represents the GUI which was developed with WPF which is part of .NET Framework. This application is always running and it can be found on the system tray. Users have the following features:

- adding a file to a protected files list

- removing a file from the protected files list

- start or stop file blocking

- see a history with allowed/denied files

- see statistics about protected files

2.5.2 Monitoring and Denying Access

This subsection emphasizes the main feature of the application, the file blocking. This is accomplished using the Filter Manager framework by filtering IRP_MJ _CREATE IRPs issued down the file system stack and matching file paths against a database of known protected files.

This is achieved by registering pre and post callbacks by calling FltRegisterFilter. I/O operations filtering is started by calling FltStartFiltering. Denying access is achieved by completing the IRP_MJ_CREATE in the pre callback and also setting the IoStatus.Status field of the PFLT_CALLBACK_DATA input parameter to STATUS_ACCESS_DENIED.
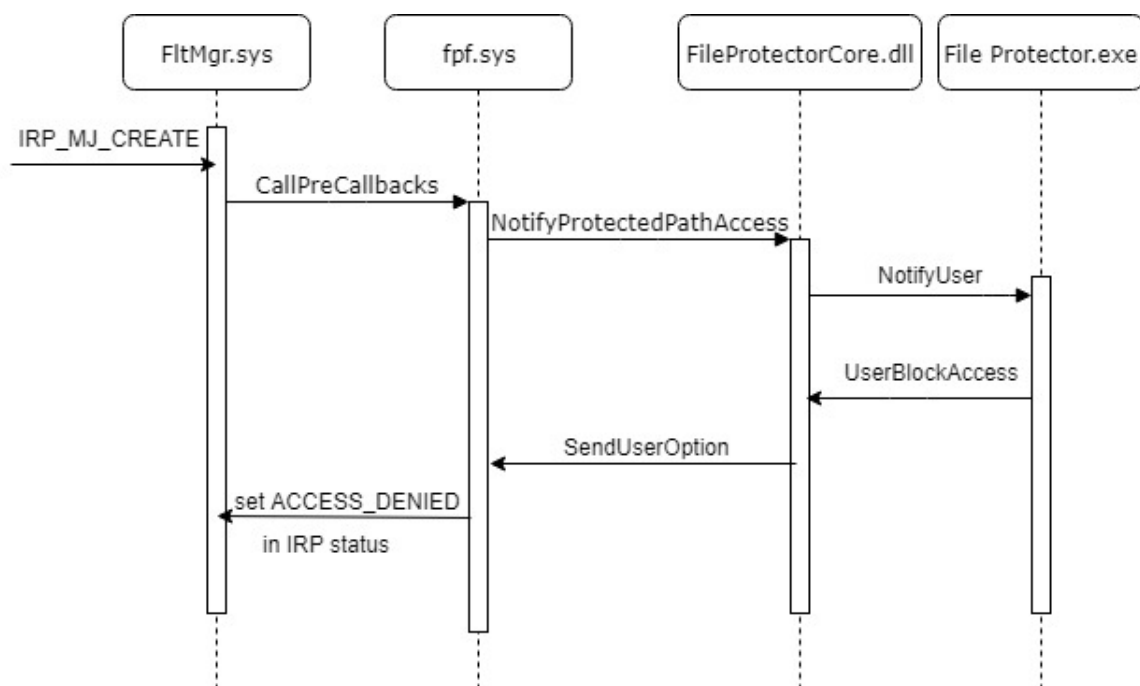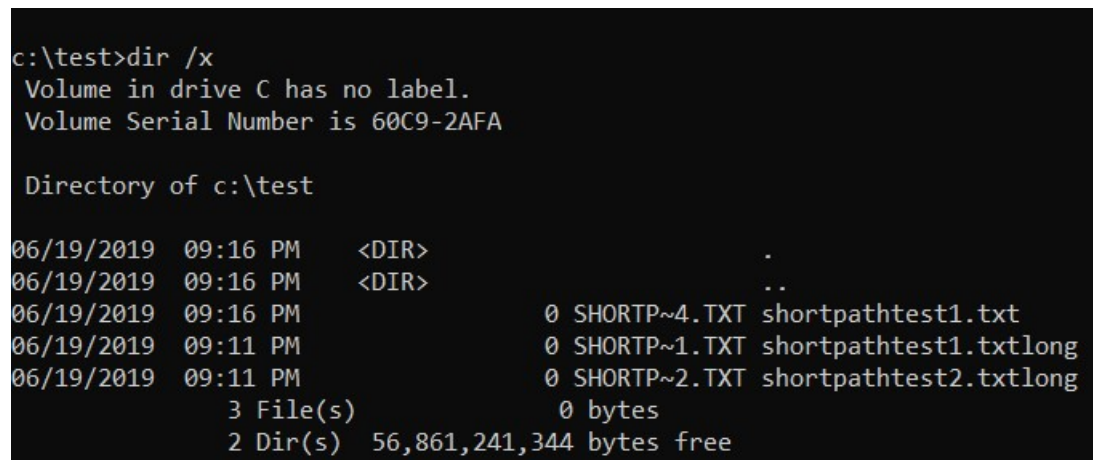

Figure 2.8: File Blocking

Diagram 2.8 shows a flow where a file from the protected files list is accessed and the user chooses to deny access to it. When a file is about to be opened, an IRP_MJ_CREATE is sent to the Filter Manager which calls the Pre callbacks for all registered minifilters with the PFLT_CALLBACK_DATA structure containing

a view of the IRP. fpf.sys has to get the file path by calling FltGetFileNameInformation. As parameters, this function requires the PFLT_CALLBACK_DATA, FLT_FILE_NAME _OPTIONS which specifies the name format for the returned file path, and lastly PFLT_FILE_NAME_INFORMATION which is an output parameter where the file path will be stored. The following code is the minifilter's call of FltGetFileNameInformation:

```
FltGetFileNameInformation(
        Data,
        FLT_FILE_NAME_NORMALIZED |
        FLT_FILE_NAME_QUERY_ALWAYS_ALLOW_CACHE_LOOKUP,
        &fileInformation
)
```

FLT_FILE_NAME_NORMALIZED is required to get the normalized name. The other possibility was using FLT_FILE_NAME_OPENED but this could possibly give us unresolved paths (i.e. short paths), basically returning the path that was used to open the file. Short paths in Windows are created in case of file names longer than MS-DOS 8.3 naming convention, meaning the file name should have maximum 8 characters and it's extension maximum 3 characters.

This is potentially dangerous because a malicious application could try to bypass the File Protector by accessing a file through its short name. Example of short for paths: c:\test\shortpathtest1.txt, c:\test\shortpathtest1.txtlong and c:\test\shortpathtest1.txtlong 2.9


```
c:\test>dir /x
 Volume in drive C has no label.
 Volume Serial Number is 60C9-2AFA

 Directory of c:\test

06/19/2019  09:16 PM    <DIR>                          .
06/19/2019  09:16 PM    <DIR>                          ..
06/19/2019  09:16 PM                 0 SHORTP~4.TXT shortpathtest1.txt
06/19/2019  09:11 PM                 0 SHORTP~1.TXT shortpathtest1.txtlong
06/19/2019  09:11 PM                 0 SHORTP~2.TXT shortpathtest2.txtlong
               3 File(s)              0 bytes
               2 Dir(s)   56,861,241,344 bytes free
```

Figure 2.9: Short Paths

It can be seen how the filename is truncated and appended a tilde and an index (i.e. SHORTP 12.9). In order to obtain the "real" NT path, we need to normalize the path by using the flag previously described.

FLT_FILE_NAME_QUERY_ALWAYS_ALLOW_CACHE_LOOKUP first checks filter manager cache if name is already resolved. In case it is not, it queries the file system for the file name, by opening the file, if it is currently safe to do so.

After the path has been solved, a notification is sent into the dll using FltSendMessage. The communication needs a PFLT_PORT which needs to be created with

FltCreateCommunicationPort in the minifilter and opened with FilterConnectCommunicationPort in the dll. Then the message is retrieved and a toast notification is displayed. 2.10
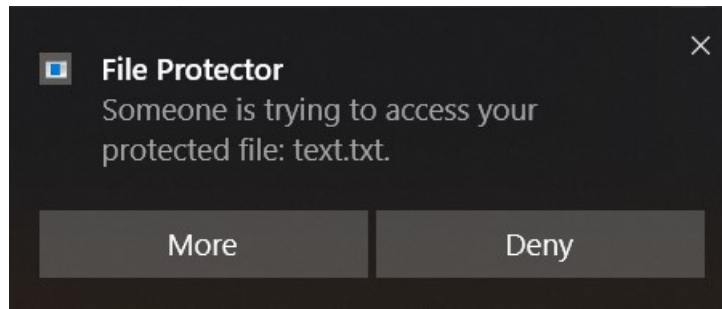

Figure 2.10: Toast Notification

In case the "More" button is pressed, a secure desktop will be opened displaying the full path and the process trying to open the file. The secure desktop is necessary because we need to create a window that cannot get messages from any other process. By creating a new desktop specifically for our window, other processes will not be able to send messages (i.e. SendMessage, PostMessage) to our window because they will not have a handle for it. For a new desktop to be created, the following functions needed to be imported from user32.dll and kernel32.dll: GetThreadDesktop, GetCurrentThreadId, CreateDesktop, SwitchDesktop, SetThreadDesktop and CloseDesktop.

Firstly, the old desktop should be preserved for recall and that was done by calling GetThreadDesktop(GetCurrentThreadId()) and keeping the returned value into a variable. To create the desktop and make it visible the calls should be performed in this order: CreateDesktop, SwitchDesktop. After the user made a choice, the old desktop must be restored to the initial state preforming the following calls: SwitchDesktop, SetThreadDesktop, CloseDesktop.

After the user denies the file opening, user's option is sent to the dll which sends it further to the minifilter by calling FilterSendMessage. The driver will block the file depending on user's response.

2.5.3   Inter Component Communication

As said before, communication between the minifilter driver and the dll are done through a filter port. The cases where the dll and the driver need to communicate are the following:

- when a user adds a file to the protected files list a notification must be sent from user mode to kernel mode

- when a file that is protected is opened the minifilter must send a notification to user mode for user approval

16

Communication between user mode and kernel mode is supported by the Filter Manager. The minifilter driver can create communication port which will begin to listen for incoming connections by default.

Port's security is handled in the minifilter driver by specifying a security descriptor. Among Windows APIs we can find a variety of functions regarding creating and initializing a security descriptor for a new object from which I used FltBuildDefaultSecurityDescriptor. This function builds a security descriptor that can be further applied to a communication port which means only users with system administrator privileges will have access to the port. [2]

FltCreateCommunicationPort demands 3 callbacks used for communication between kernel mode and user mode: one that will be called when a client connects on the port, one that is called when a client disconnects, and one which is called when user mode applications call FilterSendMessage to send a message to the minifilter driver through the port. [2]

For a client to connect on the filer port, FilterConnectCommunicationPort must be used. After the call has succeeded, FilterSendMessage can be used to communicate from user mode to kernel mode and FilterGetMessage is used to wait for messages sent from kernel mode to user mode.[2]

The path of a file can contain multiple components separated by a backslash. The prefix of a path, which determines the namespace, depends on system's interpretation. In user mode, full file paths start with a drive letter (volume name) followed by a backslash like C:\ or D:\. Another way of naming files is using the Universal Naming Convention (UNC), which is commonly used for accessing remote file shares (i.e. \\ServerName\ShareName\FileName).
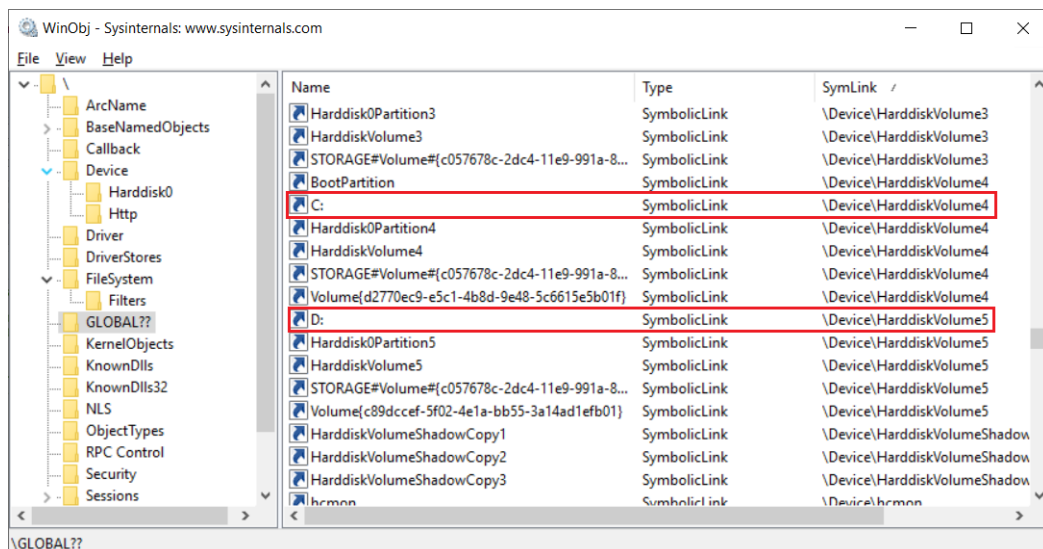

Figure 2.11: WinObj GLOBAL?? namespace

The Windows kernel uses NT paths, meaning that volumes will be named differently from the usual user mode volume name, like "\Device\HarddiskVolume4" instead of the C:\ drive letter. For example, if we have the DOS path C:\folder1\file.txt,

the corresponding NT path will be similar to: \Device\HarddiskVolume4\folder1\file.txt
[11]. That's because "C:" is a DOS device, which is actually a symbolic link to the
actual device (i.e. \Device\HarddiskVolume4).

WinObj is a Windows Sysinternals tool that uses Windows APIs to access and
display information about namespaces. If we go to the GLOBAL?? namespace in
WinObj, we can see thatthat C: and D: have as symbolic links \Device\HarddiskVolume4
and \Device\HarddiskVolume5, as highlighted in figure figure 2.11 .

When paths are sent from user mode to kernel mode, they must be converted to
NT paths by resolving the drive letter symbolic link by calling the ZwQuerySym-
bolicLinkObject API.

2.5.4   Encountered Challenges

Developing a minifilter driver development process is different from what most
of programmers are used to because it requires a lot of attention and especially
knowledge of Windows drivers in order to get a satisfying end result.

One of the first challenges I want to talk about is testing. Windows drivers must
be tested on a test machine, because any little mistake could lead to compromising
system's availability. In order to test my code I've created a Windows 10 Pro
x64 virtual machine. For drivers to be loaded, Windows requests drivers to be
signed with a certificate obtained from a trusted certification authority. However, for
testing purposes the drivers can be signed with a test certificate and thus Windows
does not load test-signed drivers.

In order to load a driver with a test signature the machine has to be set to
test mode which is done through following commandline (it must be opened with
administrator rights):

```
bcdedit -set testsigning on
```

The result can be seen almost immediately because a watermark will be shown on
the lower right side of the screen and the driver with test signature can be loaded.

Another challenge was debugging the driver. Because every bug would lead to a
BSOD(Blue Screen of Death) debugging is a good solution when it comes to testing
newly added code. Similar to setting the machine as a test one, kernel debugging
should be enabled with the command:

```
bcdedit -set debug on
```

When it comes to creating a new desktop, WPF does not support creating win-
dows in the newly created desktop. The result was a window with a close button
and nothing rendered on it. A solution for this issue was creating a Windows Forms
Window instead of a WPF one, and it will be displayed correctly.

2.6 Testing Approach

In order to achieve a reliable software, the application has been tested using both black box and white box testing methods, which will be described in the following sections.

2.6.1 Driver Verifier

First of all I will describe some basic functionalities of Driver Verifier, that I have also used to test the minifilter.

As the name suggests, Driver Verifier is a tool that can monitor Windows kernel drivers. It can detect major problems, such as memory leaks, memory corruptions or deadlocks. This tool is developed by Microsoft, making it's first appearance in Windows XP as a command line utility. Over the time this tool has gained more checks as well as a graphical user interface namely Driver Verifier Manager, that allow a programmer to select the preferences for driver's testing.

2.6.2 Static Code Analysis

In this section I will talk about source code annotation language (SAL). This is used to make the code more explicit, in terms of behavior, parameters, return values, or other things* to make it more understandable. In other words, programmer's work is to use annotations considering the desired functionalities, and compiler's work is to verify if the written code corresponds with the annotations.

All SAL annotations have a specific "look", and that is _Annotation_name_. For example, some parameter most used SAL annotations are _In_ and _Out_. _In_ makes sure the compiler interprets data as being input data that cannot be modified in function's scope. _Out_ allows the programmer to use the parameter as an output one, as long as the space for that parameter has been allocated by the programmer.

As for behavior, a simple example is _Check_return_. If this annotation is used, the caller must inspect the returned value. In case the function has a void return type, an error will be shown at compilation time. A similar example is _Must_inspect_result_, which contains the functionality of _Check_return_ and adds that any output parameter should also be used after the function call.

The following example is the MessageNotifyCallback that is called whenever a user calls FilterSendMessage to send a message to the driver.
_In_opt_ is used to indicate that there is an input parameter that may be also null.
_In_reads_bytes_opt_

```
NTSTATUS
FLTAPI FpOnClientNotify(
_In_opt_ PVOID PortCookie,
_In_reads_bytes_opt_(InputBufferLength) PVOID InputBuffer,
```

19

```
    _In_ ULONG InputBufferLength,
    _Out_writes_bytes_to_opt_(OutputBufferLength, *ReturnOutputBufferLength) PV
    _In_ ULONG OutputBufferLength,
    _Out_ PULONG ReturnOutputBufferLength
    )
```

After these examples anyone should be convinced that SAL is beneficial from multiple reasons, and that is also why I used it in my own code.

3. Further Work

Like most of the application, this one can also have functional improvements as well as user experience improvements.

3.1 File name matching efficiency

At the moment, file name matching is done in a trivial way. Let us consider that a user tries to open a file which is already protected. In the minifilter driver, the file name is searched in the list, which leads to an $O(m*n)$ complexity, n being the length of the searched file name and m representing list's length.

A solution to improve the file search would be implementing the Aho-Corasick Algorithm. At a first glance it can be said the complexity would be reduced to $O(n+m)$, with n and m having the same meaning as mentioned above. What would need to change in the current code is that list should be removed, and replaced with a trie. At system startup the driver should load the protected files into a trie containing all word possibilities. Next step is to extend the trie into an automaton which can mach any word from the given list of file names. Now if we review the complexity and we consider the fact that the trie is made just once, when the system starts, the complexity would be $\theta(n+m)$. There is only one situation when the trie needs to be modified, and that is when a user adds a new file for protection. Usually users would add most of the files when they first install the application, after that the number of files being almost negligible. To summarize the last paragraph Aho-Corasick is a great improvement in terms of complexity when it comes to string matching.

Another good approach when it comes to performance, would be replacing the list with a crit bit tree.

3.2 Enterprise environment integration

Thinking of a master with multiple slaves architecture, the application can have a master console where the sysadmins can add protected files which will be protected by default in all machines in the network.

Regarding the master console, the sysadmins would have an exact statistic on the network activity regarding file protection. This means the current history would be changed to display information divided by machines and a statistic as a whole. As

for the other features, the protection could be turned on or off only from the main console in all the network.

Users would have almost the same view of the application. The only difference is that there would be a new tab listing protected paths added by the organization's system administrators or security experts. Also, the users would still be able to add personal files if they desire, but they cannot touch the ones protected by default. The other thing that would be different is that they won't be able to turn the protection on or off.

This would be a great approach because control is important in such an environment, especially when it comes to data protection. Malware trying to exfiltrate corporate date would be easily spotted in case it got into the network. Moreover, the statistics regarding file access on infected machines would be useful for damage assessment.

4. Conclusion

Data security is an important matter when it comes to privacy. As mentioned before, integrity is a matter that is not addressed with a maximum level of protection on Windows and neither on Linux. For this little flaw I've come with a solution available only for Windows that adds functionality to the existent mechanisms.

The designed solution provides users with file protection, with guarantee that only they can access their protected files. The most important part of the application is the minifilter driver, where the logic for file blocking has been done. However, the user communicates with the minifilter through a UI having as main functionality the management of the protected files along with other features.

Lastly, the application can be improved regarding performance by implementing various algorithms for file name matching and also it can be extended to fit smoothly in a large scale enterprise environment.

Bibliography

[1] Bottazzi G., Italiano G.F., Spera D. "Preventing Ransomware Attacks Through File System Filter Drivers". In: Milan, Italy, Feb. 2018. URL: http://ceur-ws.org/Vol-2058/paper-08.pdf.

[2] *Communication Between User Mode and Kernel Mode.* https://docs.microsoft.com/en-us/windows-hardware/drivers/ifs/communication-between-user-mode-and-kernel-mode. Apr. 2017.

[3] *Device Object and Driver Stack.* http://www.windowsbugcheck.com/p/since-understandingof-device-objects.html. June 2019.

[4] Microsoft. *DACLs and ACEs.* https://docs.microsoft.com/en-us/windows/desktop/secauthz/dacls-and-aces. 2018.

[5] Microsoft. *File System Minifilter Drivers.* https://docs.microsoft.com/en-us/windows-hardware/drivers/ifs/file-system-minifilter-drivers. 2017.

[6] Microsoft. *Function Drivers.* https://docs.microsoft.com/en-us/windows-hardware/drivers/kernel/function-drivers. June 2017.

[7] Microsoft. *I/O Request Packets.* https://docs.microsoft.com/en-us/windows-hardware/drivers/gettingstarted/i-o-request-packets. Apr. 2017.

[8] Microsoft. *I/O stack locations.* https://docs.microsoft.com/en-us/windows-hardware/drivers/kernel/i-o-stack-locations?fbclid=IwAR30-_rZppRnPIp8wAkwBaD-KtFeiMqv1BluSfGMfEPOsO-RcxHLiAmRch4. June 2017.

[9] Microsoft. *Types of WDM Drivers.* https://docs.microsoft.com/en-us/windows-hardware/drivers/kernel/types-of-wdm-drivers. June 2017.

[10] Microsoft. *What is a Driver?* https://docs.microsoft.com/en-us/windows-hardware/drivers/gettingstarted/what-is-a-driver-. Apr. 2017.

[11] *Naming Files, Paths and Namespaces.* https://docs.microsoft.com/en-us/windows/desktop/fileio/naming-a-file#short-vs-long-names. May 2018.

[12] Rienhardt F. "Kernel-based monitoring on Windows (32/64 bit)". In: Dec. 2015. URL: https://bitnuts.de/KernelBasedMonitoring.pdf.

[13] Yosifovich P., Ionescu A., Russinovich M.E., Russinovich D.A. *Windows Internals, Part 1: System architecture, processes, threads, memory management, and more.* Seventh Edition. Redmond, Washington: Microsoft Press, 2017.