Excel@FIT 2019
http://excel.fit.vutbr.cz

Transparent Encryption with Windows Minifilter Driver

David Pořízek*

Abstract
The goal of my work is to implement a solution which would be able to extend a Data Loss Protection (DLP) system by preemptively protecting any data which are about to leave an Endpoint. This paper aims to describe the approach that has been used in order to achieve this type of data protection. I have chosen to implement the application using a Windows Minifilter Driver Framework, which provides developers with an interface to directly filter, modify, and create requests sent to file systems (FSs). This approach also inherently provides better security than user mode (UM) solutions, since the framework runs fully in kernel mode (KM) and utilizes security elements that are already in place to protect the Windows Kernel. The application is finished and has been already briefly tested with Safetica DLP. It is able to seamlessly protect user-specified files and provide the user with a plain-text view of said files despite them being stored in an encrypted form on the disk. There are certain limitations to this approach, which will be described in section 3, but the solution attempts to overcome them in the best possible way. On top of extending DLP systems' functionalities, this work should provide an insight into transparent encryption and more generally, explore the possibilities of a kernel driver when it comes to modification of file views. To a reader, it should present a general idea of what an implementation of this solution entices, a clear starting point for his/her own work, and suggestions how to further improve it.

Keywords: Transparent encryption — Minifilter driver — Kernel mode

Supplementary Material: *N/A*

*xporiz03@stud.fit.vutbr.cz, *Faculty of Information Technology, Brno University of Technology*

1. Introduction

This project approaches the encryption and DLP ideology from a different angle. Instead of preventing sensitive data leaks and focusing on multiple different channels through which the data could escape from an endpoint, it preemptively protects such data and makes sure that in case of a leak, the leaked data are accessible only in their protected form outside the original endpoint.

As it was mentioned earlier, the main goal of this work is to extend existing DLP systems and provide a new functionality to the,. The core part of this project is the transparent encryption logic. It can be further separated into two parts – an application providing the transparency and a module securing files by encryption. There should be an interface, which the DLP system can use to configure the solution. Additionally, users might need to access the protected data without having the option to install a driver, such cases should be taken into account. Furthermore, implementing the encryption logic as an independent module could prove beneficial for DLP systems to fulfill their security requirements for external solutions.

I decided to analyze multiple widely known DLP systems which provide a similar approach to the data protection. Since there is a number of such DLP systems, I rather focused on analysing their encryption features from a higher level, for the sake of this paper. The approaches can be summarized into three different categories. Unfortunately, majority of the existing solutions are commercial products, therefore information that can be obtained about them is limited to only public white-papers and other materials released by them. Nevertheless, it is still possible to gather valuable information and at least understand the pros and cons of each of the approaches. They will be described in greater detail in the following section 2.

In my case, I was able to cooperate closely with Safetica to deliver them a solution, which is able to provide the file protection mentioned in previous paragraphs. Specifically, my solution implements a file system minifilter driver, which loads itself into the file system driver (FSD) stack and modifies requests of files and responses to these requests coming to/from a file system. This driver performs the transparent encryption on-the-fly. It also focuses on the extensibility of the encryption logic, and utilizes security features that are already in place to protect the Windows kernel. While the the idea of the transparent encryption is fairly simple in itself, the driver solution must mind Cache Manager (CM) and Virtual Memory Manager (VMM) on top of the file system and adhere to their requirements to make sure it does not break the flow of either of them.

The final solution is able to transparently protect files that have been specified as sensitive. It also provides a UM application to decrypt any of the protected files, in cases where the user is unable to install the driver. This enables it to work as a stand-alone solution as well as in a conjunction with a DLP system.

2. Summary of Different Approaches

As it was mentioned in the introduction, there is a number of DLP systems, which provide a similar approach to the file security as in this work. Since these systems are usually commercial products, obtaining information about the specifics of their implementations is unlikely. Rather than discussing each of the DLP systems separately, in this paper, I will summarize the approaches into three different groups and describe them. The approaches have been summarized as follows: full-disk encryption, server-based proxy encryption, file-based transparent encryption.

Full-disk encryption works just as the name suggests – it encrypts the whole disk during the initialization phase and then works in the background to encrypt any data that are being copied to the disk and to decrypt data copied from the disk. While this is certainly a complete solution, it lacks some useful features. It is not possible to encrypt only few selected files. Although, it is possible to encrypt a folder instead of the whole disk, it is still not possible to mix encrypted and plain-text files in this folder. Usually, it is required that an encrypted unit (disk or folder) is mounted in order to access the decrypted data. On the other hand, it is easy to setup and once the initialization is completed, it guarantees protection of all files inside the encrypted unit.

Server-based proxy encryption utilizes a proxy server, which is located at the company's network node and functions as a gateway for the network. This allows the server to filter and analyze all communication that is leaving company's premises. The server is then able to react to sensitive data leaving the premises, for example, by encrypting said data or by notifying an administrator. This approach usually requires a hardware component in order to work properly, although installing such component is very simple most of the times, it may be a disadvantage for some companies, since they have to physically obtain the component first. A disadvantage to this approach is the inability to protect any data copied to an external storage. This introduces a security risk, which is impossible to solve with just this approach alone and may require an additional solution, to protect local data.

Lastly, there is the file-based transparent encryption approach. In its core, this approach provides user with an illusion that he/she is working with unencrypted files, while in the background, these files are stored in an encrypted form on the disk. This is similar to the full-disk encryption, but the main difference is that the encryption is performed Just-in-Time (JIT) therefore there is no big overhead required, before it is ready to be used. It also maintains the protected state of files even when they are being moved across different devices. The main disadvantage with this approach is the complexity of an implementation. This, of course, depends on the chosen design. Implementing transparent encryption inside UM provides more flexibility, but if utilizing KM security features is our goal, there are only two design options. One of them is file system filter driver, that would modify any incoming requests to satisfy the requirements of the file system, the CM, and the VMM, which is the design I chose for my work. The other design is based around implementing a full scale cache and memory management in the driver. This would require the driver to assume these functions from the file system and become independent on it. This approach is obviously exponentially more complicated, since the functionality would be comparable to a full-scale file system. But it would solve issues with obtaining cache locks and flushing the cache mentioned in section 3.

3. Proposed Solution

Based on the analyzes and research briefly described in section 2, I decided to implement a transparent encryption with the filter driver. The following section describes the chosen approach and discusses its advantages and disadvantages in greater detail.

The implementation consists of two core parts – the driver, which implements the transparency, and the encryption module. There is also a graphical user interface application (GUI), which is implemented in UM and allows users to decrypt any file even if they lack administrator privileges (this issue will be further discussed later in this section).

The driver is implemented using the Windows Minifilter Framework [1] and therefore the language used is C. The main advantage of a minifilter driver lies within its high abstraction from system internals compared to other types of drivers. It can filter requests made towards a disk without having to worry about power management of said disk while also being able to use a full set of functions defined for filter drivers within the Minifilter Framework. It can also access and modify certain parameters of requests and responses made towards disk, also called Input/Output Request Packets (IRPs).

Modification of the IRPs' parameters allows the driver to change a file view that is being presented to the user and thus is the basis for the transparency logic. Therefore, the user is seamlessly working with decrypted data, although it has been already secured by the driver's encryption logic. The file view modification is described in detail in section 4.

The encryption logic is implemented as a separate module inside a library. This allows anyone to adjust the encryption algorithms to suit their needs. It is also possible to change the format which defines, how the file is stored on a disk. By default, the driver uses AES-256 encryption. The encryption module is described further in section 5.

Implementing the transparent encryption as a driver provides number of advantages. A Windows driver has to fit within the interface defined by Windows for drivers and thus it utilizes all the KM protections in place to secure native drivers. Generally speaking, a driver implementation provides a better performance than a UM application, because the driver is allowed to work in a highly elevated mode, bypassing majority of security checks and verifications imposed onto UM applications.

Furthermore, the transparent encryption approach provides a way to protect only specific files within a folder. This falls well within the usual DLP workflows, where sensitive data are detected and a predefined operation is issued as a response, for example, alerting the administrator or encrypting the file. The way the encryption module is implemented, allows DLP systems to change any of the algorithms used if they deem it necessary.

On the other hand, the biggest disadvantage of this approach is that it relies on implementation details of FSDs and will likely misbehave on other file systems which it was not adjusted for. That being said, it would be possible to modify the driver behavior in order to work properly on other file systems as well. The driver has been tested on NTFS and FAT32 file systems and works as expected.

There were two issues with this approach, which had to be overcome. The first has been hinted earlier in this text – the requirement of administrator privileges. A Windows driver can be installed on a computer only with an administrator elevated account. This is usually not an issue for DLP systems, since the installation is done by the company's administrator. But it would be useful to be able to access the protected files on an external computer, where the administrator privilege may not be easily acquirable. For this reason, the solution provides a GUI application, which is able to access files previously secured by the driver. This obviously breaches the transparent approach, but it is meant rather as a last-resort tool.

The second issue is caused by "lieing" to CM. When a request is made towards a disk, for example, to read a file's content, the file system fills in the information about the requested file and sends the information back alongside with the requested file's content. This also means, that it has to initialize the file in CM and VMM. Consequently, the FSD becomes the owner of locks issued for access to the cached data of the file. This complicates our transparent approach, because we need to access the cache of files we are protecting in order to flush it and initialize it again with modified parameters. This is required in cases where the driver was not loaded fast enough and the file has been already cached in its unmodified state. The issue becomes almost nonexistent by requesting the FSD's locks before flushing the cache and making sure we will be able to actually obtain them, by cancelling the current file request and issuing a new one for any file we haven't filtered yet. While, in theory, the issue might not be fixed completely, I have not encountered it during my tests. It would also pose only a small inconvenience to the user, since he/she would

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

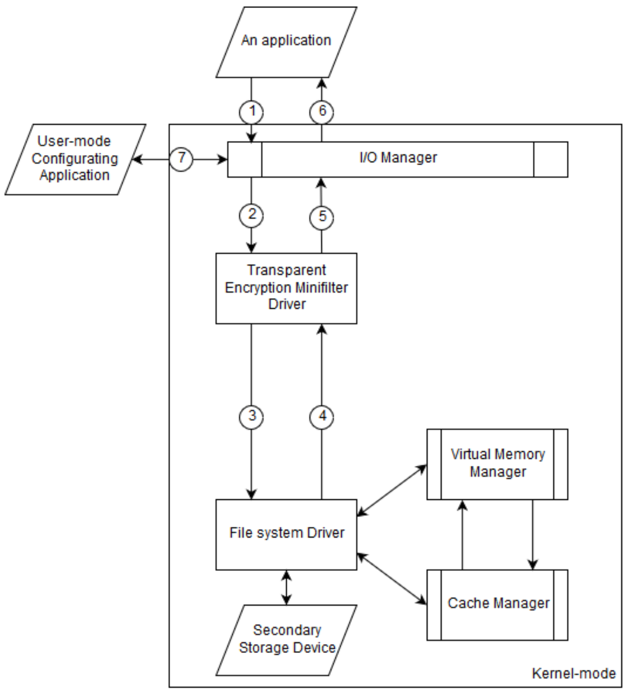

Figure 1. Figure displays a flow of I/O request as well as the position of the driver in this exchange.

be presented with the secured content of a file he/she is trying to access and to fix the issue he/she would need to restart the computer.

4. Modified File View

Modifying the file view is a fundamental logic in a transparent encryption approach. The main idea behind changing the file view is to provide an illusion that a user is currently working with normal files, while, in fact, they have been already modified by the driver (for example, encrypted). It requires the driver to modify a number requests and responses made towards a disk to achieve acceptable results. Figure 1 shows all participants in a file request as well as driver's position in this communication. The information in the following section has been gathered from the book [1].

When modifying a file view, there are two important participants who we have to indirectly communicate with, in order to modify the view properly. These participants are the CM and the VMM. The cache manager allocates and caches the memory that is accessed or is about to be accessed. A file is usually cached when it is first accessed by a handle. This can happen even before a user reads any of the file's content, because the system may open the file's handle to query some details. This presents a fairly big complication, because requests to read a file's content can be satisfied from cache, bypassing the file system driver stack, thus bypassing the minifilter as well. Therefore, the driver must preemptively store plain-text data when the cache is crated for a file, otherwise it would not be possible to read file's content when satisfying the requests from cache. Since a minifilter driver is not directly responsible for managing the file's context in the system, it cannot directly contact the CM and must rely on the file system to contact the CM for him/her.

Moving onto the VMM. VMM manages memory-mapped files (MMF). There is a similar issue as with the cached memory, that is, if the memory mapped files become initialized without the minifilter's interference, the user would be able to only access the encrypted data. Instead, the driver has to detect the MMF initialization process and provide the VMM with decrypted view of the file, which consequently becomes mapped and accessible to the user.

Even though the text mentions communication with the CM or the VMM, none of the communication is done directly with the relevant manager as described in the previous paragraphs. Every request has to go through the file system first and then the file system decides whether it needs to communicate with either of them. The minifilter is able to influence this communication by modifying requests it sends to the file system (Figure 1 – step 3). This modification needs to be done in such a way, that the file system does not reject the request (for example, with an `END_OF_FILE` status when the driver modifies the file's size), but rather forwards or completes the request as planned.

There are three different file size values which a FSD recognizes and keeps for each file. All of these values are utilized during a file request in some way, but to provide a modified file view the driver has to modify only one of them – `FileSize`.

The three different file size values:

- The `AllocationSize` is a value which reflects the actual on-disk space reserved for the file. It is a multiple of the minimum allocation size of the file system which manages the storage where the file resides.
- The `FileSize` value defines the end-of-file (EOF) mark. All read operations return EOF when attempting to read beyond this size.
- The `ValidDataLength` represents the amount of data stored within the file.

5. Encryption Module

The encryption works by encrypting the file's content and storing relevant information in a header which is appended to the file. This allows the driver to work without the need to store any information about the current session in its memory and to work out-of-the-box.

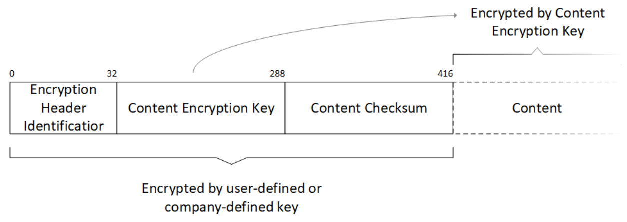
Encrypted by Content Encryption Key
0
32
288
416
Encryption Header Identificatior
Content Encryption Key
Content Checksum
Content
Encrypted by user-defined or company-defined key

Figure 2. Figure describing the encryption header structure.

Figure 2 describes the structure that is used for the encryption header. The header begins with a fixed length identification which is there to help the driver distinguish its encrypted files. The encryption key is generated by the driver for every encryption request. The user/company defined key is used to encrypt the whole header that is appended and has to be obtained externally. It can be obtained either by directly calling the driver's interface or by using the provided external application to set a user password. It also contains a simple checksum to verify, whether the header has been altered.

As for the algorithms used, by default the solutions uses AES-256 and MD5 for checksum, but the algorithms can be easily changed by compiling a different encryption library (as long as all the required interfaces of the driver are provided). The majority of other encryption based DLP solutions tend to use AES-128 or AES-256. Based on this and the fact that AES-192 and AES-256 are deemed strong enough by the National Security Agency (NSA) to protect TOP SECRET information [2], these algorithms should be secure enough to be used as default ones in the final solution.

6. Conclusions

This paper summarized and described the approach which was used to implement the transparent encryption inside Windows kernel. The implementation consists of two core parts – the transparency logic and the encryption module. While the transparency logic is implemented as a minifilter Windows driver, the encryption module is a library, which can be potentially swapped to adjust the encryption algorithms and/or the encryption header.

The solution has been tested and works properly on NTFS and FAT32, either closely with Safetica or as a stand-alone application. It provides all the necessary interfaces to properly configure it and use it.

The work explores possibilities of Windows KM and proposes a transparent encryption solution, implemented as a minifilter driver. While there are some limitations to this approach, the solution makes the best effort to work around them.

This work could also be treated as an introduction to file view modification by using Windows Minifilter Driver Framework. Despite the fact, that the solution may not be the most optimal, it should provide the reader with enough information on the subject, to properly understand all the advantages and pitfalls of such project.

I would like to use this work as a basis for my future project, where I would focus on implementing a complete file system driver, which would take over the cache and memory management, thus, heavily improving the extensibility and the usability of the project.

Acknowledgements

I would like to thank my supervisor Doc. Dr. Ing. Dušan Kolář for his help and multiple consultations. I would also like to thank Ing. Martin Dráb for his guidance and advice with Windows kernel development.

References

[1] Rajeev Nagar. *Windows NT File System Internals: A Developer's Guide*. O'Reilly Media, 1997.

[2] CNSS Secretari. *CNSS Policy No. 15, Fact Sheet No. 1* , 2003. [Online].