TRANSPARENT ENCRYPTION SOLUTION FOR END-POINT DEVICES

TRANSPARENTNÍ ŠIFROVÁNÍ PRO KONCOVÁ ZAŘÍZENÍ

MASTER'S THESIS
DIPLOMOVÁ PRÁCE

AUTHOR **Bc. DAVID POŘÍZEK**
AUTOR PRÁCE

SUPERVISOR **Doc. Dr. Ing. DUŠAN KOLÁŘ,**
VEDOUCÍ PRÁCE

BRNO 2019

Brno University of Technology
Faculty of Information Technology

Department of Information Systems (DIFS) Academic year 2018/2019

Master's Thesis Specification

22055

Student:	**Pořízek David, Bc.**
Programme:	Information Technology Field of study: Information Systems
Title:	**Transparent Encryption Solution for Endpoint Devices**
Category:	Security

Assignment:

1. Analyze current implementations of transparent encryption used in practice on endpoints. Focus on Windows and multi-platform solutions.
2. Propose a solution of a transparent encryption system for external devices consisting of a kernel driver and an encryption architecture. Base the design of the encryption architecture on already existing solutions analyzed in (1). Use standardized encryption algorithms and standardized formats of encrypted file envelopes.
3. Implement the solution proposed in (2) for Microsoft Windows platform.
4. Verify whether the implementation meets security expectations defined in (2) and Safetica requirements.
5. Propose and discuss potential extensions (for example, an outgoing e-mail attachment protection), future development and multiplatform possibilities.

Recommended literature:

- RUSSINOVICH, Mark E., David A. SOLOMON a Alex. IONESCU. Windows internals Part 1. 6th ed. Redmond, Wash.: Microsoft Press, c2012. ISBN 978-0735648739.
- RUSSINOVICH, Mark E., David A. SOLOMON a Alex. IONESCU. Windows internals Part 2. 6th ed. Redmond, Wash.: Microsoft Press, c2012. ISBN 978-0735665873.
- CATLIN, Brian, Jamie E. HANRAHAN, Mark E. RUSSINOVICH, David A. SOLOMON a Alex IONESCU. System architecture, processes, threads, memory management, and more. Seventh edition. Redmond: Microsoft, [2017]. ISBN 978-0735684188.
- NAGAR, Rajeev. Windows NT file system internals: a developer's guide. Sebastopol, Calif.: O'Reilly, c1997. ISBN 978-1565922495.
- File Encryption Driver Development with per Process Access Restriction. Apriorit [online]. 01 April 2016 [cit. 2018-06-16]. https://www.apriorit.com/dev-blog/371-file-encryption-driver-development-with-per-process-access-restriction

Requirements for the semestral defence:

- First two items, and initial parts of the item 3.

Detailed formal requirements can be found at http://www.fit.vutbr.cz/info/szz/

Supervisor:	**Kolář Dušan, doc. Dr. Ing.**
Head of Department:	Kolář Dušan, doc. Dr. Ing.
Beginning of work:	November 1, 2018
Submission deadline:	May 22, 2019
Approval date:	October 24, 2018

Abstract

The goal of this thesis is to propose and implement a transparent encryption solution for the Microsoft Windows platform. The solution should be able to integrate with the data loss prevention (DLP) product and extend it. The implementation utilizes the Microsoft File System Minifilter Driver framework, which allows it to monitor and modify access to files on an external device or a disk in a running system. Protected files are encrypted in the background in order to not interfere with user's work. The driver ensures, that the user is always working with decrypted data. Furthermore, an external application will be developed, which will allow the user to access the protected files even in a network without the DLP product active.

Abstrakt

Cílem této práce je návrh a implementace řešení transparentního šifrování pro platformu Microsoft Windows. Řešení by mělo být propojitelné s produktem prevence proti úniku dat (DLP) a rozšiřovat jej. K implementaci byl využit framework Microsoft File System Minifilter Driver, s jehož pomocí je možné sledovat a upravovat přístup k jednotlivým souborům na externích zařízeních nebo discích za běhu systému. Soubory jsou zabezpečeny na pozadí tak, aby uživatel nebyl neovlivněn při práci. Ovladač zajišťuje, že uživatel vždy pracuje s rozšifrovanými daty. Dále bude také vyvinuta externí aplikace, která umožňuje uživateli přistoupit k zašifrovaným datům, aniž by musel být v síti, kde DLP produkt běží.

Keywords

transparent encryption, encryption header, minifilter, file system, driver, windows driver model

Klíčová slova

transparentní šifrování, šifrovací hlavička, minifiltr, file system, ovladač, windows driver model

Reference

POŘÍZEK, David. *Transparent Encryption Solution for Endpoint Devices.* Brno, 2019. Master's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Doc. Dr. Ing. Dušan Kolář,

Rozšířený abstrakt

S rostoucím počtem služeb, které poskytují pohodlný způsob, jak přistupovat a sdílet data na dálku, se stává ochrana firemních citlivých dat stále složitější. Počet případů, kdy dojde ke ztrátě nebo úniku citlivých dat, které způsobí nenávratnou škodu firmě také roste. Ve větší části případů, jsou na vinně zaměstnanci daných firem, i když by se mohlo zdát, že ochrana proti externím útokům je důležitější.

Tento fakt vedl ke vzniku nového odvětví, které se zabývá právě ochranou dat uvnitř firmy a snaží se zabránit jejich úniku. Jelikož je většina firemních dat uložena v digitální podobě, většina firem zabývajících se tímto odvětvím, poskytuje softwarové řešení, které monitoruje operace uživatele a při detekci operace, která by mohla zapříčinit únik dat mimo firmu, ji zablokuje.

Tato práce se snaží výše zmíněného trendu využít a popisuje návrh a implementaci řešení, které by data chránilo bez ohledu na operace uživatele. Cílem je zabezpečit data na tolik, aby při potenciálním úniku tato data nebyla dostupná neautorizovanému člověku a tím rozšířit již existující DLP řešení. Cílem práce je tedy nejen implementace nového řešení transparentního šifrování, ale také integrace s existujícím DLP produktem.

Teoretická část je tvořena ze tří hlavních částí. První část popisuje řešení prevence úniku dat (DLP) z obecného hlediska a vysvětluje základní pojmy týkajících se tohoto tématu. Dále je zde provedena analýza rozšířenějších DLP řešení, kde je každé analyzované řešení popsáno hlavně z hlediska přístupu, jaký používá pro šifrování. Na závěr této části jsou jednotlivé přístupy k šifrování shrnuty do tří kategorií a rozebrány jejich výhody a nevýhody v rámci těchto kategorií.

Druhá teoretická část se zabývá návrhem řešení této práce. Jsou zde definovány požadavky na řešení, spolu s předpokládaným procesem používání a popisem očekávaných uživatelů. Poté je zde vysvětleno transparentní šifrování a všechny pojmy s ním související. Je zde také popsán proces, který je potřeba k tomu, aby transparentní šifrování bylo použitelné a funkční. V poslední podkapitole je popsán návrh šifrovací logiky a šifrovací hlavičky, která je použitá pro uchování stavových informací.

Třetí teoretická část je nejrozsáhlejší a popisuje strukturu systému Windows z pohledu ovladačů a souborových systémů. Jelikož je řešení implementováno jako transparentní šifrování uvnitř minifiltr ovladače, bylo pochopení souvislostí v rámci ovladačů souborových systémů naprosto nezbytné pro vytvoření návrhu a implementaci práce.

Výsledné řešení se skládá z několika částí – transparentní logiky, šifrovací logiky, externí uživatelské aplikace a aplikace s grafickým rozhraní pro nouzové rozšifrování souborů. Transparentní logika je implementována jako minifiltr ovladač. Každý takovýto ovladač se nejprve musí přihlásit do zásobníku ovladačů, které jsou zachyceny nad jednotlivými disky. Díky tomu dokáže zachytit obsah souborů uložených na daných discích ještě dříve, než se dostane k uživateli. V momentě, kdy do ovladače přijde požadavek na přečtení nebo zápis obsahu, ovladač se podívá, zda je to požadavek na soubor, který by měl chránit. Pokud ano, tak provede rozšifrování nebo zašifrování obsahu. Nicméně, uživatel vždy uvidí rozšifrovaný obsah a se souborem bude schopný pracovat bez jakékoliv změny. Jakékoliv šifrovací operace tedy probíhají na pozadí uvnitř ovladače a uživatele nijak neovlivňují.

Šifrovací logika je implementována jako knihovna. Je ji tedy možné jednoduše nahradit, za předpokladu, že jsou ovladači poskytnuty ta stejná rozhraní pro komunikaci s knihovnou. V základu je implementován algoritmus AES-256 pro šifrování obsahu i hlavičky a MD5 pro kontrolní součet. Hlavička se skládá z identifikační části, klíče pro zašifrování obsahu souboru a kontrolního součtu. Tato hlavička je uložena před samotným obsahem a ovladačem je vždy při přístupu k souboru přeskočena. Při uložení na disk je samotná

hlavička zašifrována klíčem, který si uživatel zvolil nebo, který byl dodán přes komunikační rozhraní DLP řešením. Součástí řešení je také konzolová uživatelská aplikace, která demonstruje způsob, jakým lze komunikovat s ovladačem. Aplikace umí nastavit uživatelské heslo, přidat novou chráněnou složku, odebrat chráněnou složku a vypnout a zapnout šifrování.

Poslední částí řešení je aplikace s uživatelským rozhraním, která umožňuje rozšifrovat soubory zašifrované ovladačem. Tato aplikace byla přidána do řešení, aby bylo možné pracovat se soubory i v případě, že nelze ovladač nainstalovat. V takových případech může uživatel využít tuto aplikaci, ale jejím použitím dojde k porušení transparentnosti a tím pádem i celkové bezpečnosti citlivých dat. Jedná se tedy o nástroj, který by měl být použit jen v případech nouze.

Kompletní řešení bylo otestováno za pomocí dvou různých typů testů. Nejprve byl testován výkon řešení při přístupu a otevírání souborů o různých velikostech – od 10 megabajtů (MB) do 500 MB. Výsledkem bylo jen nepatrné ovlivnění, jelikož systém k souboru přistoupí většinou ještě dříve, než si jej uživatel reálně vyžádá. Dále byla testována správná funkcionalita řešení. Tyto testy byly provedeny manuálně a jejich cílem bylo zašifrování souboru a následné porovnání zobrazeného obsahu, zda odpovídá originálu. Ve většině případů tento test dopadl pozitivně. V závěru kapitoly popisující testování je pak ukázáno, že řešení splnilo všechny požadavky jak nastavené DLP řešením, tak i požadavky vycházející z návrhu. Na základě těchto testů a ověření je možné říci, že výsledné řešení je funkční a také integrovatelné s existujícím DLP řešením a tím splnilo cíl této práce.

Největší komplikací této práce byla komunikace se správci mezipaměti a virtuální paměti. Jelikož je potřeba při každém požadavku na přístup k souboru upravovat velikost, bylo zapotřebí korektně upravovat parametry požadavku a také obsah souboru, který se posílá v parametru požadavku.

Transparent Encryption Solution for Endpoint Devices

Declaration

Hereby I declare that this master thesis was prepared as an original author's work under the supervision of Doc. Dr. Ing. Dušan Kolář. All the relevant information sources, which were used during preparation of this thesis, are properly cited and included in the list of references.

. . . .
David Pořízek
May 21, 2019

Acknowledgements

I would like to express my thanks to my supervisor Doc. Dr. Ing. Dušan Kolář for his guidance, patience, and dedicated time. I would also like to thank Ing. Martin Dráb for his consultations and ideas which helped me with the general understanding of Windows driver architecture as well as the final design of the driver. Finally, I would like to thank the whole Safetica team for providing me the opportunity to work on this project and namely Zbyněk Sopuch who has helped me with the specification of the design.

Contents

Chapter 1

Introduction

With growing number of services which allow convenient access and storage of data, such as clouds or remote access, it becomes increasingly more difficult for companies to protect their business secret and other sensitive data which are private and valuable to the company. Incidents, where a considerable amount of sensitive data was leaked, become more and more common. These incidents often cause irreparable damage to the affected companies and in some cases even harm the people who entrusted this company with investments, data or other information. External intruders are not the only cause of them, but quite the contrary. In more cases, the incidents are caused by the company itself or its employees either knowingly or unknowingly.

The necessity to contain and resolve this issue lead companies to reach out and accept help from solutions which focus on protection and prevention of data leaks. Since majority of sensitive data is stored in digital form, these solutions primarily develop software applications and methods which help prevent and protect data from leaking outside of the company premises.

The thesis takes into account the aforementioned trend and aims to solve the issue from a different angle. While preventing the leak itself is sufficient in most cases, a data protection logic which would ensure that data are secured even outside company's network would cover cases where prevention is not enough. The goal of this thesis is to propose and implement a transparent encryption application which would enhance a data loss prevention (DLP) solution.

Chapter 2 introduces general concepts of DLP solutions and describes typical modules which are often present. In order to propose a complete and functional solution, the thesis looks at existing models used in different DLP products and analyses them in section 2.4.

The following chapter 3 presents the proposed design. It defines the main requirements for the solution in section 3.1. The architecture of the driver and the encryption header is described in sections 3.2 and 3.3 respectively.

Basics of driver development are described in chapter 4 focusing on the development on the Microsoft Windows platform. It introduces different models available to the developer and describes basic driver architecture and concepts important for drivers. Lastly, it describes frameworks which are provided by Microsoft to better support the development.

Chapter 5 expands on information provided in chapter 4 and focuses on development of file system drivers. The beginning of the chapter discusses the file system architecture in Microsoft Windows platform. Later in the chapter, the core system resource managers are presented which take part in managing file access.

Chapter 2

Data Loss Prevention

This chapter talks about approaches that are utilized within different DLP products. Since the DLP market is becoming quite saturated, section 2.4 focuses mainly on established companies and innovative solutions. Each company will be presented briefly along with the scope of their product, and their approach to encryption will be shown in more detail.

2.1 What is DLP?

DLP is a set of tools and processes which helps to ensure that sensitive data are not lost, misused or accessed by an unauthorized user. In order to distinguish sensitive data, most DLP software utilize multiple data classification approaches. Development is usually driven forward by regulatory compliance for example GDPR[1] and HIPAA[2]. To help DLP software identify sensitive data and generally dangerous actions, administrators have the option to introduce policies. These policies should define safe or unsafe flows of data inside the network. [19]

Sensitive Data

Sensitive data are defined by the security standard SEC-501 as any data that would influence interest of an involved party, course of the organization, or privacy of individual in case their privacy, integrity, or availability was to be compromised. The rate of sensitivity is directly proportional to potential damage done to the company if these data were to be leaked. [1]

Furthermore, DLP solutions usually implement their own algorithms which categorize data and mark data as sensitive. Generally speaking, there are two common approaches to detecting sensitive data – context and content approach. The context approach analyses only the process, which lead to obtaining said data. For example, an export of data from Microsoft Excel, working with a previously marked sensitive spreadsheet file. On the other hand, the content approach does not take context into account. It simply scans files, which it suspsects might be sensitive and analyses only the file's content. For example, a file containing social security numbers would be deemed as sensitive.

[1]https://eur-lex.europa.eu/eli/reg/2016/679/oj

[2]https://www.govinfo.gov/content/pkg/PLAW-104publ191/html/PLAW-104publ191.htm

Policies

Policies are the core object of every functional DLP product. They allow administrators to define complete rules and guidelines on how business data can be shared outside the company. These rules are stored digitally inside DLP solutions which enforce them. The DLP solutions then make sure that these rules are followed properly by all the users.

Furthermore, sometimes companies are required by law to follow certain regulations and standard. Defined policies can be used as means to ensure that these regulations are being followed properly as well as a proof in case of a review. A recent example of this is the GDPR regulation.

Policies can be applied through two different processes – content and context scan. Solution can scan the content of the file and decides whether there are any sensitive data or not. If there are, it may for example tag the file for other modules, so that they can block the file from leaving the company. Contrary to this approach, the context scan is not interested in the content of a file, but rather in the operations which lead the file to the state it is in right now. Therefore, it takes into account multiple actions and not a single one.

Data-at-rest

Any data that are stored physically on a storage medium are described as data-at-rest. DLP solutions must be able to seek out specific files wherever they may be stored. If there are such files present, it opens them and scans them for sensitive data. [13]

Data-in-use

Data-in-use refers to files and other data containers which are currently opened and being accessed. The main goal is to monitor the movement of the aforementioned data. It is usually accomplished through agents, which closely monitor separate applications and report back to the management server.

Thanks to being so closely connected with applications, the agents are able to react to data changes based on the defined policies and rules. Agents acquire policies from the server and have to process them in order to apply them in reaction to data modifications. Some policies may be fairly complex and too difficult to process on an endpoint. In that case, some compromises are required in order to process any policy without disrupting the user's workflow. [13]

Data-in-motion

Data-in-motion describe any data which are being transferred from one point in the network to another. A file transfer across the network usually consists of sending multiple packets containing chunks of the original file. The DLP solution must be able to properly identify different packet streams and reconstruct the file in order to analyse its content, similar to the analysis of data-at-rest. [13]

2.2 Security Points of Deployment

There are two main places at which DLP solutions usually get deployed – an endpoint and a network. Each of them has different advantages and disadvantages. A complete

DLP solution deploys most of the time on both, thus, providing a combined security and advantages of both. [18]

Endpoint

A solution deployed on endpoints monitors and controls data access. They focus on protecting data-at-rest and data-in-use. Their main advantage is being able to protect the endpoints even outside the company's premise. While the solution is able to apply and process policies by itself, it typically doesn't store them locally, but rather receives them from a central server, located in the company's network. [18]

Network

Network protection usually analyses the network traffic inside the company and is able to block, allow, or secure outgoing data. This includes encryption of data, which should not be read by an unauthorized person. These solutions usually have probes, which monitor and analyse multiple points in the network. The probes send the collected information to the central server located at the egress of the company's network. The server then decides based on active policies, how to proceed. The network solution primarily protects data-in-motion. [18]

2.3 Solution Modules

This section presents the usual modules introduced in DLP solutions. The following information is based on the analysis later in this chapter in section 2.4.

Agent

DLP solutions need to have a connection to each endpoint they manage. This is typically done by running an agent application on each endpoint which communicates with the management center and receives commands from it. The agent monitors and logs file accesses and scans the file content. It usually attempts to run hidden from the user.

Management Center

The main purpose of the management center is to manage agents and provide a centralized server which the agents can use to send collected information. The center usually takes care of auditing and providing users (usually administrators or managers) with summarized view of all the information collected by the agents.

Probe

A network probe is used for monitoring network traffic. It is typically located on the company's gateway and scans all passing traffic going outside the company's network. Furthermore, some probes are also able to prevent data leaks from happening upon their detection. Since the whole traffic is being monitored, the probe is able to protect everything that is related to network including emails, clouds, and file shares.

2.4 Analysis of Existing DLP Solutions

I would like to point out, that all the information presented in this section is taken from materials available to public and my understanding of them. Therefore, some information may be inaccurate or simply wrong. This fact is not a big issue for a collective analysis like this, because the thesis is primarily interested in general concepts that the solutions utilize. Nevertheless, I still feel it is important to warn readers about this as to not interpret the analysis as definite facts.

The analysis is based on DLP solutions available on multiple platforms. All analysed solutions provided a support for Windows and MacOS. Linux platform support was more rare, but still present in a couple of solutions.

Endpoint Protector[3]

Endpoint protector focuses mainly on data encryption on all the leading platforms. While it is able to monitor and block certain actions, it doesn't seem to be the main goal of the solution. It uses a content aware protection and is able to protect data at rest as well as data in motion. Data at rest can be protected by using a discovery function, which is able to detect unprotected sensitive files and immediately perform a defined action, for example a file encryption.

The solution is provided in three different appliances. Virtual appliance, hardware appliance, and cloud-based appliance. This is an unique approach, since majority of DLP products tend to focus on only one of the mentioned methods.

The most interesting part is definitely the Device Control. It is able to monitor, control, and block USB storage devices and peripheral ports. The basic workflow starts with detection of vulnerable external device. It is than scanned for sensitive data and if there are any unencrypted sensitive data, it encrypts them at once. It is possible to whitelist certain external devices and have them excluded from this process and also to entirely block an external device from being mounted. Another step is executed, when there are any data copied onto the external device, they are encrypted just in time (JIT). Unfortunately, there isn't enough information available to know for sure how is the software part of the product implemented.

Check Point[4]

Check Point provides a complete DLP solution for Windows and MacOS. It is able to make decisions based on content, users, and processes. It scans the defined files and tags them. These files are then protected and prevented from leaving the organization. Customers can manage the product in two different ways. Central policy management allows admins to control all clients in one centralized place. It is possible to create policies which identify stored data and tag them based on that policy. There is also an event management which is used for real-time monitoring and log reporting.

Another feature is a network protection. It works by being active on an existing gateway, monitoring any network traffic going through it and analysing wide range of transport protocols, thus, protecting data in motion. The protocol detection works in conjunction with applications to make sure it is properly filtering data.

[3] https://www.endpointprotector.com/products/endpoint-protector/features

[4] https://www.checkpoint.com/products/endpoint-policy-management/

Encryption logic is also part of the product. It provides full-disk encryption for endpoint disks and external devices. The algorithm used is AES-256. Moreover, there is also an option to block certain devices or ports from being accessed at all. Admins can choose to receive notifications instead of denying the access.

On top of that, there is a ransomware protection which uses behavioral analysis to detect any unwanted file modifications. Also, a zero-day attack prevention is available. It uses advanced heuristics to detect and block any unknown phishing sites.

Symantec Endpoint Encryption[5]

Symantec solution provides protection of all the important channels on which data can appear – Endpoint, Network, Cloud, and Storage for Windows, MacOS, and Citrix XenDesktop platforms. It is able to monitor and notify administrators when there is a possibility of data leak or a breach of a policy.

Endpoint protection consists of discovery and prevention functions. Discover can scan local storage and look for sensitive unsecured data. Upon discovering the data, it can perform certain actions, for example, encrypt, delete, or move. The network part provides monitoring features and prevents sensitive data from leaking to email and websites. The storage protection is similar to endpoint protection but on top of discovery and prevention provides also an API, which developers can use to implement custom responses. Lastly, there is also a cloud protection service. It focuses mainly on the well-known cloud providers.

As mentioned earlier, one of the possible responses to policy violation on an endpoint is data encryption. While this is true for external devices, the solution implements a full-disk encryption for the local hard drive. The files copied to external device are encrypted JIT. Furthermore, an encryption process similar to the external devices occurs when there are sensitive data detected being moved to an external storage or a cloud.

Verdasys Digital Guardian DLP[6]

Digital Guardian DLP is able to protect endpoint, network, and clouds on Windows, Linux, and MacOS platforms. Network and cloud protection work as a software solution and it is a separate product from the endpoint protection. We are primarily interested in encryption function in the endpoint solution, therefore only that part is mentioned.

Similarly to the Symantec solution, Verdasys is able to define actions which are performed upon detection of sensitive data, such as an encryption. There is a specific function when it comes to external devices. The user can choose to use either portable or non-portable encryption. The non-portable one is used to transport data only between machines which run the Verdasys solution. The portable encryption method is used to transport data outside the company's premise while ensuring data are properly secured. The public material is not very specific on whether it uses an open-source encryption method or there is an external application which can be used outside of the solution's network.

[5] https://www.symantec.com/products/data-loss-prevention

[6] https://digitalguardian.com/products/endpoint-dlp

CA Data Protection[7]

CA Data Protection DLP solution protects data at rest, data in motion, and data in use. It secures endpoints by distributing agents, which then report either to gateway server or directly to central management console. Network is secured by a software solution that is able to control number of different protocols including SSL traffic that is decoded locally. External file storage and email attachments are protected as well.

The logic primarily focuses on encrypting data, which are leaving the endpoint. There are agents running on every endpoint and reporting to the gateway server. The server scans data going through it and decides, whether there is any action required based on defined policies. For example, it can block outgoing data transfer and store data being transferred on the server in an encrypted form to be reviewed by the admin at a later time. It uses AES-128 to encrypt any captured data.

The solution is also able to detect data being transferred to external device and encrypt them JIT. This action is controlled by their Client File System Agent, which most likely works as a file system filter. It is also able to transparently encrypt files being synchronized through cloud services.

Sophos - SafeGuard Encryption[8]

Sophos doesn't offer the usual DLP functionality, such as user monitoring and action control. They rather focus on securing user's data and making sure that they are secured at any point while inside the company's premises. This includes endpoint, network, and email protection. Since they primarily protect data, encryption logic is an important part of the product. They are able to protect Windows and MacOS platforms.

There are several encryption methods used in each of the products, however we are mostly interested in the endpoint protection. It utilizes two different approaches to encryption. The first one is a full-disk encryption used for encrypting local hard drives. The second one is a transparent encryption which is used to secure external devices even when they are outside the Sophos's network.

2.5 Encryption Approaches

As we can see from the analysis above, there are several methods used in practice when encrypting data. The following sections attempt to summarize the approaches mentioned into three groups and describe their common advantages and disadvantages. While this list is not exhaustive it should cover the most common approaches and provide enough insight into each of them to come to a conclusion as to which of the approaches is the most suitable to be used in the solution of this thesis.

Full-disk Encryption

This approach is the most used one. It is based on encrypting the entire disk and/or the folder and, thus, securing any data stored within it. The approach is suitable for protecting data from external threats. This means, that the full-disk encryption does not see its user as a threat and trusts him/her with the data completely.

[7] https://www.ca.com/us/products/ca-data-protection.html
[8] https://www.sophos.com/

The main advantage of this implementation is that it is easy to implement and get going. It is also fairly fail-proof, because it is generally used to encrypt more (e.g. the entire disk) rather than less. This makes it more robust then the other approaches.

The disadvantage is the trust in the user. Conceptually speaking, this goes against what any DLP product is trying to achieve. The main goal of DLP is not to protect the data from external threats, but rather to protect the data from user's mistakes. To expand on this, DLP is trying to prevent a user from leaking any sensitive data, by protecting the data from him/her. This means, that by giving the user full control over the data, it would make the product very prone to failure in its core goal.

That being said, the full-disk encryption is still a viable solution for protecting external devices' data, from being accessed by an unauthorized person, for example, when stolen. Despite that, this work aims to extend an existing DLP product, therefore, this approach is not suitable.

Server-based Proxy Encryption

The server-based proxy encryption works by securing data, which are about to leave the company's premises over a network. As the name suggests, the idea behind the server is, that it assumes the role of company's gateway router and, thus, it is able to monitor all data that are being routed outside the company's network. This approach is very good at protecting data-in-motion on the network channel. It is usually implemented as a hardware appliance, which is physically connected to the company's network as a gateway.

There are several advantages to this approach. The server is able to monitor, filter, encrypt, or block any outgoing packets containing sensitive data. The encryption process is done on the server, so there is no performance impact on any of the endpoints. Futhermore, protecting a storage to cloud service is possible. This is usually quite problematic for the other approaches.

The main disadvantage is, that it is able to protect only one channel – network. Data leaving the company's premises on any other channel (for example, physically on an external storage) are neither protected nor detected. This is usually the reason, why most of the companies opt to use another solution alongside the server-based one.

Since the goal of this work is to be able to protect data on external devices, this approach is not usable.

Transparent Encryption

The goal of transparent encryption is to protect the defined data, without affecting the user's work. The idea behind this approach is similar to the full-disk encryption. The transparent encryption logic encrypts the data on a disk. The difference here is that the user is virtually normally working with a file, but if he/she attempts to, for example, copy a file to an external device, the file will stay encrypted. Of course, the user will not know that, because the encryption is working transparently. But if he/she removed the external device and plugged it into a different computer without the transparent encryption, the file be unreadable.

The advantage of this approach is that it gives a user access to the decrypted data only when necessary. It allows him/her to work with the files, but does not allow him/her to share the protected files with others, unless they are also running the transparent encryption solution. This allows the solution to ensure, that no files are accessed by an unauthorized user. Furthermore, any overhead can be eliminated, if all protected files are encrypted just-in-time (JIT), when they are first accessed.

Although there is plenty of advantages to this approach, there are still several disadvantages. The main disadvantage is, that it is fairly complicated to implement it properly. The idea behind the implementation is to basically „fake“ read and write information, so that the file system and any application that is trying to read a file can access a different content, then is actually stored on the disk. On top of that, usually such solutions have to rely on implementation details of file systems and user applications to get them to work properly.

Based on the description above, I decided to implement the transparent encryption solution.

Chapter 3

Proposed Transparent Encryption Solution

This chapter introduces the proposed transparent encryption. The first section 3.1 defines the requirements and expectations of the encryption solution. The approach chosen for this thesis is described in section 3.2. Finally, the encryption header as well as the algorithms are shown in section 3.3.

3.1 Requirements and Expectations

This section defines the requirements and expectations of the encryption solution. The specific requirements are described in the first subsection. The following subsections then describe the targeted end-user and the expected work flow respectively.

Encryption Solution Requirements

There are multiple requirements for a successful integration with a DLP solution, which should be considered during the design:

1. An option to define which data should be protected.
2. Data must be protected even outside the company's premises.
3. Data must be accessible and readable outside the company's premise.
4. The protection algorithms used must be strong enough to resist simple cracking methods.
5. The encryption solution should be easy to deploy and use.
6. The solution should be able to easily integrate with the existing DLP product.
7. The solution should work hidden to the end-user.

There are two commonly used options to define files which should be protected. The first one is by specifying protected folders. Then, any file that is copied into the folder will be protected automatically. The second one is a secondary file stream. It would be possible to write an information into the secondary file stream to notify the encryption logic that this file should be encrypted.

By using a transparent encryption, it can guarantee both second and third point of requirements at the same time. Since files would be encrypted JIT, any protected file will be automatically encrypted and only readable by a user if he/she accesses the file inside the company's network. Furthermore, by implementing a custom encryption logic, an external application can be introduced to provide a proper access to the encrypted data. This also means, that a secondary user-defined password would be usable in cases where the user authentication cannot be verified through the DLP solution deployed in the company.

Appendix B shows a figure which defines use cases that should cover the requirements mentioned above. There are two actors – an administrator and a user, because the person who is configuring the encryption solution will not be the same person who will be using it.

End-users Description

This solution is aimed at users who use the computer at work on daily basis and are a potential source of data leak whether intentional or unintentional. That is, the features it provides, are to be used by the user. There are two different groups of users who could benefit from it. The first group, are all users, who are unaware of the potential damage they may cause and their data should be protected automatically to prevent any possible leaks. The second group, are users, who are aware of potential leaks, but choose to use the features in order to protect their data, for example, while they are being transferred between two destinations and the transfer path itself is not secure.

Expected Workflow

It is expected that the more common use case of the encryption solution will be on-demand with external devices instead of protecting a local storage path and running constantly on a system. The reason for this is that usually full-disk encryption is sufficient for situations like these. The encryption solution does not attempt to replace it, but rather extends it by adding new features which would not be available in a full-disk encryption approach.

There are two different workflows worth noting. The first is a user working with a transparent encryption set-up on a local disk:

1. An admin deploys the encryption solution and sets a protected folder on a local disk alongside the DLP solution.
2. A user logs into his/hers Microsoft Windows account and a DLP solution is automatically launched – launching the transparent encryption as well.
3. The transparent encryption detects the added folder and encrypts all the content in it, while the user does other work.
4. If the user attempts to access any data inside the folder before it has been encrypted, the file he/she is attempting to access is prioritized and encrypted before the user receives an access to it.
5. Once all the files have been encrypted, the folder structure is displayed to the user identically as before.
6. The user continues his/hers work and every file that is added to the secured folder is encrypted and displayed to the user in decrypted form.

7. If the user attempts to copy the file to an external device and bring it home, he/she will be able to access only the encrypted form of the file.

The second workflow describes a user transporting data on an external device between two places – one with the encryption solution active and the other without it.

1. An admin deploys the encryption solution and configures it to encrypt only on-demand.
2. A user logs into his/hers Microsoft Windows account and a DLP solution is automatically launched – launching the transparent encryption as well.
3. The user realizes, he/she will need to bring data outside the company's premises to showcase something to a customer. This would not be typically possible without breaching the DLP policies and rules and copying the unsecured data to an external device.
4. The user copies the files to an external device and is prompted by the running DLP solution, whether he/she will need to use these files outside the company's network.
5. If he/she selects yes, a password prompt will pop up. He/she will be asked to enter a password, which will be used to authorize the user on the other endpoint. In case he/she selects no, the data protection will not be modified, and if he/she attempts to access the data on the other endpoint, it will not be possible.
6. The user brings the external device to an unsecured location and runs an external application which is present on the external device. The application prompts him/her for the password he/she inputs before and if the password is correct, the application loads the transparent encryption and allows the user to access and read the data in the encrypted files.
7. The transparent encryption becomes inactive in the system once the external device is removed.

It should be noted, that in both workflows the transparent encryption has been successfully hidden from the user. The only time only he/she would be able to realize an encryption is present, is during a potential data leak or when he/she would need to use a specific feature the solution provides.

The deployment of the transparent encryption should be seamless and required only initially. After that, the encryption solution should be able to run on its own, while not disturbing the user unless specifically requested.

The second workflow is more important and the main focus of the encryption solution. While local disk storage and its protection is an interesting topic, it can be replaced by full-disk encryption as mentioned above. Therefore, the design focuses on external devices and how to protect them effectively.

3.2 Transparent Encryption Logic

This section describes the core ideas of the design and talks about them in greater detail. The first subsection describes the reasoning behind choosing the minifilter approach, while the other ones expand on that and discuss the data flow, file view provided to the user, and external communication and settings interfaces.

Filesystem Minifilter Approach

Based on the analysis provided in the chapter 2.4 and requirements defined in the chapter 3.1, I have chosen to implement a minifilter driver. The main advantages provided by this approach are the following:

- Transparent encryption applied only to a part of a logical storage.
- An encryption logic which is easily extensible and not limited to use with external devices.
- Just-in-time encryption – files are encrypted only when they are copied to the protected folder.
- Possibility to change encryption algorithms when needed.
- Open-source encryption header allows other DLP solutions to extend their support to the presented approach.
- Complete control over file movement and access.
- It is an expected functionality for a minifilter driver and in some parts even encouraged by Microsoft by providing samples of such functionality.

There are number of concepts this approach introduces that should be addressed. Probably the most important one is that even though the minifilter approach provides freedom and gives developers higher control over files, any potential error could cause the system to become unstable or crash. The fact that kernel drivers have to be developed with more caution and must be thoroughly tested is true for all kernel drivers. Luckily, Microsoft is well aware of this and provides multiple tools to verify and test whether driver's behavior is correct.

Another concept to keep in mind is that the driver runs in kernel-mode. This means, that it has access to system structures and functions. It would be possible for the minifilter to work with cache manager and essentially take over the cache management. While it is indeed a possible way to approach the implementation, it would also exponentially increase the scope and difficulty of the implementation. While some communication with the cache manager is expected, it is not to the extent of taking over cache entirely.

On top of that, running in kernel-mode has other advantages. For example, it provides an extra layer of security for the driver. The reason being that it is more complicated, to attempt to temper with a kernel-mode process than a user-mode one. Furthermore, it is also protected by the Microsoft driver infrastructure and limitations that Microsoft introduced to improve the security of drivers.

Data Flow

The flow of data is shown in figure 3.1

The following list explains each of the important points during the communication of different system modules in reaction to a request from an application:

1. A user-mode application requests to read data from a secondary storage.
2. The I/O manager receives the request and forwards it to the appropriate file system driver.

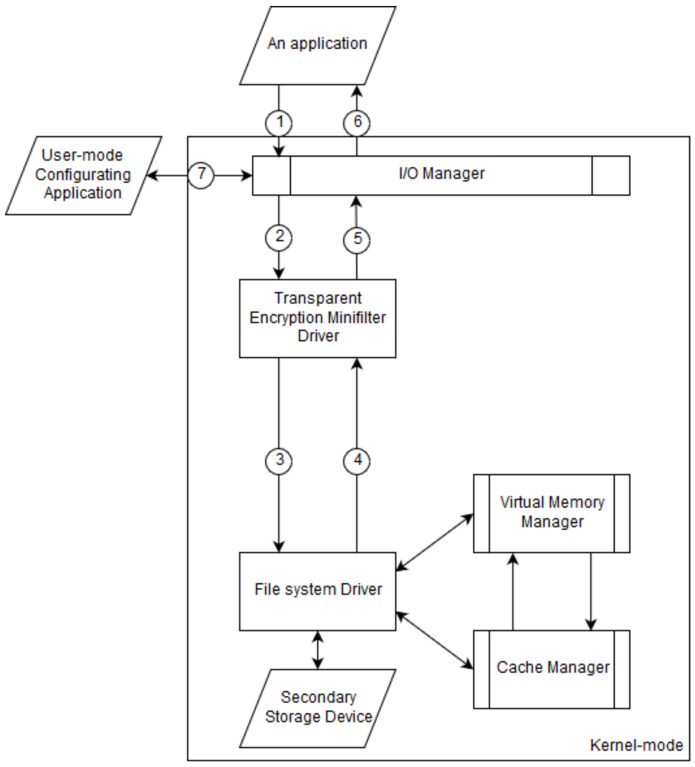


Figure 3.1: Figure which shows the I/O request and data flow.

3. The transparent encryption minifilter receives the request before it reaches the file system, because it is higher in the driver stack. The minifilter stores identifying information about the request and reads the data buffer associated with the request. It then decides whether to encrypt, decrypt, or only modify file sizes based on the request type and contained data. In this point, it is safe to modify data and to adjust file sizes in order to virtualize the file content to cache manager (CM) and virtual memory manager (VMM).
4. The file system obtains or processes the request and returns the resulting data. It communicates with the secondary storage device to obtain data, the CM to cache read or written data, and the VMM to create memory mapping of the file associated with the request.
5. Transparent encryption driver adjusts the file sizes and forwards the processed request back to I/O manager. In this point, we can adjust file sizes as long as it adheres to what the requesting application expects.
6. The I/O manager returns any result back to the requesting application.
7. This is the communication channel provided by the driver to the user-mode, which allows to modify the driver's settings.

Virtualized File View

The main idea behind virtualizing the file view is to provide an illusion that the files a user is currently working with are the original ones, while in fact they have been already modified by the driver (for example, encrypted). Even though the explanation is fairly simple, the implementation can be quite complex. In order to fulfill the requirements presented in the previous section 3.1, the encryption driver must modify the file location as well as its size and content.

The real file location needs to be modified, so that the encrypted files can be hidden from the user in case he/she attempts to access them without the driver running. The file size modification allows the driver to display the encrypted files as if they have not been encrypted and appended with an encryption header.

There are two potential issues with this approach. The first issue is created by the cache manager and the second issue arises during memory mapping of files. Caching of accessed files will happen regardless whether we attempt to block it or not. To reliably prevent files from being cached is impossible in the context of a minifilter. The implementation will need to keep the decrypted data in the cache in order to allow any cached reads to go properly through. On top of that, it will be required to encrypt data associated with a protected file which the cache manager is trying to write through, in order to keep the encrypted data up to date with what the user sees.

The second issue is mapped memory files since it allows a direct access to a content. This means, that if a protected file would have to become memory mapped without driver's interference, the application could only access encrypted data. Consequently, the driver must detect these requests and react accordingly by providing decrypted memory to initialize the memory mapping.

User-mode Kernel-mode Communication Interface

Conceptually speaking, there should be two communication interfaces provided by the driver – an internal interface and an external interface.

The internal interface would deal with communication coming from a DLP solution. The driver should implement the following messages:

- Enable, load, disable, and unload the driver.
- Request a subject key and receive a response containing the key.
- Request to delete a subject key.
- Set a protected folder/file.
- Remove a protected folder/file.

These messages would need to be implemented by a DLP solution in order for the driver to function properly. They are also meant to be used by an administrator, that is why they should be separated from the interface, which would be externally accessible and only provided internally.

The external interface would be used for communication with the user. Only one message would be needed for the communication and that is setting a key for access outside premises. Other than that, the user should not know, that there is a transparent encryption active and does not need to communicate with it in any other way.

External Application

The external application should provide a user with an interface to communicate and use the transparent encryption outside the premises. It should be a standalone application independent of the minifilter.

The user will need to use the application only in cases where an input is required from him/her. This is only the case when he/she needs to set a password to access the data outside a premise. The DLP solution should communicate with the driver and determine, whether there are any external devices which store encrypted files and prompt the user for an input if such files are found.

3.3 Encryption Architecture

A proposed encryption logic can be found in this section. It describes the encryption header, which is the main part of the encryption logic and proposes algorithms to be used on the data and on parts of the encryption header.

The encryption is done by appending an encryption header to any file that needs to be protected and encrypting the content. The header than contains all the information required to decrypt the file, so that the driver does not have to store any session information and can work with already encrypted files.

Encryption Header

The minimal encryption header should consist of a header identifier, a symmetric content encryption key, and a subject key.

The header identifier is a predefined set of bytes of fixed length, which should be as unique as possible. The driver would read this header and based on that determine whether the current file is already encrypted or not.

The symmetric content encryption key is used for encrypting data of a file. It is generated by the driver for each request. The key is then stored in the header, so that the driver does not have to keep the information in the memory.

Lastly, there is the subject key. This key has to be obtained externally from the DLP solution which implements this header. This key is expected to be asymmetric and would be used to encrypt the entire header. This means, that the driver needs to ask for this key before attempting to access an encrypted file and also needs to store it in its memory. Furthermore, the driver also needs to deal with managing the subject keys and reacting to their changes.

A simple checksum should also be added to provide an extra verification that the header or file data were not tampered with or modified.

Encryption Algorithms

Based on the analysis in the previous chapter in the section 2.4, majority of the 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 NSA[1] to protect TOP SECRET information [17], these algorithms should be sufficient for use in the implementation as well.

[1] https://en.wikipedia.org/wiki/National_Security_Agency

Chapter 4

Microsoft Windows Driver Development

This chapter explains the basics and motivation behind developing a driver. Section 4.1 introduces Windows driver model and explains available driver types. Basic structures used during driver development are described in section 4.2. Finally, section 4.3 goes through the frameworks provided by Microsoft for driver developers and explains their differences.

4.1 What is a Driver?

The definition of a driver can be difficult to properly formulate. Simply said, a driver is a software component which enables a communication between the operating system and a device. For example, if an application needs to get data from a device the application has to call a function implemented by the system. The system then calls a function implemented by the driver corresponding to the one called by the application. The whole exchange can be seen in figure 4.1. [7]

Usually, it is not only one driver which participates in the communication chain, but rather multiple drivers which are layered on top of each other. Microsoft Windows system uses a layered design, where each driver forwards a call request to the underlying driver, until it reaches the device. A driver which relays information to the device and manages control messages for the device is called a **function driver**. A driver which only forwards requests (or performs minor modifications and/or monitoring) to other drivers is called a **filter driver**. Similar exchange is described in figure 4.2. [7]

To specify the definition further, drivers don't have to always communicate with any device. It is possible to implement a driver which communicates only directly with the operating system and extracts data from its memory. In such cases, there is an option to write a **software driver**. Software driver runs in kernel memory and has access to all structures exposed by the system. To access these structures in the user-mode, the driver can send a message to the user-mode application containing any data it needs. This process may seem unnecessary, but the user-mode and kernel-mode is implemented this way, in order to prevent user applications direct access to kernel memory. [7]

Software drivers always run in the kernel-mode, since the main reason for writing such a driver is to access system structures and data. On the other hand, device drivers may not require access to any system structures or data. This allows similar drivers to run

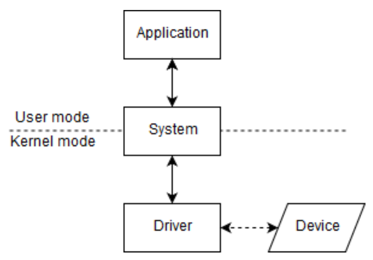


Figure 4.1: A default driver location when participating in a data exchange with a device.

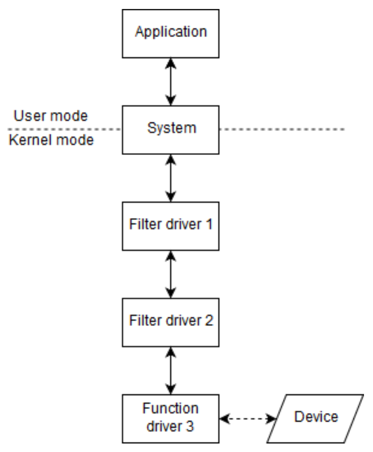


Figure 4.2: A Layered driver arrangement when participating in a data exchange with a device.

in user-mode instead of kernel-mode making them much safer and easier to manage and debug. [7]

Furthermore, filter drivers are divided into upper filters and lower filters. Upper filters are located above the function driver and filter any messages that come through the driver stack. Lower filters are located below the function driver. All filter drivers are optional and may not participate in the communication at all. [7]

Lastly, a driver which coordinates information exchange on a bus and manages it is mandatory in data exchange with a device. Such driver is called a **bus driver**. An example of how the driver stack looks in this case is shown in figure 4.3. [7]

Device Classes

The Microsoft Windows system defines classes and GUIDs which allow developers to manage a specific group of drivers together. The only available classes are predefined and it is not possible to define additional ones. [9] All the classes that system defines are documented online in Microsoft's public documentation archive[1]

Function Driver

A function driver is a main driver which manages a device. It is typically written and provided by the vendor of the device. A specific function driver can service multiple devices if there aren't any other drivers which would be a better fit for the device. [12]

An interface to control a device is usually provided by the function driver. It handles reads and writes to the device as well as power policy. The bus driver encapsulates any communication going directly to device, therefore the function driver doesn't have to be aware of available hardware interfaces, neither any bus timings and synchronization. [12]

The function driver is always loaded by the PnP manager. The manager loads only the best-fit driver. This means, that each driver has to specify, which devices it is meant

[1] https://docs.microsoft.com/en-us/windows-hardware/drivers/install/system-defined-device-setup-classes-available-to-vendors

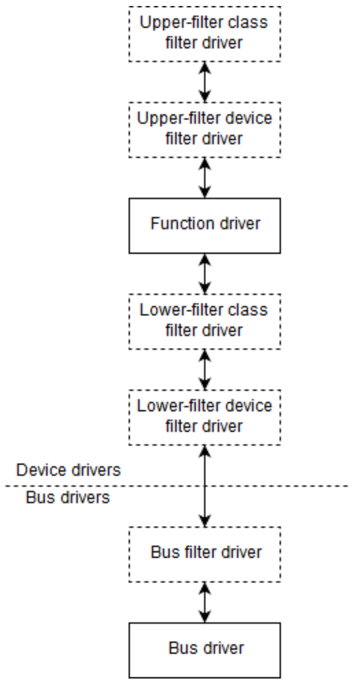

Upper-filter class filter driver
Upper-filter device filter driver
Function driver
Lower-filter class filter driver
Lower-filter device filter driver
Device drivers
Bus drivers
Bus filter driver
Bus driver

Figure 4.3: Different filter drivers in a driver stack.

to work with. The manager then goes through list of devices and its assigned devices and selects the one, which matches the device's hardware identification the best. [12]

Bus Driver

A bus driver manages a bus controller, adapter, or bridge. Drivers for most common buses are provided by Microsoft. The bus driver is required. Every physical bus needs to have an assigned bus driver in order to function properly. There can be multiple bus drivers loaded at the same time for a bus, but always at least one. [10]

The main responsibilities of a bus driver are:

- Enumerate the devices on its bus.
- Respond to Plug and Play I/O request packets (IRPs) and power management IRPs.
- Multiplex access to the bus.
- Generically administer the devices on its bus.

The device enumeration is a process where the driver identifies any new devices on the bus and creates new device object for each of them. There are some operations which the bus driver makes on behalf of the device like accessing the device's registers to physically change its power state. [10]

It is important to note that the bus driver doesn't handle read and write requests. These requests are handled by the function driver mentioned above. Although, it is possible for a device to be used in a raw mode. Then, the bus driver must handle these requests regardless of the function driver. [10]

Filter Driver

Filter drivers are usually associated with other main driver like a function driver or a bus driver. They add value to or modify the device behavior. Filtering a bus driver is typically used to implement proprietary enhancements to a standard bus interface. [11]

Upper-level and lower-level filter driver describe the filter driver's location in the current stack. If it is below the main driver it is filtering for, then the driver is called lower filter. Upper filter is a filter driver which is above the main driver in the driver stack. [11]

Upper-Level Filter Driver Since the upper-level filter drivers are above the main driver in the driver stack they receive any requests before the main driver does. Therefore, they are usually used to further improve a device's capabilities. For example, they could perform additional security checks and stop any malicious requests before they reach the device or monitor what messages the device receives and send the logs back to a user-mode auditing application. [11]

The number of upper-level filters is not limited, and one main driver can have any number of them. The only limitation I found is the size of the driver stack. [11]

Lower-Level Filter Driver The lower-level filter drivers are typically used to modify the behavior of devices. Such modification is possible, because the filter is able to modify any IRPs going to the device from the function driver which is associated with this filter driver. The filter driver can also be used for monitoring the IRPs on the way to the device. [11]

4.2 Driver Architecture

There are several concepts which are crucial for proper understanding of the Windows driver design and development. While the list in this section is not finite, it should cover the most important ones which are heavily utilized by this work.

User-Mode and Kernel-Mode

A processor in a computer running on a Windows system uses two different modes – kernel-mode and user-mode. Applications run in user-mode while system processes and core components run in kernel-mode. The processor is able to switch between the two as needed, depending on what type of code it is running at the moment. As mentioned above, some special drivers are still able to run in user-mode despite being a system process. [6]

When an application is started, the system creates new process for it. It provides the application an allocated private virtual address (VA) and a private handle table. Since the VA is private, it is protected from unintentional or malicious access by other applications. Each application run is isolated from another. Consequently, if an application crashes the crash and its effects are limited to only that application. [6]

In addition to being private, the VA space is protected by the processor mode. When a processor runs in user-mode, it is unable to access any addresses belonging to kernel-mode address space range. Thus, critical system data are protected from a user application accessing, modifying and damaging them. [6]

Comparatively, all code running in kernel-mode shares one virtual address space. This means, that unlike user applications which are isolated from each other, this is not the case for kernel processes such as kernel-mode drivers. Every kernel-mode driver has access to the whole kernel-mode address space and is able to write to or read from any of its addresses. Although, such address must still be legally writable, the write must be properly aligned, it mustn't be locked by synchronization, etc. If any of the kernel-mode drivers crashes, for example, by accessing a locked memory without requesting the lock ownership, it causes a crash for the entire operating system. [6]

Device Objects and Device Stacks

The device tree represents a complete description of all devices currently present on the host machine. It is constructed by Plug and Play (PnP) manager during startup and when a new device is connected. The PnP manager asks each bus to enumerate its devices currently connected. The bus driver creates **physical device object** (PDO) for each of the connected devices. The PnP manager then associates a device node witch each PDO that has been created. [3]

Every device is represented by a **device node** in the Plug and Play device tree. Each device node contains an ordered list of device objects, which is called the **device stack**. Each of these objects has an associated driver. Every device node has its own device stack. The device object is an instance of a `DEVICE_OBJECT` structure. [3]

I/O Request Packets

When a request is sent to a device, it is usually packaged in an I/O request packet (IRP). To send an IRP to a driver, Microsoft provides a function `IoCallDriver` which takes a pointer to a `DEVICE_OBJECT` and a pointer to the IRP. [5]

Typically, when an IRP is sent for processing, it is first sent to the top of a device stack based on the associated `DEVICE_OBJECT`. Every `DEVICE_OBJECT` structure contains a pointer to its associated `DEVICE_OBJECT`. Upon receiving the IRP, a callback is invoked, which is defined by a driver associated with the aforementioned `DEVICE_OBJECT`. [5]

IRPs are self-contained. This means that they hold all the information which a driver could require in order to process the I/O request. The IRP structure contains both information common to all drivers and information specific to the current driver. [5]

Driver Stack

Requests to device drivers are usually sent packaged in an IRP. When sending a read, write, or control request to a device, the I/O manager locates a device node which is associated with the device and sends the IRP to its device stack. There can be multiple device stacks involved in processing one IRP. To process the given request, a driver, which is associated with the current `DEVICE_OBJECT` processing the IRP, is invoked. The driver processes the IRP and either completes it or sends it further down the device stack to be further processed by other drivers. The sequence of all the involved drivers is called a **driver stack**. [4]

4.3 Frameworks Available

Microsoft provides two core frameworks for Windows driver developers. The first and older one is called Windows Driver Model (WDM) and the newer one is called Windows Driver Framework (WDF). The features and capabilities of the framework came a long way since

WDM. Both WDM and WDF are still widely used. WDF is preferred for device drivers and in cases in which we can sacrifice some control over the driver functions. [16, p.68-80]

Windows Driver Model

WDM replaced VxD and Windows NT Driver Model. The goal of this framework was to unify driver models for Windows 9x and Windows NT by defining standard requirements and structures. All the crucial structures for driver developers came from WDM and are described by it. This significantly simplified the driver development process. The WDM is supported by all Windows systems since Windows 98. [16, p.68-80]

Although this framework was an improvement, there were still some issues with it. Namely, managing power policies of devices was unreasonably difficult and even seasoned driver developers had difficult time trying to get it right. I/O cancellation was quite difficult as well. And there is no support for writing user-mode drivers. [16, p.68-80]

Windows Driver Framework

WDF complements the older WDM. It extends it and abstracts away multiple processes and requirements when developing a driver. It consists of two frameworks – Kernel-Mode Driver Framework and User-Mode Driver Framework. [2]

Kernel-Mode Driver Framework

The drivers developed with the use of kernel-mode driver framework (KMDF) are always kernel-mode drivers and are supported by Windows 2000 and later. The KMDF is object-based and built to extend the WDM. Its goal is to reduce the amount of code that is required to write a kernel-mode driver. It manages power policies of devices and handles plug and play, making the driver less complicated. [2]

User-Mode Driver Framework

The user-mode driver framework (UMDF) is quite similar to the kernel-mode one. User-mode drivers do not provide as much versatility and freedom as kernel-mode drivers but on the other hand are less prone to crashes and should never cause a system wide crash. Kernel-mode drivers also have the advantage performance wise. The main advantage of user-mode drivers is that they simplify a lot of development details, such as power policies and device states during multi-threaded processing. Drivers developed using the user-mode driver framework are supported by Windows XP and later. [2]

Chapter 5

File System Driver Development

This chapter defines basic types of file system drivers in section 5.1. Later in the chapter, the concepts and system managers are described. They are important for the file system development, general understanding of the role which file system drivers take, and the flow of requests and communication between a storage and a user application.

5.1 File System Drivers

A file system driver (FSD) manages storing and retrieving of data. It is part of the storage management subsystem. The commercially available local file disk implementations provide the following functionality [15, p.21]:

- Create, modify, and delete files.
- Sharing files and transferring information between them easily through a secured and controlled manner.
- Structuring the contents of a file in a way that it makes more convenient for the application to access.
- Identifying stored files by their symbolic or logical name, instead of using the physical device name.
- Viewing the data logically, instead of a more detailed physical view.

Although, it is a kernel-mode driver, it is quite different from standard kernel-mode drivers. The most notable difference is that it is required that the FSD registers itself as a FSD service to the I/O manager. FSDs also interact with memory manager quite extensively and rely on the services of cache manager for enhanced performance. There are two main types of FSDs which are possible to design, implement and install in the Microsoft Windows environment – Local FSDs and Network FSDs. [15, p.22]

As the name suggests, local FSDs manage data stored on storage devices connected directly to the host. A simplified view of an I/O request and FSD interaction is shown in Figure 5.1. The FSD receives requests to open, create, read, write, read file information, write file information, read directory information, and close files on a storage. [15, p.22]

Network FSDs provide an option to share a locally connected storage over the network. A network FSD consists of two components – the client-side redirector and the server on a node which has the shared disk connected locally. The client-side takes care of issuing

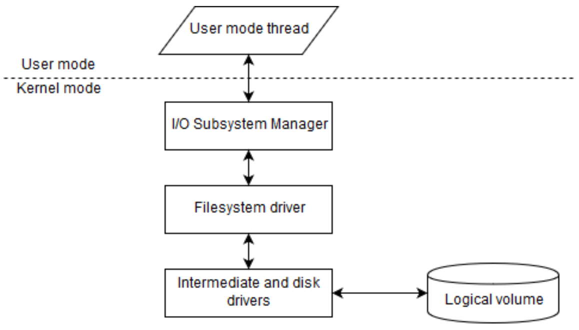


Figure 5.1: Example of I/O request being passed to logical volume.

requests to access data stored on the server and receive response from the server containing the requested information. The server-side must be able to receive these requests, access the local data and provide their content back via the network. [15, p.22]

5.2 File System Filter Drivers

A file system filter driver is an intermediate driver, which is able to intercept requests targeted at recipients lower in the driver stack. Therefore, a file system filter driver is basically an extension or even a replacement to the original recipient of the request, typically the file system driver. [15, p.47-48]

Developing a file system filter driver has been originally quite difficult, despite being a common approach to add range of security features to file access. Therefore, Microsoft introduced a new model called **Filter Manager (FM) Model**. It encapsulates legacy filter driver interface and provides a simplified one with functionality, that is usually required by file system filter drivers. A driver which utilizes the FM is called a **Minifilter**. [8]

5.3 I/O Manager

The I/O manager manages and defines the NT I/O subsystem. All the kernel-mode drivers which control or interface with peripheral devices reside within the subsystem. Since there are new types of peripheral devices being designed and developed continuously, the Windows NT subsystem has to be well-designed and extensible in order to properly accommodate new devices without issues. [15, p.118]

Multiple generic system services are provided by the I/O manager. These services are used by other subsystems to perform I/O or request other services from kernel-mode drivers. Consider the following example: a user application initiates a read request. This request is directed to Win32 subsystem. Now, Win32 subsystem does not directly target the underlying FSD, but rather invokes a system service called `NtReadFile()` provided by the I/O manager. This service then assumes the responsibility of handling the read request and returns back to Win32 with a correct result. Consequently, the `NtReadFile()` also checks the validity of provided data and memory buffers. For security reasons, it is important to not trust some buffers being provided to kernel-mode drivers from user space. The received memory address has to be verified, that it belongs to the calling application and that it has rights to access it. Furthermore, every buffer has to be properly aligned a provide the specified length of allocated memory. If the buffers are validated successfully, it also performs the necessary operations which make the user space address readable inside the kernel-mode. [15, p.118-122]

All the drivers in the system must conform to the I/O model defined by the I/O manager, otherwise they won't be allowed to make I/O requests. As mentioned above, it consists of objects and provided services which access and manipulate these objects. Thanks to this approach, kernel-mode driver developers don't need to concern themselves with technicalities of requesting I/O. Furthermore, it allows the drivers to define a generic behavior for incoming I/O requests disregarding the initiating subsystem. The single I/O model allows kernel-mode drivers to use services provided by each other and consequently allows the kernel-mode drivers to use the layered hierarchy approach. [15, p.118-122]

The I/O manager interacts with both the Cache Manager (CM) and the Virtual Memory Manager (VMM). Virtual block caching is supported thanks to the CM. Interaction with the VMM and file system allows support of memory-mapped files. [15, p.118-122]

5.4 Virtual Memory Manager

One of the goals of VMM is to manage, provide, and optimize an access to the physical memory available on the host machine. On top of that, the provided memory must be protected from access of other applications. This means, that both the code and the data location must be provided in a way that any other application is not able to access it. This applies to the operating system and its memory as well. Furthermore, some applications may require sharing some part of their memory with other applications. This has to be facilitated and supervised by the VMM and allowed only in cases where proper access rights are provided by all the participating applications. The VMM also makes sure the integrity and security of the machine is not violated. [15, p.195-196]

Memory-Mapped Files

Memory-mapped files (MMFs) offers applications an option to access files on disk the same way they access dynamic memory. This means, that it is possible to map any portion of file on disk to memory and once the mapping is complete, accessing the memory is as simple as dereferencing a pointer. The same applies to writing data to memory-mapped files. This approach also provides a way for two applications to share memory, because it is possible to map a file to the memory more than once. [15, p.213] [14]

The memory-mapped files are frequently utilized by the cache manager during read and write requests. It is also possible to explicitly create a mapping of a file by calling functions provided in the WinBase[1] API.

5.5 Cache Manager

The cache manager is a component which closely cooperates with the VMM. It provides and manages a consistent system-wide cache for files. The cache manager also attempts to provide the ability to perform read-ahead on files. Since buffered file requests go through the manager, it is able to track file accesses, analyse the access patterns and provide read-ahead on per-file basis for files which are most likely to be accessed in the near future. Furthermore, it is able to delay writes to disk. This means that it holds changes made to

[1] https://docs.microsoft.com/en-us/windows/desktop/api/winbase/

files in the memory and writes the modified data to disk after some time has passed and it's less likely that the data would be modified again soon. Thanks to this approach ensures better responsiveness for the user application which performs the write. [15, p.243-246]

Cached Read

The idea of using the cache to satisfy a read operation is to improve performance when issuing multiple reads to the same file with the same content. The process starts in I/O manager as any other request. The I/O manager forwards the request to the appropriate FSD. The FSD then recognizes that the request is directed to a file that is opened for buffered access. The request is then passed to the cache manager. Consequently, the cache manager is now responsible for managing and transporting the request data to the user's buffer. [15, p.248-252]

Cached Write

The cached write operation is very similar to the cached read operation. The difference being, that the caching occurs for data, which are provided by the user (or any other requester). This means, that data which are written are being cached at the same time. [15, p.252-255]

File Size Information

There are three different file size values which are used by FSDs [15, p.267]:

- 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.

An important thing to note is that any changes done to one or more of these fields has to be synchronized with other read or write operations and the cache manager has to be informed immediately about the changes. The reason for this, is that the FSD can be bypassed by I/O manager, when it transfers data using the fast path and cache manager. It should also be noted that changes in the `FileSize` are not usually synchronized with paging I/O requests. [15, p.267-269]

5.6 File System Dispatch Routines

Dispatch routines are invoked as a reaction to a request coming to a driver. If the driver defines a routine which corresponds to the type of the incoming request it is called with parameters corresponding to the request. If it wasn't defined, the driver is skipped and the request is passed to a next underlying driver. [16, p.12-30]

Typically, each of these routines has two definitions – a pre-operation routine or a post-operation routine. This means that a driver has a chance to process a request on its way down the stack and also on the way back going up the stack. The pre-operation usually contains all the information about a request that is needed by the driver to complete it (usually the FSD when dealing with files). The post-operation routine on the other hand, contains the requested information obtained by completing the request. [16, p.12-30]

Driver Entry

This is the main routine of a driver. It is invoked by the I/O manager when a driver is loaded into the system. Drivers usually perform following operations in this routine [15, p.390-392]:

- Allocate and initialize memory for global structures.
- Create device object which can accept any request sent to the underlying device.
- Register itself with the I/O manager.
- Initialize function pointers for other dispatch routines.
- Initialize function pointers for the fast I/O path.
- Initialize any timer or synchronization objects.

Every Driver needs to have this routine defined and implemented. Returning anything other than success here means that the whole process of loading the driver will fail and continue with other drivers. Since this is an default initialization routine for drivers, this is where designing boot drivers becomes prone to bugs. If there is a boot driver which is required by the system (or specified as such by the user), failing during the load of a driver will prevent the system from booting. [15, p.390-392]

Create

This routine has to be always called before a read or write operation can happen. Even though the routine is called „create“ it is typically used for opening a file and obtaining its handle. When this routine is invoked, the driver has access to multiple different structure members which specify information about the request and its parameters, such as desired access, user authentication, requested shared access, pointer to the file object, etc. [15, p.397-401]

The pre-operation routine contains primarily the path that to the file which should be created and/or opened. Therefore pointers to file object or file buffer are invalid, since the file doesn't exist isn't opened yet. [15, p.397-401]

The post-operation routine contains valid pointer to file object and file buffer. [15, p.397-401]

Read

This routine is invoked when a driver receives a request to read the contents of a file. The read can be requested in multiple ways. For example, the user may require satisfying the request from cache or on the other hand to never satisfy it from cache. The way the request is made also affects valid parameters in the read request structure. The request might not even come from a user. It is common to receive a request which originated in Cache Manager to read data which are meant to be cached. [15, p.424-426]

The pre-operation contains information about the requested file and details about the request, such as read length, starting address, length of the buffer which receives the result, etc. [15, p.424-426]

The post-operation receives the result and information about the course of the request. Data that has been read is stored as a member as well as information whether the request succeeded and how many bytes have been read. [15, p.424-426]

The read operations are round by FSDs up to a multiple of the sector size of the underlying file storage device. Therefore, when modifying the read data, the new data buffer needs to rounded to a multiple of the sector size of associated device. [15, p.424-426]

Write

This routine gets invoked whenever there is a request to write data to a logical storage. Just like the read request, the write request can be created with different requirements. It may be requested to cache the written data, prohibited to use the cache, required to page the data, etc. Once again, it is possible for the request to originate from a different source than user application. For example, the cache manager can request to write-through its cache to a logical storage. [15, p.437-439]

The pre-operation contains data which are supposed to be written to the destination. It also contains some details about the request – length of the data, destination of the request, etc. [15, p.437-439]

The post-operation only receives information about the result of the request and number of bytes that have been really written. [15, p.437-439]

File Information

File information is requested when a participant needs to know some information about a file. There are number of different types of file information requests where each of the type results in obtaining different information about the file. These types are distinguished by different structures supplied to the request. File information is often requested by other subsystems in order to properly complete other requests like a read or write request. [15, p.476-479]

The pre-operation describes the requested type of file information and ways to identify the target file. [15, p.476-479]

The post-operation contains filled structure which corresponds to the requested type of file information. [15, p.476-479]

Directory Control

There are two different types of the directory control request [15, p.503-504]:

- Request to obtain information about the contents of a directory.
- Request to notify the driver about changes occurring to the files/directories inside a directory.

The first one occurs more often and is generally used when navigating directory structures. The request can be specified further by providing information about what type of data is the requesting actor interested in. The requester may for example want to know metadata of the files in a directory. [15, p.503-505]

The second one is not as common and provides a transparent way for drivers to monitor directory structure and changes to it. Since the directory change notification occurs only when there is a change in a directory, applications are able to provide contents of directories which are always up to date, without having to worry about polling the directories. [15, p.509-510]

The pre-operation specifies whether this request is a change notification or a content information request as well as the target directory. [15, p.509-510] [15, p.503-505]

The post-operation contains the result of the request – either buffer with stored information about the contents of the directory or status whether the notification succeeded in reaching FSD. [15, p.509-510] [15, p.503-505]

Cleanup and Close

There is a slight difference in these two routines. The cleanup routine is invoked for each successful `Create` operation which opened a file object. The close routine is similarly invoked for each `Cleanup` operation, but can be delayed. Receiving a `Cleanup` operation means, that all user references have been closed, but in cases when there are any references still pending, it waits for them to close. Therefore, it is important to understand, that a file object is not closed until there has been a `Close` operation for the specified file object. [15, p.525-526] [15, p.529-530]

Chapter 6

Implementation

This chapter describes the implementation of all the modules in greater detail. Section 6.1 introduces the specifics of the driver and how it achieves the transparency. At first it discusses the steps and the configuration required in order to install a driver and get it running. The transparency process is divided into number of different subsections, where each of the subsections corresponds to one of the phases in the transparency process. Each of the phases is described in regards to the IRPs that it has to modify. The encryption module is presented in the next section 6.2. In this case, the default algorithms are defined here as well as the structure of the encryption header. There are two other extra modules implemented to complete the solution – configuration application and UM decryption application. These modules are described in sections 6.3 and 6.4 respectively. These are relatively small modules, compared to the other ones. These modules were added to the work to make it usable as a stand-alone application.

6.1 Transparency Filter Driver

The driver is implemented as a file system minifilter driver, therefore the language used is C. The core idea behind the transparent approach is to detect a moment, when a user is accessing a file, and change the way the file is displayed to him/her. This way, it is possible for the user to work with the file, even though it is stored in its encrypted form on the disk.

Driver Initialization Phase

The driver is written based on the Minifilter Framework standard. It implements the default initialization function `DriverEntry`. In this function, it calls `FltRegisterFilter` standard API function, which allows the driver to register as a filter driver and to let Filter Manager (FM) know, that it is available. Furthermore, the driver registers different types of callbacks, which are invoked whenever the driver receives IRP of the same type. The driver registers for callbacks of the following IRPs:

- `IRP_MJ_CREATE`
- `IRP_MJ_READ`
- `IRP_MJ_WRITE`
- `IRP_MJ_QUERY_INFORMATION`

- `IRP_MJ_SET_INFORMATION`
- `IRP_MJ_DIRECTORY_CONTROL`
- `IRP_MJ_NETWORK_QUERY_OPEN`

During the `DriverEntry` initialization phase, the driver also creates a new list, which is used to store the protected paths and sets its version to 1. Then, the driver creates new communication port, which serves as the public interface for internal communication with the driver. This is done by calling standard API function `FltCreateCommunicationPort`.

After all the resources have been properly initialized, the driver calls standard API function `FltStartFiltering`, which notifies the FM, that the driver has finished the initialization phase and can now be injected into a driver stack.

Driver Attachment Phase

A filter driver is able to monitor requests made towards a disk, only after it is attached to the given disk. As it was mentioned earlier, every device has its own driver stack. This means that every disk has its own driver stack. Therefore, the filter driver must attach to the correct driver stack in order to filter requests.

The filter driver can either be attached automatically or manually by invoking the filter manager. In both cases, the filter manager asks the running drivers, whether they would like to attach to the current disk. If a driver responds positively, the filter manager then initializes resources for him/her in the driver stack and injects him/her into the stack.

The automatic attachment occurs every time a new storage device is connected to the endpoint. This also means, that the attachment phase will occur at least once during the boot for all the connected storage devices. This is also the moment, when the default file system driver is determined. File system drivers are inquired first and once the default driver has been established, the filter manager then asks the filter drivers.

Per-operation Context and State Information

Thanks to the Windows Minifilter framework, it is possible to define a context, that can be attached to an IRP we are currently processing or to some of the IRP's parameters. This allows us to store information about requests, which can be persistent across different requests towards the same file or even across different files. Similarly, it may be only defined for the current IRP.

In my case, I decided to implement `FILE_CONTEXT` which is defined on per-file basis. This means, that the defined context is available in all requests which are made towards the same file (more specifically, towards the same `FILE_OBJECT`). Therefore, the driver can define, for example, whether the current file has been already protected or not. This also considerably improves performance, because the driver does not have to check a file every time it is accessed to read the encryption header identifier. The entire structure that is stored inside the context is as follows:

```
typedef struct _FILE_CONTEXT {
    PFLT_FILE_NAME_INFORMATION FileName;
    ULONG OriginalFileSize;
    ULONG NewFileSize;
    ULONG EncryptionHeaderSize;
    BOOLEAN OffsetShifted;
    BOOLEAN Encrypted;
    BOOLEAN PathMatched;
    LONG PathArrayVersion;
} FILE_CONTEXT, *PFILE_CONTEXT;
```

Listing 6.1: File context structure definition

Protected Paths and Files

The driver receives information, about which paths should be protected and stores the information in its own path list. Whenever there is a request made towards a file and an IRP is issued, the driver extracts the file's path from the IRP and cross-checks it with the previously mentioned list. If it finds the same path or sub-path, it stores the result in the file context. This considerably enhances its performance, since traversing list and comparing strings, especially, when there is tens to hundreds of requests coming to the driver in a second, is a costly operation.

The list of protected paths can change at any point of processing a request. For this reason, the driver uses a simple versioning system. The version is incremented, every time there is a new path added or removed to or from the list. The version is also stored within the file's context. This ensures, that the driver always knows, whether it should go through its list again, or if it can rely on the information read from the file's context.

First Time File Access

When a file is accessed for the first time, the first IRP type that is issued is `IRP_MJ_CREATE`. Mainly, because the Windows system has to open the file's handle everytime it needs to access it, even for querying information from the file. In order to obtain a handle to the file, this IRP must be issued.

`IRP_MJ_CREATE`

Pre-operation callback first attempts to obtain the path information about the current file that is being requested. If it is successful, the file's path is stored as a string in a callback completion context which is accessible only by the post-operation callback. The callback then returns status, which requests a post-operation to be called once the request is traversing back up the driver stack.

Post-operation callback checks for the completion context and verifies, whether it is dealing with a file that should be protected. If the file should be protected, it creates new file context and attaches it to the `FILE_OBJECT`. Then, it stores the path from the completion context passed from the pre-operation callback in the newly created context. After that, the driver goes through its list of defined paths that should be protected and compares them to the current file's path. The result of the search is then stored in the context alongside the version of the list mentioned earlier. If the driver is dealing with a file which should be protected, it flushes the cache defined for the current file and returns `STATUS_REPARSE` and restarts the operation to release any locks.

Processing any consequent IRPs allows the driver to quickly recognize, whether it should be protecting the file or not. It does so by reading the information stored within the file context. If the driver is not interested in this file, it simply forwards all IRPs regarding this file to the underlying driver. On the other hand, if this is a file, which should be protected, the driver reads first few bytes to determine whether the file has already been encrypted or not. In case it reads the specified encryption header identifier, it assumes that the file has been protected by the driver. This information is stored in the context and the callback is completed.

In case the driver does not read the encryption header identifier at the start of the file, it performs a second read, to read the whole file. Then, it calls the encryption library and encrypts the content of the file and also appends the encryption header. After that, the driver performs a write operation and replaces the previous content of the file with its encrypted content.

After the file's handle is obtained, the system will usually query some information from the file by issuing `IRP_MJ_QUERY_INFORMATION`. This IRP has to be handled by the driver as well, although it modifies only one parameter.

`IRP_MJ_QUERY_INFORMATION`

Pre-operation callback only checks for the file context. If there is the context attached, the driver returns a status to indicate, that the post-operation callback should be invoked.

Post-operation callback gets invoked, on its way back up the driver stack, after it reached the file system and has been filled with the requested information. Since this callback got invoked, the driver knows, that it should modify the returned information. Generally, the driver attempts to modify any file size information, to match the size of the file in its original state. The exception to this rule are cases, when this callback gets invoked right before a read request is issued. The driver recognizes these cases and returns the real size of the file. This is done to avoid issues, where the file system appends null characters to the end of the file, since technically we have to read beyond the end of said file when we are processing a read request and some applications may not deal nicely with that. In these cases, the driver returns the original size to avoid this issue.

Data Read

When a user requests to read the content of a file, the system issues an `IRP_MJ_READ` towards the disk that stores the file. The pre-operation contains all the information necessary for the file system to properly read the data from the disk. The file's content is then returned in the post-operation in IRP's parameters.

`IRP_MJ_READ`

Pre-operation callback gets invoked before the request reaches the file system. The driver checks for the file's context and if there is none attached, it forwards the packet towards the file system. In cases when there is a context defined, the driver knows that it should protect this file. Instead of sending the IRP down the driver stack with modified parameters, it creates a new IRP and sets it with the new parameters. This ensures that the underlying drivers are working with the modified IRP. The driver modifies the offset from which the data will be read, it moves the offset beyond the encryption header, in order to read only the content of the file. The header and all the required information are read before in a separate request. Once the content has been obtained from the file, the driver fills in how many bytes have been read and completes the whole request, thus, sending the IRP back up the driver stack without invoking the post-operation callback.

Data Write

Whenever system needs to write some data to a file it must issue an `IRP_MJ_WRITE`. Similarly to the read operation, the system can issue a write operation without the user requesting one. For example, the lazy writer is a component, which is part of the CM and takes care of cache coherency. This means, that the lazy writer will write through any data, that has been modified inside the cache, but has not been updated on the disk yet. Generally speaking, these writes can occur at any point during the system run-time, but usually they happen right after the user modifies data inside the cache.

`IRP_MJ_WRITE`

Pre-operation callback gets invoked and is processed similarly to the read operation. The IRP is filtered by the driver before it reaches the file system. The driver then checks for an attached context and continues to process the request if it finds the correct one. In the case of the write operation, the driver does not move the file offset. Instead, it increases the parameter, which defines the size in bytes that are about to be written to the disk, to include the encryption header as well. It also creates its own IRP and sets the new parameters. Once the new IRP is completed, it transfers all information to the original one and adjusts the value, which defines the actual number of bytes written to the disk. It subtracts the length of the encryption header, thus, creating an illusion the application has written only the size of the content.

6.2 Encryption Module

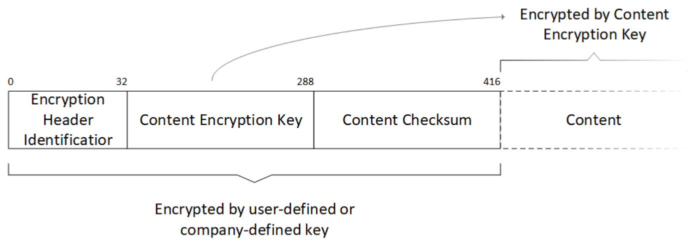
The encryption part of the thesis is implemented as a library. This allows anyone adopting the library to modify, for example, algorithms used or the structure of the encryption header to suit their needs. The encryption logic works by encrypting a content of a file and appending the encryption header to the beginning of the file. This header is later used to read all information needed to access the content of the protected file. By default, the library uses AES-256 for encrypting the content as well as the header itself and BCrypt's random generator[1] to generate random key to encrypt the file's content.

The library is implemented in C, so that it can be easily linked to the driver. The AES encryption uses algorithms implemented by Brian Gladman[2]. This implementation is supposedly secure enough to be recommended by the AES group at Ecole Normale Supérieure which evaluates multiple different implementations[3]. Furthermore, this specific implementation is written in C and even provides an assembler implementation. This allows us to

[1] https://docs.microsoft.com/en-us/windows/desktop/api/bcrypt/

[2] https://github.com/BrianGladman/aes

[3] https://www.di.ens.fr/david.pointcheval/Documents/Papers/w1999_AES2.pdf



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 6.1: Figure which shows the binary structure of the encryption header.

use the encryption algorithms in kernel quite easily and the library only requires a couple of modifications in order to work properly.

Encryption Header

As it was mentioned above, every file that is protected by the driver contains a header at the start of the file. This header contains all the necessary information for the driver to access the file's content. The header's structure is describe in figure 6.1. The design of the header is inspired by the patented header structure by ESET[4].

The content encryption key is generated with real randomness. It uses the BCrypt's API function `BCryptGenRandom()` to generate the key. The function uses true randomness which complies with NIST SP800-90[5]. This ensures, that the key is truly random and therefore minimizes the issue of key re-use. As the name suggests, this key is used to encrypt the entire content of the file.

To make sure, that an unauthorized person cannot access the content key stored inside the header, the entire header is stored encrypted as well. The header is encrypted using either a user-defined key or a company key provided externally to the driver. When the header is decrypted inside the driver, first, it verifies the checksum stored at the end of the header, to make sure that the content has not been tampered with.

6.3 Driver Configuration

The driver provides an interface, which can be used to communicate with it. The interface is implemented using the `FltPort`[6] interface. This means, that anyone who knows the identifier of the `FltPort` can implement an application, which would be able to configure the driver. The driver accepts the following commands:

- Set user key.
- Add company key.
- Remove company key.
- Add protected path.

[4]https://permalink.orbit.com/#/patent;xpn=3XF3B5E3UZPjCo0EB2EWlnfDUqlXTJ5uwQdFuycu4uk%3D%26n%3D1;id=0;base=FAMPAT

[5]https://en.wikipedia.org/wiki/NIST_SP_800-90A

[6]https://docs.microsoft.com/en-us/windows-hardware/drivers/ddi/content/fltkernel/nf-fltkernel-fltcreatecommunicationport

- Remove protected path.
- Stop the encryption.
- Start the encryption.

Furthermore, I decided to implement a simple command line application, which demonstrates the proper communication with the driver. The application is able to set a user's password, add a protected path, and remove a protected path. Once again, the application is implemented in C, in order to easily utilize the `fltUser.h` library, which provides implementation of `FltPort` interface for UM applications.

6.4 User-Mode Decryption Application

This application has been implemented to provide a complete encryption solution. It offers a very simple Graphical User Interface (GUI) which can be used to access any file without requiring the driver to run. Since this type of access obviously breaks the transparency, it is meant rather as a last resort and should not be provided as a common feature to the user. Instead, it should be under the supervision of an administrator and provided on demand.

The application uses the same encryption library as the driver. Therefore, it is able to decrypt the content in similar fashion that the driver is. It requires the user to specify which file they want to decrypt and also an output file. The application then invokes the library's functions, accesses the content and stores the output to the user-specified file. The output file can then be accessed without the driver's interference.

It is important to note, that this completely removes any protection from the file provided by the driver and it is then up to the the user to properly delete any files that have been accessed this way. The original file will, of course, stay protected the same way it was before.

Chapter 7

Testing and verification

The aim of this work was to provide a solution which would be able to extend an existing DLP system and provide new functionality to it. Therefore, the transparent encryption minifilter driver has been proposed and implemented.

To verify that it meets the requirements specified in subsection 3.1, a number of tests were introduced. These tests are aimed to verify that the solution does not impact the performance of the endpoint in a considerable way, as well as a proper functionality. There are two categories of tests. The first category is aimed specifically on testing of performance under heavy load and to what extent does the driver affect the endpoint. The description of these tests can be found in section 7.1. The second category of tests focuses on proper functionality and unlike the performance tests, it is manually verified. Section 7.2 describes these tests. All results are presented in section 7.3. The section also discusses and explains the results to provide a better insight into the performance impact of the solution. Finally, section 7.4 goes over the requirements of the integration process with the DLP solution and verifies that they have been met and properly addressed.

7.1 Performance Load Tests

I prepared a set of three tests, in order to determine the driver's impact on the performance of the endpoint. Since the endpoint's performance may be affected by different processes during the test time, each of the tests have been run 5 times to ensure that the measured results are not skewed by a random process suddenly requiring processor's time. To properly compare the driver's impact on the endpoint, the tests were run with the driver enabled and then the test was repeated with the driver disabled. Since the tests were the same, the results can be compared to assess the driver's impact.

The tests are implemented as a script file, which runs the selected application and start the application with an argument to immediately open a file. The script then waits for the application to finish opening the file and loading its content and then closes the application. The time it takes to do all these operations is measured and then reported. Despite the fact, that the tests results are influenced by the application start and exit time, the difference in times when running these tests with and without driver is still important and relevant, because we are not interested in the absolute time, but rather the time difference.

All the tests were performed in a Windows 10 virtual machine provided by Hyper-V, since debugging and analysing a driver is very limited on a local machine. The file system present was FAT32. This also allowed me, to run all tests on a clean system and to guarantee an identical environment for all tests, even after the machine was restarted.

Big Files Loaded into Memory

The first test is heavily aimed at performance and stress testing under a load. To minimize the impact of start and exit times of the used applications, tests were performed with `wordpad.exe` application, which is a simple text editor.

The main goal of this test is to verify how is the performance impacted when there is a large chunk of data being accessed and loaded into memory. A simple text file is enough to the time it takes to load.

The process of the test is as follows:

1. Open a 50 Megabyte (MB) file in `wordpad.exe`.
2. Wait for the application until it is done with processing and loading the file.
3. Close the application and move to another file and repeat from 1..
4. When 5 files have been tested, repeat from 1. with 100 MB file.
5. When the total of 10 files has been tested, end the process.

When the test was finished, the computer was restarted and the test was run again, this time using the other application. Thus, providing 2 sets of results.

High Volume of Small Files

The second test is similar to the first one. Once again, the main focus here is the performance load. Also, the same applications and file formats were used.

Rather than measuring the load time of large data, this test was used to measure the time it takes an application to access the file. This test is an important threshold, because most overhead happens, when the file is first accessed, as this is the point, where the file becomes protected.

The process of the test is as follows:

1. Open a 25 Megabyte (MB) file filled with a single character in `wordpad.exe`.
2. Wait for the application until it is done with processing and loading the file.
3. Close the application and move to another file and repeat from 1..
4. When 5 files have been tested, repeat from 1. with 25 MB file filled with randomly generated content.
5. When the total of 10 files has been tested, end the process.

7.2 Proper Functionality Tests

As mentioned above, these tests are aimed to verify real world scenarios. Usually, users will not use text files, but rather more complicated formats. This test was introduced to cover such cases and to better verify, how the driver will function in reality. These tests were performed and verified manually by me. They were performed only once, since their aim is to solely verify the functionality and that should not change with multiple runs.

These tests are also considerably easier to perform than the performance measurement. They only provide a simple answer of yes or no. For this reason, it was possible to perform and verify them manually, otherwise an automated process would have to be introduced.

Similar to the automatic tests, these tests were run on a clean Hyper-V virtual machine with Windows 10 installed and NTFS file system.

The test was performed with multiple different file formats: Adobe Portable File (PDF), PowerPoint 2007 XML presentation (PPTX), Joint Photographic Experts Group (JPEG), Moving Picture Experts Group (MPEG), and XML Word document (DOCX).

The applications used were FoxIt Reader, LibreOffice Impress, Photos, VLC, and LibreOffice Writer. There was no particular reason, other than that these applications are easy to install on a virtual machine without them requiring a license. That being said, they are well known alternatives and they are able to open the tested formats as expected.

The entire testing process form is available in appendix E.1. The simplified process of the test is as follows:

1. Open a sample file in one of the designated applications.
2. Wait for the application until it is done with processing and loading the file.
3. Verify, that the content matches the original file.
4. If the content matches, close the application and finish the test as positive. Otherwise, finish the test as negative.

7.3 Results

This section summarizes the results from both categories of the tests. The first category produced time measures, which are presented in side-by-side tables, to better present the run-time differences of tests that have been run with the driver and without the driver. The results from the manual test are presented as a percentage successful rate.

The run-time of scripts which performed the automatic tests was measured with a powershell utility command `Measure-Command`[1]. This command is able to measure the time it takes to run a different command, a script, or a batch file.

The Automatic Performance Test Using Big Files

The test is described in the subsection above. The test was run with five 100 MB files and five 50 MB files. The measured time is equal to reading 5 files of 100 MB and 5 files of 50 MB. The results of the test when run without the driver are presented in table 7.1 and the results of the test run with the driver are in table 7.2.

As we can see from the results, the read time is slightly slower when the driver is present. The time difference is most likely caused by the driver decrypting the content. Although, the difference is almost negligible, the read with the driver present is still slower, but as we will see in the other test results, it may not always be the case.

[1]https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.utility/measure-command?view=powershell-6

Another thing that is worth pointing out, is that the first run takes the same time as any of the other runs, despite the fact, that the driver should be encrypting the data on its first access. This happens, because the system already opened the file's handle, before the test was run and, therefore, the driver already encrypted the file. This case is hardly avoidable, because it is not guaranteed when the system may try to access the file. That is why I decided to not attempt to measure the time with the first time access, but rather focused on the consequent accesses. This stands true even for the other automatic test.

Run	Time	Ticks
1	1m12s153ms	721532435
2	1m12s148ms	721486909
3	1m11s146ms	711464734
4	1m13s150ms	731505056
5	1m12s146ms	721461106

Table 7.1: Results of second test run without the driver.

Run	Time	Ticks
1	1m15s172ms	751725967
2	1m14s157ms	741571194
3	1m15s154ms	751548330
4	1m16s155ms	761550841
5	1m15s155ms	751558476

Table 7.2: Results of second test run with the driver.

The Automatic Performance Test Using High Volume of Files

The test is described in the subsection above. The test was run with five 25 MB files filled with one character and five 25 MB files filled with random content. The measured time is equal to reading ten 25 MB files. The results of the test when run without the driver are presented in table 7.3 and the results of the test run with the driver are in table 7.4.

Here we can immediately see the difference between the previous test and this one. In this case, the driver's read time is significantly faster. This is caused by the driver reading the entire file into memory on its first access. The entire file gets cached, and the cache does not have to be updated later, when the file is actually being read. When the file is accessed for the first time without the driver running, the cache is filled by data chunks with each of the read request. Reading the entire file into the memory is more advantageous in the case of smaller files, because the file system and CM did not have enough time to optimize and predict the read requests.

Run	Time	Ticks
1	49s104ms	491043834
2	49s102ms	491024898
3	49s102ms	491022530
4	49s101ms	491018874
5	49s102ms	491025698

Table 7.3: Results of first test run without the driver.

Run	Time	Ticks
1	32s77ms	320773019
2	31s81ms	310811816
3	32s68ms	320683707
4	32s67ms	320674719
5	31s66ms	310665763

Table 7.4: Results of first test run with the driver.

The Manual Functionality Test

The functionality verification is described in subsection 7.2. The test was run with a total of 40 files with varying file formats and sizes and always with the driver present. The result of each of the tested file was either yes or no, depending on whether the file was opened by the application and the content of the file was identical to the original, before the encryption. The results of the first time the test was performed is presented in table 7.5.

The first results presented a serious issue. That is, some of the files were not possible to open in their respected applications. After examining the driver behavior and the application's expected response, I managed to pinpoint and fix the issue. The error was caused by the LibreOffice Impress application, because it requested the content of a file differently in different chunks than the other applications. A new check was introduced to the code, to verify that we are properly responding to both types of the requests. Table 7.6 presents test results, after the fix was applied.

A different issue was discovered with MPEG files. While it was possible to open the file, the content of one of the movies was slightly changed. The frames were still very similar, but some of pixels had a different tone of color. I attempted to fix this issue multiple times, unfortunately, all the parameters of requests processed by the driver looked correct. To properly pinpoint the issue, I would need to analyse every chunk of data sent from the driver and compare the data in binary form. Such analysis would be very complicated and time consuming. Therefore, I ultimately decided to give up on this issue and accept it as an odd case, since all the other movies played properly.

Format	Files	Succeeded	Failed	Success Rate
PDF	10	10	0	100%
PPTX	5	0	5	0%
JPEG	10	10	0	100%
MPEG	5	4	1	80%
DOCX	10	10	0	100%

Table 7.5: Results of the manual test of functionality before a fixed issue.

Format	Files	Succeeded	Failed	Success Rate
PDF	10	10	0	100%
PPTX	5	5	0	100%
JPEG	10	10	0	100%
MPEG	5	4	1	80%
DOCX	10	10	0	100%

Table 7.6: Results of the manual test of functionality after a fixed issue.

7.4 DLP Solution Integration Requirements

There were also requirements for the DLP solution integration process mentioned earlier – hidden from the user, easy integration and configuration, and secure. This section should show, that all the requirements were met in the final solution and that it can be integrated with the DLP solution.

By design, the protection works transparently. Whenever a user accesses a file, the driver ensures, that he/she is working with the file in its decrypted form. This is done for every file that is protected. Similarly, the file size of a file is modified as well. Therefore, once the solution is deployed and properly configured, the user will be able to continue his/hers work as before, but with the added benefit of data protection for certain files. The fact, that the protection works this way is shown, for example, in subsection 7.3, where the file accessed is encrypted, but the displayed data do not look encrypted.

The solution is able to integrate with any other system, because it provides interfaces for configuration. These interfaces can be easily accessed and utilized to configure the driver, as demonstrated in the application I provided in the final solution, to showcase the communication. The interface as well as the application is described in section 6.3.

Finally, the security of algorithms, which were chosen by default, is discussed in section 6.2. Based on that, the encryption can be deemed secure enough to be utilized in customers' environments. Furthermore, the library which provides the encryption can be easily swapped to modify, which algorithms are used to provide the encryption. This makes the solution extensible and easily maintainable in the future.

Chapter 8

Conclusion

The goal of this work was to extend an existing DLP product by implementing an encryption solution which would be able to integrate with it. The final solution is able to transparently protect data on endpoint and external devices. It is also configurable through a provided interface, which a DLP product can implement.

The design approach was chosen after the analysis of multiple DLP products and their approaches to encryption modules. Based on that decision, a significant part of this work was an extensive research of the file system internals and Windows Minifilter Framework. An understanding of these subjects was crucial to proposing and implementing the solution. The design has been adjusted multiple times, based on the experience gained during the research and even the implementation.

The final solution consists of multiple parts. The first part is a transparent module, which is the core of the work and implemented as a Minifilter driver. The second part is the encryption module, which protects the requested data. It also implements the encryption header and provides access to any of the encryption libraries. Furthermore, the encryption module is implemented as a library. Therefore, it is possible to simply swap the header structure and/or the used algorithm for a different one. This considerably improves the extensibility of the solution. The default algorithms used should be secure enough, as they are widely used by organizations which deal with security on daily basis. On top of that, the encryption header is inspired by a header structure that is patented by ESET.

The solution was then implemented based on the design and its functionality was demonstrated by automatic and manual testing on both NTFS and FAT32 file systems. The automatic tests showed, that the solution does not impact the endpoint in a significant way. On the other hand, the manual testing proved, that the solution is functional even in real life scenarios.

To demonstrate that the requirements of the DLP product integration process were met, an external application was implemented, which demonstrates the integration of provided interfaces. The application was then able to configure and control the driver, by sending the driver appropriate command messages.

The solution also works around a limitation that a driver implementation has – a requirement of administrator privileges when installing a driver. An external GUI application is provided, which is able to decrypt any file encrypted by the driver and store the content somewhere on the disk. Unfortunately, this approach breaks the transparency and, therefore, should be used with caution.

Finally, there is a considerable possibility for an extension of this work. The biggest weakness of the implemented approach is that it has to depend on some of the file system implementation details. While a drastic change to a file system is unlikely, there is still the potential risk, that the solution may stop working. Fortunately, it is possible to remove this dependency. That is, by implementing a complete file system basically from scratch. Although, a time and experience required to implement a project of such scale is beyond the scope of Master Thesis, I am definitely going to pursue this as the next step. In fact, I have already implemented a proof of concept driver to verify the validity and usability of this extension project.

Bibliography

[1] AGENCY, V. I. T.: *Information Technology Resource Management.* COMMONWEALTH OF VIRGINIA. 2016. [Online]. Retrieved from: https://web.archive.org/web/20170207183053/http://www.vita.virginia.gov/uploadedFiles/VITA_Main_Public/Library/PSGs/Information_Security_Standard_SEC501.pdf

[2] Graff, E.; Bazan, N.: *Choosing a driver model.* [Online]. Retrieved from: https://docs.microsoft.com/en-us/windows-hardware/drivers/gettingstarted/choosing-a-driver-model

[3] Graff, E.; Bazan, N.: *Device nodes and device stacks.* [Online]. Retrieved from: https://docs.microsoft.com/en-us/windows-hardware/drivers/gettingstarted/device-nodes-and-device-stacks

[4] Graff, E.; Bazan, N.: *Driver stacks.* [Online]. Retrieved from: https://docs.microsoft.com/en-us/windows-hardware/drivers/gettingstarted/driver-stacks

[5] Graff, E.; Bazan, N.: *I/O request packets.* [Online]. Retrieved from: https://docs.microsoft.com/en-us/windows-hardware/drivers/gettingstarted/i-o-request-packets

[6] Graff, E.; Bazan, N.: *User mode and kernel mode.* [Online]. Retrieved from: https://docs.microsoft.com/en-us/windows-hardware/drivers/gettingstarted/user-mode-and-kernel-mode

[7] Graff, E.; Bazan, N.: *What is a driver?* [Online]. Retrieved from: https://docs.microsoft.com/en-us/windows-hardware/drivers/gettingstarted/what-is-a-driver-

[8] Hollasch, L. W.; Kim, A.; Stroshane, M.: *Filter Manager and Minifilter Driver Architecture.* [Online]. Retrieved from: https://docs.microsoft.com/en-us/windows-hardware/drivers/ifs/filter-manager-and-minifilter-driver-architecture

[9] Hudek, T.: *Device Classes.* [Online]. Retrieved from: https://docs.microsoft.com/en-us/windows-hardware/drivers/install/device-classes

[10] Hudek, T.; Sherer, T. D.: *Bus Drivers.* [Online]. Retrieved from: https://docs.microsoft.com/en-us/windows-hardware/drivers/kernel/bus-drivers

[11] Hudek, T.; Sherer, T. D.: *Filter Drivers*. [Online].
Retrieved from: https://docs.microsoft.com/en-us/windows-hardware/drivers/kernel/filter-drivers

[12] Hudek, T.; Sherer, T. D.: *Function Drivers*. [Online].
Retrieved from: https://docs.microsoft.com/en-us/windows-hardware/drivers/kernel/function-drivers

[13] ISACA: *Data Loss Prevention*. [Online].
Retrieved from:
http://www.nortoninternetsecurity.cc/2011/03/data-loss-prevention.html

[14] Kath, R.: *Managing Memory-Mapped Files*. [Online].
Retrieved from:
https://msdn.microsoft.com/en-us/library/ms810613.aspx?f=255

[15] Nagar, R.: *Windows NT File System Internals: A Developer's Guide*. O'Reilly Media; 1 edition (September 11, 1997). ISBN 15-659-2249-2.

[16] Russinovich, M. E.; Solomon, D. A.; Ionescu, A.: *Windows Internals, Part 2 (6th Edition) (Developer Reference)*. Microsoft Press; 6 edition (September 25, 2012). ISBN 07-356-6587-7.

[17] Secretari, C.: *CNSS Policy No. 15, Fact Sheet No. 1* . [Online].
Retrieved from: https://web.archive.org/web/20101106122007/http://csrc.nist.gov/groups/ST/toolkit/documents/aes/CNSS15FS.pdf

[18] Shabtai, A.; Elovici, Y.; Rokach, L.: *A Survey of Data Leakage Detection and Prevention Solutions (SpringerBriefs in Computer Science)*. Springer. 2012. ISBN 14-614-2052-0.

[19] Zhang, E.: *What is Data Loss Prevention (DLP)? A Definition of Data Loss Prevention*. [Online].
Retrieved from: https://digitalguardian.com/blog/what-data-loss-prevention-dlp-definition-data-loss-prevention

Appendices

Appendix A

Contents of the CD

```
/
├── thesis.pdf ........................................Text of the thesis.
├── tex/ .............................................LaTeX source files.
├── source/ ..........Visual Studio 2017 solution containing all projects.
├── Readme.txt ...Text file with instruction on how to install the driver.
└── License.txt ..........................Text file containing the license.
```

Appendix B

Use-case Diagram

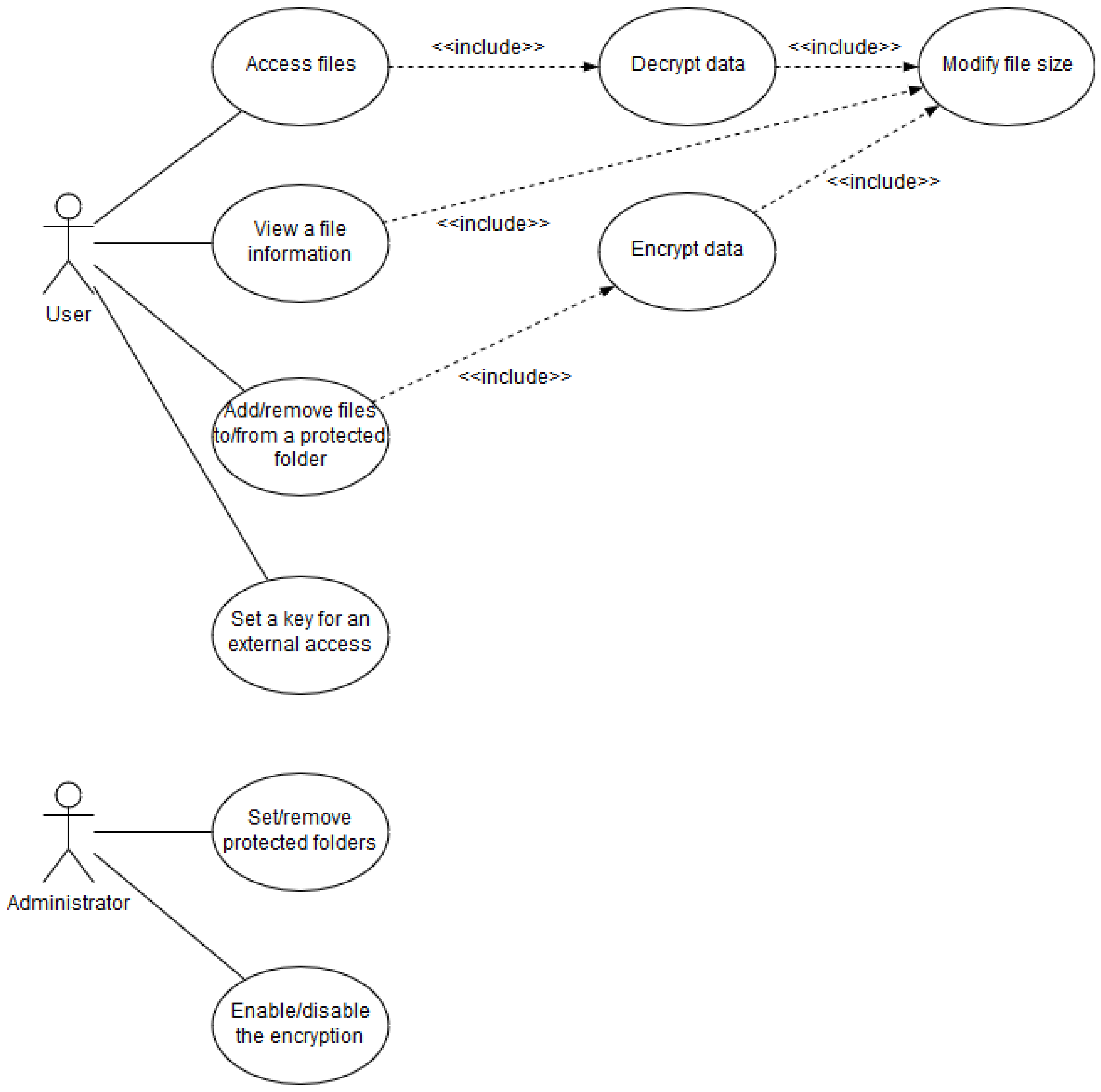


Figure B.1: Use case diagram describing the proposed solution.

Appendix C

Driver Installation

In the Windows system, each driver is represented as a service. Therefore, in order to install a driver a service with the corresponding name must be created. There are certain requirements regarding the attributes of the service and there must be specific keys defined in the service's registry entry for the service to be regarded as a driver.

By default, there are two options of installing a driver. The first option is a .inf file, which is usually included with the driver's binary file and it can be installed via context menu. This file creates all the necessary configuration and values in the registry. The second option is by manually creating the service and appending all the information in the system's registry. Both of these approaches are equivalent and is up to the user to decide, which is more convenient for him/her.

Appendix D

Driver's Registry Structure

HKLM\System\CurrentControlSet\
Services\MinifilterName
Type [REG_DWORD]: 1
Start [REG_DWORD]: 3
ImagePath [REG_EXPAND_SZ]: PathToTheDriversImage
ErrorControl [REG_DWORD]: 1
Description [REG_SZ]: DescriptionString
DisplayName [REG_SZ]: DisplayNameString
Services\MinifilterName\Instances
DefaultInstance [REG_SZ]: NameOfTheDefaultInstance
Services\MinifilterName\Instances\NameOfTheDefaultInstance
Flags [REG_DWORD]: 0
Altitude [REG_SZ]: 265000

Table D.1: The driver's registry structure.

Appendix E

Manual Test Protocol

Step	Expected result
Launch the external user-mode application	The console application is shown with the options to set a protected path, set user password, and stop encryption
Choose to set user password	The application prompts to enter a password
Enter a password	The application responds positively
Choose to set the protected path	The applications prompts to enter the path
Enter the location which contains the sample files	The application responds positively
Navigate to the sample file location and double click the file	An application that is assigned to the format by default starts to open
Go through the content displayed in the assigned application	The content is same as was in the original file
Close the application	The application closes properly

Table E.1: The protocol describing the manual test.