Windows 8.1 kernel driver security: Exploitation and security enhancements

Thomas BOUÉ, Théo BOULAIRE, Thibaut FRIN
Mouslim BERSANOUKAEV, Lucas GIORDANI
Supervisor - Alexandre GONZALVEZ

ABSTRACT

Ransomware attacks may be developped by abusing vulnerabilities of legal device drivers. From a minimum set of knowledge, this paper hightlights the ease to reproduce some parts of those attacks. In order to prevent them, a simple Proof-of-Concept is developped over a legacy operating system. Kernel security mechanisms improvements are proposed from consideration of this implementation.

1 INTRODUCTION

The RobbinHood attack damaged entire information systems of the city of Baltimore [9] in 2019. This attack uses critical vulnerabilities in a specific driver in order to get an access on the victim's machines and extort data. The gang injects from the victim's network a payload to exploit a signed graphic card driver. They obtain a privileged access to the kernel. Then, they activate their malicious binary. RobbinHood attacks are a type of Malware that belongs to the Ransomware family.

Ransomware are a particularly harmful form of Malware which limits an individual's access to their data and asks payment to restore functionality [49]. Ransomware attacks are usually decomposed into four phases: *Infection, Communication with Command and Control, Destruction, and Extortion* [8]. The success of those attacks strongly depends on the bypassing of protections available on the targeted information systems, such as the privilege rings.

The architecture of *Intel-x86* processors implements four rings, or hierarchical layers, of protection to deal with access privileges for the memory and resources [37]. Ring 0 is the most privileged whereas Ring 3 is the less privileged. The operating system's kernel operates in Ring 0, meaning that it can have access to any part of the memory and any resource of the machine. Usually, operating systems like Linux or Windows only use Ring 0 and Ring 3 [48]. As a consequence, device drivers operate in the same ring as the kernel. They must be developed carefully to prevent malicious behaviours.

A device driver is a piece of software attached to a computer telling the operating system how to communicate with a corresponding piece of hardware [11]. Every OS [1] and particularly the kernel have their own way of dealing with drivers. In *Windows OS*, all drivers have a Ring 0 memory access and share the memory space of the kernel.

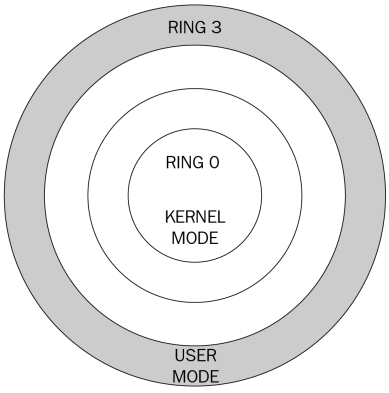

Figure 1: ***Privilege rings for the x86 instruction set [37]***

This work is based on the replication of certain phases of the RobbinHood attack. Using a PoC[2] [52], we will try to understand which kernel security mechanisms of Windows OS 8.1 may be improved to struggle against driver-based exploitation. We will first introduce the context of the attack, then we will talk about its implementation. Finally, we will discuss about kernel security features that could be enhanced or added to prevent this kind of attack from damaging Windows OS 8.1 environments.

2 WORKING CONTEXT

The aim of this work was to realize a partial reproduction of the RobbinHood attack. The replication of the attack will focus on the *Communication with Command and Control* and the *Destruction* phases of Ransomware attacks. The aim is to exploit a vulnerable driver to gain kernel access and then encrypt part of the user's data. The *Infection* phase will not be treated here, assuming that the exploit is already present in the targeted information system. Also, the attack will not implement the *Extorsion* phase for obvious reasons, as this attack is used as a demonstration. The targeted OS is *Windows OS 8.1 version 6.3.9600*, running on a virtual machine. This working environment was chosen mostly for local kernel debugging [33].

3 ADVERSARIAL MODEL AND OFFENSIVE ARCHITECTURE

Let's define an adversarial model for a driver-based attack on Windows 8.1 with a PASTA model for the defensive part and a MATE model for the offensive part.

[1] Operating System

[2] Proof of Concept

3.1 PASTA modeling

The Process for Attack Simulation and Threat Analysis (PASTA) is a risk-centric threat-modeling framework developed in 2012 [55]. In the two following paragraphs, a defensive model will be set thanks to the different steps of PASTA.

In terms of its business objectives, the Operating system provider aims to cultivate a satisfied customer base and deliver a smooth user experience with their products. One of the main threats in the context of Ransomware attacks on Windows OS lies on kernel token manipulation [7] because a `NT/System` token is needed to import a new driver and disable Windows Defender. Users should not be able to do these two things. That's why securing that point is a major concern for the company. To study this, the technical scope will focus on rights on kernel manipulation. The entry point of the threat is that anybody can attach to a driver and act as a kernel object with the driver's functionalities. In the case of vulnerable graphic card drivers, it leads to memory allocation, writing and token manipulation in kernel land.

Drivers recognized as vulnerable cannot just be considered like malicious tools. The kernel is required to permit them for the sake of ensuring compatibility. Two main ways have been identified to stay secure considering this point. The first one is to purely prohibit users to load well-known vulnerable drivers. The second one is to keep a trace of functions calls to watch which functions get access to the kernel. By keeping an eye on these two points, the kernel may be able to detect if unusual behaviours occurs. For the resources enumeration part, plenty of resources dealing with vulnerable driver exploitation on Windows OS are available on the web[2], but also lots of exploits on various drivers. As a consequence, it is easy for anyone to reproduce this attack even with few knowledge on Windows OS.

3.2 MATE modeling

The Man-at-the-End (MATE) Attack is a security model that was devised in the wake of software releases [10] in order to formalize reverse engineering-based attacks. For instance, drivers are often shared with users, who enjoy unrestricted access to them. However, a Windows driver is equipped with an exploit framework that curtails certain attack vectors. Specifically, code injection is rendered impossible by the MATE Attack. This is accomplished by leveraging a remote attestation protection mechanism that requires execution attestation by a higher authority. It is worth noting that Windows drivers are signed by Microsoft [24], and any alteration of the original binary will result in a loss of signature, thereby rendering the driver unexecutable.

In the event that binary modification is not feasible, analysis becomes a necessary alternative. Program obfuscation is a technique that aims to disguise the original code's intentions by a program transformation. It is commonplace for high-risk programs to undergo obfuscation, thereby impeding human comprehension of the binary [15]. As such, the latter's absence can lead to inadequate protection. Our inspection of the driver binary revealed a lack of this security feature, which further facilitated comprehension by making the binary more transparent, with debug traces readily available. The comprehensibility of the binary may be gauged by evaluating the depth of function calls and identifying "branch function" type protection that hindered the easy reconstruction of function calls. An example of the call graph of the dispatcher function is presented in Figure 2. The absence of protection at this level significantly eases the comprehension of the driver's workings.

To conclude, the driver targeted in this study represents an optimal choice for an attacker aiming to maximize their Return on Investment (ROI). Our comprehensive analysis has identified multiple vulnerabilities in the security architecture of the system, which have greatly facilitated the process of exploiting the target machine. Consequently, an attacker would be able to gain complete control over a Windows machine with relative ease and a high probability of success.

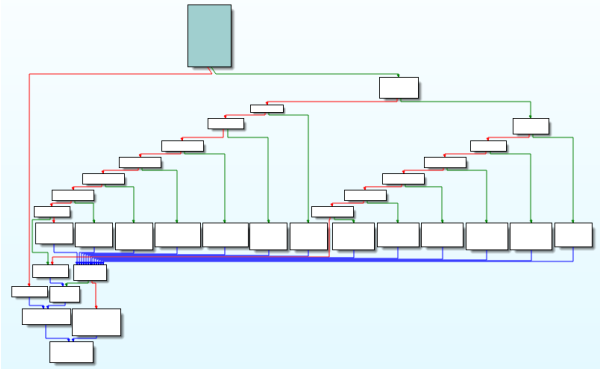

Figure 2: ***IDA graph view - Call Graph of the driver loader's Dispatcher Function***

3.3 Offensive architecture

Let's detail the security mechanisms implemented in Windows 8.1 and how to bypass them to craft the attack.

The most famous protection in Windows OS is the *Blue Screen of Death (BSOD)* [29]. This feature was implemented to stop the operating system in case of a critical error. To prevent more damage, the kernel enters in a special routine that will throw the famous blue screen and shutdown Windows. Various conditions can lead to a BSOD. We can mention hardware failures, issues with device drivers or unexpected events in the kernel memory. For instance, a BSOD is triggered when a device driver attempts to access a memory address that it was not supposed to. This behaviour is usually caused by outdated drivers.

Moreover, one of the major kernel security features available and strongly enhanced in Windows 8.1 is PatchGuard [5]. This feature, which consists in a highly obfuscated code residing in the kernel, was developed by Microsoft to prevent thirdparty applications or drivers from accessing and modifying the Windows kernel. PatchGuard periodically checks the integrity of the key codes and structures of the kernel but also elements of the CPU like special registers. If anything unusual is detected, such as copying a payload into the kernel memory without allocating a region for it, the system is shut down immediately thanks to the BSOD mechanism.

Like the kernel and drivers, PatchGuard operates in Ring 0. One of the use cases of PatchGuard lies on the denying of kernel memory modifications performed by unauthorized third-party drivers.

Furthermore, Windows OS uses the concept of System Services [28], which are services that start at the launching of the operating system and that cannot be disabled entirely without the appropriate access rights. This feature is used to prevent the disabling of the Windows Defender service. However, plenty of services related to Windows Defender can be disabled with `System` rights [18] such as script and email scanning, network analysis, or real time monitoring.

To be able to bypass these features, the goal is to find a function in the signed device driver that allocates a memory region in Ring 0 thanks to a kernel method. As the driver is signed, PatchGuard will not interfere with this function and let us insert a shellcode into the kernel memory and about higher privileges. A shell with `NT Authority/System` rights (the targeted rights in our attack) will possess the required privileges to disable security features such as monitoring or script scanning. To overcome the difficulty of killing the Windows Defender service, the easiest solution found lies on the use of an existing tool that is able to get the correct access rights and disable the majority of Windows 8.1's security services [43].

4 OFFENSIVE IMPLEMENTATION

The context in which the attack is performed is now settled. The following steps detail how to obtain the full control of a machine by exploiting a vulnerable signed driver with the support of a resource published to explain how kernel driver exploitation can be performed on Windows [47].

4.1 Reverse the vulnerable driver

The first step focuses on getting an access on the kernel. Knowing that drivers in Windows OS operate in Ring 0, the idea is to try to do reverse-engineering in order to understand how this driver works and maybe find a vulnerability in a particular function. The vulnerable driver used is *Gigabyte AORUS Graphics Engine v1.34* [53], installed and running on the machine. To be able to interact with the Gigabyte hardware, this version installs the *gdrv2.sys* file, or also named the *gdrv-loader* on the system. This file is the loader driver used by the kernel to ensure the communication between the operating system and the hardware. As a consequence, the reverse-engineering part is focusing on this particular file.

Ghidra is a free tool that is able to perform this part [35]. The main goal is to find a function that allows reading or writing in the kernel memory. The loader driver implements a dispatch function used by the entry point function. This dispatch function is simply a huge `switch` between different I/O control codes or IOCTLs[3] (cf. appendix 1) [23]. These values are used by the Windows API are more precisely the `DeviceIoControl` [26] function to send an IOCTL to a device driver and then perform the operation that corresponds to this code (*ie.* the block in the `switch` of the dispatch function that fits with this code).

[3]I/O Control Code

Figure 3: Ghidra - View of the dispatch function of the driver loader

```
if (uVar1 != 0) {
    lVar5 = lVar5 - (longlong)puVar4;
    uVar3 = (ulonglong)uVar1;
    do {
      *puVar4 = puVar4[lVar5];
      puVar4 = puVar4 + 1;
      uVar3 = uVar3 - 1;
    } while (uVar3 != 0);
}
```

Figure 4: *Conditional structure handling memory copying*

After inspecting all the methods related to the available IOCTLs for this device driver, there is a block in a function that seemed like an implementation of `memcpy` [27] as the code shows in Figure 4.

The `DeviceIoControl` function is called from the user land, meaning that this is the entry point to the kernel memory. By using the IOCTL related to that method, a lower-privileged user is allowed to write into the kernel memory, so in Ring 0.

4.2 Privilege Escalation

Having a `memcpy` like function is interesting but it is useless without allocating memory space for the content of the payload. Fortunately, there is also a function used by the loader driver that allocates a contiguous space in the kernel memory by using the function `MmAllocateContiguousMemory`, a kernel method [31]. The next step is to craft the payload. In this case, the solution is to craft a shellcode that performs token stealing on the `cmd` process to obtain higher privileges like `NT Authority/System`. This level of privilege is not the highest available on Windows OS but allows all the operations needed to build the PoC.

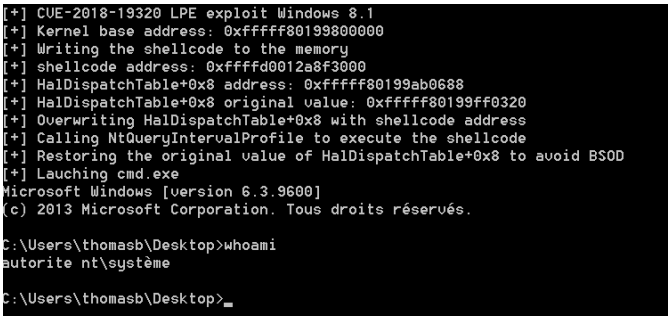

Figure 5: Privileged shell

Let's explain how token stealing will give us a shell with the right privileges. In most of modern operating systems, the kernel architecture relies on structures, and processes follow also this rule. In Windows OS, the structure used to represent a process is called `EPROCESS` [42]. This structure is composed of many fields used to describe the process and the field used for rights and policies is `Token`. The overall goal of this shellcode is to find the `EPROCESS` structure of `SYSTEM` and then use its `Token` instead of the original one of our lower-privileged `cmd` process that we control as a user. In order to find the various offsets and fields of the `EPROCESS` structure, the tool `WinDbg` enables to debug the kernel and set breakpoints to stop the system [30].

After crafting the shellcode, the final step consists in calling the functions discovered before and the Windows Driver API to write our payload into kernel memory.

4.3 System rights

The shellcode is in memory but is not executable at this time. For that, we will use a famous technique discovered in 2007, still relevant today and mentioned in the offensive implementation. This technique uses the kernel dispatch table `HalDispatchTable` to overwrite a function pointer of this table with the address of our shellcode [47]. Indeed, there is an undocumented function named `NtQueryIntervalProfile`, used rarely, that uses one of the function pointers of the `HalDispatchTable` in his implementation [36]. Overwriting this function pointer with the address pointing towards the beginning of our payload and calling `NtQueryIntervalProfile` enable us to execute our shellcode. Moreover, to avoid any BSOD triggered because of the modification of the table, it is crucial to save its original value in a temporary variable and reset it after the execution of our shellcode. The result of these manipulations is a `cmd` with `NT Authority/System` rights without any interaction with the GUI other than the double-click of the user on the exploit's executable (cf. Figure 5).

4.4 Bring Your Own Vulnerable Driver

The BYOVD method consists in bringing on the targeted machine a signed driver to execute malicious or higher-privileged operations on it [38]. The most famous driver used in this case is `mimidrv`, a driver provided by `Mimikatz`, a tool that gives to the attacker a set of primitives to exploit a Windows machine [16, 40]. The aim of this step of the attack is split in two parts. First, update the cryptographic libraries of Windows in anticipation of the final step about ciphering the user's data with the cryptographic libraries available on *Windows OS*. Second, use functionalities of `mimidrv` that disable all the protections related to Windows Defender and `lsass`[4], allowing us in a second time to use other payloads to disable them permanently [25].

Having already disabled various protections with our shell facilitates the downloading of Mimikatz, which is performed without any alert from Windows Defender.

4.5 Disabling Windows Defender

In the context of the RobbinHood attack, the strategy adopted was based on disabling the Windows Driver Signature Enforcement [24] in order to load a malicious driver used to disable the security services of Windows OS. The final step is to disable the antivirus protections of the OS and other strategies may be applied to reach this situation. By default, the targeted service to shut down is Windows Defender. As the Windows Defender service is not killable by a process running with `System` rights, a method would consist in creating an access token with the right SID attributes to disable Windows OS's security services [51]. This strategy is applied by the tool `btsp.exe`, available on *GitHub* [43]. At this moment, the security features stoppable with `System` rights were already disabled, so `btsp` can be downloaded stealthily and then executed. After execution, all antivirus features available on the OS are stopped.

4.6 Ciphering data

The exploit has now obtained the full control of the Windows machine thanks to a vulnerability in the *Gigabyte AORUS Graphics Engine v1.34*. To finish the replication of the RobbinHood attack, the final step deals with ciphering the user data.

The cryptographic API often used by Ransomware is the Microsoft Crypto API (MS CAPI) [39]. This legacy API, presents since Windows 95, implements an easy-to-use interface to cryptographic functions for user-land applications. Even if its usage is depreciated, replaced by Cryptography API Next Generation, it still used by *OpenSSL*. With RobinHood, the choice was made to encrypt data with the *AES-ECB* symmetric block cipher mode operation. For each file, it generates an *AES* key and encrypts the data. Although ECB operation mode has many flaws, it is still the fastest and can be easily parallelized because the encryption part must be as fast as possible in a Ransomware [6]. Then, the *RSA-4096* asymmetric cryptosystem is used to manage the secret key of the *AES* operation [39, 54]. This secure encryption algorithm has a security level of around 140 bits [13, 39]. The PoC implements the *AES-ECB* part of the ciphering operation with a 128-bits key. The range of encryption is limited to the all files on the *Desktop* folder of the user for the demonstration and the same key is used. The `CryptEncrypt` function, provided by MS CAPI, is utilized to encrypt data file [22]. Then, an embedded RSA public key at the end of the Ransomware is used to cipher the AES key. Finally, this one is exported in the *temp* directory. Several tests were conducted to measure the encryption time on three files of different sizes (1 KB, 1MB, 1GB). The table 1 reports the measured time in seconds.

[4]Local Security Authority Subsystem

Table 1: Measure encryption time on different file sizes

File size	1 kB	1 MB	1 GB
Encryption time (in second)	0.00542	0.02443	20.00868

5 IMPACT ANALYSIS

The PoC will be now analyzed in the prism of our adversarial models.

5.1 PASTA

Regarding the PASTA model defined before, all the threats mentioned can be found again in the PoC. First, attaching to the driver by getting a handle on it as a user. Then, allocating memory and writing in the kernel on behalf of the driver and that leads to the execution of a shellcode that is stealing a token to gain higher privileges. Besides, fundamental security mechanisms such as the Blue Screen Of Death are unable to stop the execution of the PoC whereas overwriting a function pointer in the `HalDispatchTable` should trigger this feature. The PoC is able to circumvent this issue through rapid restoration of the concerned pointers.

5.2 MATE

The main focus of our analysis was on binary analysis, as previously discussed. It was clearly unobstructed which led us to complete understanding of the binary. The debug messages left allowed us to find important functions. The depth of function calls was traceable by human analysts and provided a comprehensive understanding. The called functions did not have call stack checks. However, it is one of the recommended MATE protections. Drivers should normally control important functions such as `memcpy` [27]. This was one of the main security flaws of this driver.

6 PROTECTIONS AGAINST THIS BAD BEHAVIOUR

Windows 8.1 and its security mechanisms such as PatchGuard are not strongly efficient and don't monitor dangerous operations such as writing in the kernel memory with a device driver. Let's determine the various mechanisms that could be improved in Windows 8.1 to prevent driver based attacks from damaging the system. This part will broach security mechanisms related to the kernel but also to Windows Defender and Ransomwares.

6.1 Kernel-level protections

The entrypoint of the attack is the possibility to perform arbitrary allocating and writing operations into the kernel memory. To prevent that behaviour from happening, the following techniques could be settled and enhance the security of the operating system.

Kernel data structures integrity. To gain `Authority/System` rights, a shellcode was crafted to perform a Direct Kernel Object Manipulation, or DKROM, and modify critical data structures in the kernel [41]. As the driver used is signed, PatchGuard was bypassed, exposing critical data structures in the kernel. This dangerous behaviour could be forbidden with an new strategy dealing with privileges. Indeed, the `Token` field of a low-privileged shell could be altered and modified with the one coming from the `System` process. By designing a new layer of privilege in the existing hierarchy, Token manipulation [7] and other operations on critical data structures must be restricted to this new level of privilege and not `System`. In this case, the shellcode in our PoC responsible for performing Token stealing would be inefficient, as a process with `System` privileges would not be capable of modifying the `Token` field of the `EPROCESS` structure of the `cmd` process.

Kernel isolation. As said before, the kernel and device drivers work in the same space in Windows 8.1. It means that if a device driver is able to compromise the kernel by arbitrarily allocating memory or copying data into kernel memory, no other security feature can stop the process. This issue might be resolved by dividing the kernel into two separate spaces. The most sensible and secure part of the kernel would run into one virtual space whereas the other part of the kernel, where we could let device drivers operate, would run into the other separate virtual space. The goal of this technique is to let the secure space handle all memory operations performed by elements operating in the other space.

As an example, the PoC uses a function of the device driver that calls `MmAllocateContigousMemory` to allocate memory in the kernel. By using this technique, the function will be called from the secure part of the kernel, allowing the latter to handle that region and have full control over what actions are performed in that region. This technique could take advantage of the kernel data structures integrity technique that we mentioned before to handle malicious DKROM.

Hardware-based security and enclaves. To continue on the idea of separating the kernel from device drivers, trusted execution environments (TEE) may be interesting [46]. A trusted execution environment is designed as an isolated processing environment in which applications can be securely executed irrespective of the rest of the system. This method may be implemented using low-level hardware and software-based mechanisms. On the one hand, hardware-based features could provide a trusted environment for executing sensitive operations dealing with device drivers and verify the integrity of the kernel before preforming those operations. On the other hand, software-based mechanisms like enclaves may be used to isolate the device driver in an isolated environment and detect any unauthorized access of a kernel resource located outside the trusted environment [50].

As several kernel resources are manipulated in the PoC, this system might prevent the DKROM from being executed for instance.

6.2 Antivirus strategies

During the whole execution of the attack, and even before when the exploit is located on the target machine, Windows Defender does not trigger any security alerts. However, various upgrades could be set up to fight against the Malware.

Protection of Windows Defender settings. Even if Windows Defender seems to be a difficult service to kill, it is still possible to do it with several tools like `btsp` (cf. section 5.5). The experiments highlight that when all security services, including Windows

Defender, are disabled, the operating system is slowed and completely sensible to all kind of threats. A important security feature that could overcome this problem is to forbid any unauthorized modification of the security settings and modifications of Windows Defender. The kernel could keep a list of critical security features that cannot be altered from their default state, for instance by using the registry keys reserved for the kernel and that can only be modified at boot time. It will then be impossible to disable all the features related to Windows Defender whereas the exploit has full rights on this operation during the attack.

Real time protection. One of all protections available on Windows Defender is called the real time protection. This feature is designed to monitor system activity, detect and block malicious software in real-time. This feature is divided into several techniques such as signature based detection, heuristic analysis, behavior monitoring, sandboxing and cloud-based protection.

All of these features can be upgraded. We will focus on the heuristic analysis, which consists of analyzing specific blocks of a file and looking for suspicious code sequences. One way to improve this feature could be to use advanced machine learning detection in order to easily detect suspicious patterns in the code. This would effectively block the execution of our proof-of-concept, as the binary would be immediately detected by Windows Defender after its creation and moved to quarantine because of the DKROM performed in the shellcode and the suspicious system call graph.

6.3 Ransomware protections

The previous security mechanisms that may be improved are dealing with kernel mechanisms. Their range of action is limited to Ring 0. We also need to take account of what would happen in Ring 3 if the exploit succeeds by any means. In this case, the angle is based on encryption activities.

Here, a feature that checks if an application wants to apply changes to trusted files defined by the user seems to be an efficient way of detecting encryption operations. If an application wants to modify a list of trusted files and folders, the user is alerted with a notification and the process is blocked. It does not have any effect on what the exploit has done before, like disabling all the security features, but it may prevent huge data corruption.

Besides, a data recovery process after a Ransomware attack is a conceivable strategy. This process would operate in the Cloud in order to retrieve an old backup. This service seems to be suitable with `OneDrive`, a Windows service used to store data on the cloud [32].

7 IMPACT ANALYSIS OF PROTECTIONS

Improving the kernel security mechanisms of Windows OS 8.1 with the strategies discussed above would lead to various implementation changes and adjustments, both on hardware and software. Analyzing the impact of potential changes in the kernel with the protections previously suggested is necessary before making any changes on hardware and kernel architectures.

7.1 Addition of new privileges

Adding a new layer of privilege to avoid dangerous operations such as token manipulation may have positive and negative impacts.

On the one hand, a restriction on token manipulation with a new level of privilege may prevent privilege escalation and reinforce the security of the OS. It gives also more options to control user access and define new security policies. On the other hand, this new strategy may cause compatibility issues with older applications that do not work with the same layers of privileges. Moreover, this solution might increase the complexity of the system, as the kernel would not handle privileges as before. A new layer of security might complexify the inner-workings of Windows OS and change its internals [44, 45]. Finally, this strategy could induce a decrease of performances as additional processing would be required to handle this newly created privilege.

7.2 Kernel isolation

This technique seems to be promising. The kernel is now not as powerful as before because it is separated into two virtual regions, the most secured one isolated from the other. However, this choice of implementation would definitely have an impact of the performances of the operating system. Indeed, the kernel is now forced to run into two separated virtual environments, and the most secured part must monitor the activities triggered by the other part of the kernel [12].

7.3 Hardware-based security and enclaves

These low-level strategies may be used as an additional protection to strengthen the security of the system in case of an ill-written driver. The combination of enclaves and hardware mechanisms to ensure a trusted environment may reinforce the security of data manipulation. For instance, *Intel* CPUs implement mechanisms like *Software Guard Extensions* and *Trusted Execution Technology* to implement a trusted execution environment [17, 50].

However, this approach has several downsides. To start with *Intel*'s technologies, *SGX* is not reliable since the discovery of critical CPU vulnerabilities such as *Spectre* and *Meltdown* [20, 21]. Furthermore, without taking account of theses vulnerabilities, this strategy implies a compatible hardware. As an example, *Intel*'s features are unusable without a motherboard supplied with a compatible TPM [5] [34]. Thus, it implies to review the architecture of the operating system with various hardware setups.

Moreover, in order to have a completely secured enclave, the OS must be certain that the hardware component works as expected, not only at boot time. The *Spectre* and *Meltdown* attacks also showed that a legitimate program might abuse the trusting level of a hardware component. All these disadvantages render the practical implementation of this strategy difficult. Finally, the trusted execution environment is created at boot time, meaning that for this system to be efficient a reboot would be necessary after the driver's installation.

[5] Trusted Platform Module

8 OBFUSCATION

As mentioned before, modifying the binary to evade detection by Windows Defender may be achieved with obfuscation [15]. A lot of obfuscation methods are used by malicious software developers. For our PoC we chose the compile time obfuscation. This is a technique used to protect the source code of a program by making it more difficult to understand or reverse engineer. Several techniques [19] were applied to the PoC such as instructions substitution, bogus control flow and control flow flattening .

The tool used to perform obfuscation is the `Obfuscator-LLVM` project [19]. It can perform several obfuscation transformations. The tool used for the inner working is the LLVM intermediate language. It is an intermediate code generated between the system language (C++) and the assembly. Before performing the transformation our PoC binary was instantly triggered by Windows Defender on more recent versions of Windows OS. After the transformation Windows Defender was unable to detect malicious code in our executable. To compile our PoC we used the modified binary of clang with the the following LLVM-obfuscator options :

```
-mllvm -bcf -mllvm -bcf_prob=100 -mllvm -bcf_loop=3
↪ -mllvm -sub -mllvm -sub_loop=2 -mllvm -fla
↪ -mllvm -split_num=10 -mllvm
↪ -aesSeed=DEADBEEFDEADBEEFDEADBEEFDEADBEEF
```

Figure 6: *Clang Obfuscator-LLVM compilation directives*

The first option *bcf*, the bogus control flow option add opaques predicates before random instruction blocks. Opaque predicates are a portion of random code that are evaluated at runtime to a predetermined logical value. Then the *sub* option provides instruction substitution. It replaces simple arithmetic operations with more complex but equivalent operations. Finally the *fla* option provides flattening of the control flow. It disrupts the sequence of instructions block by placing them on the same level in a looped switch statement. The purpose of the *split* option is to apply the transformations multiple times on the same block.

As a result, the obfuscation techniques applied deeply altered the control-flow graph of the PoC . The size of the original PoC binary has grown from 4 kilobytes to 4 Megabytes with the obfuscation. The two binaries have been analyzed on *Virustotal* [4]. The original binary is flagged as a malicious file by 15 of the 70 security vendors. The obfuscated one is flagged as a malicious file by 12 of the 70 security vendor. There is a difference of 3 points. The gap is little between the two files because the obfuscation techniques used are well-known of the security vendors. Some upgraded version of the non commercial version of the `Obfuscator-LLVM` [19] provides much more transformations. As an example the `Hikari-LLVM15` [3] obfuscator contains a bunch of new interesting features like constant encryption and anti-hooking features. The result of the obfuscation is visible in the control flow graph of the PoC (cf. figure 8), which is the control-flow graph [14] after applying transformation and the original control flow graph of the exploit (cf. figure 7).

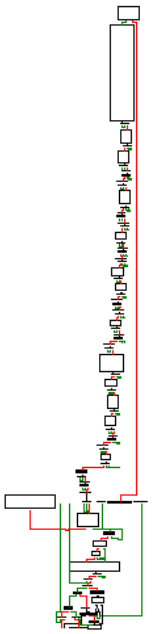

Figure 7: IDA graph view - PoC control flow graph

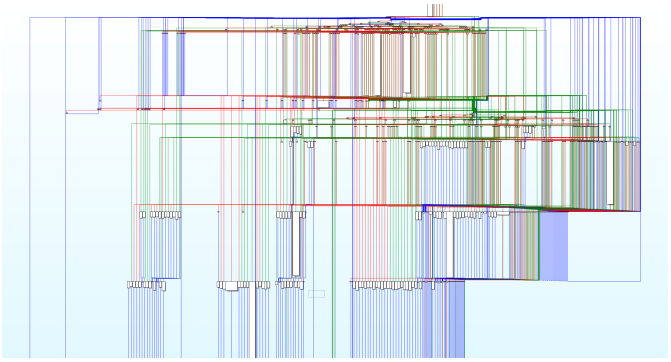

Figure 8: IDA graph view - Obfuscated PoC control flow graph

8.1 Discussions

The result of all these transformations is an executable that is extremely tedious to analyze and has become undetectable by several consumer anti-viruses including Windows Defender. There is no noticeable difference in the execution time between the obfuscated version of the PoC and the non-obfuscated one, it may not be the case for bigger programs. The problem remains that malicious code is easily identified by security vendors who incorporate deobfuscation mechanisms into their solution. Obfuscation is not the only way to bypass anti-virus like Windows Defender, a lot of techniques are nowadays used. For example we could use a PE32 packer [1]. The tool used for the PoC to obfuscate the code [19] is free and accessible to everyone. Other versions such as `Hikari-LLVM15` [3] offering more features are even often updated to provide more functionalities. So we can say that it is within the reach of everyone who wants to create Malware to obfuscate its code with a disconcerting ease.

9 CONCLUSION

This paper introduced kernel driver exploitation and how to deal with it in the Ransomware context. Vulnerable and signed driver-based exploitation is enough accessible to craft an efficient payload in a short amount of time in the context of Windows OS 8.1. To reduce the attack surface, various enhancements could be implemented in the operating system in terms of kernel security mechanisms, antivirus strategies and Ransomware protections. Other leads are conceivable, such as other virtualization-based security features or randomizing the offsets of kernel structures and implementing a new function in the Windows API that is the only piece of code capable of retrieving these offsets.

REFERENCES

[1] 2011. Detection of packed executables using support vector machines. https://www.researchgate.net/publication/224258044_Detection_of_packed_executables_using_support_vector_machines. (2011).

[2] 2023. CVE-2020-15368. https://github.com/stong/CVE-2020-153685. (2023).

[3] 2023. Hikari-LLVM15. https://github.com/61bcdefg/Hikari-LLVM15. (2023).

[4] 2023. VirusTotal. https://www.virustotal.com. (2023).

[5] Arush Agarampur. 2021. Windows Kernel Patch Protection - Achilles Heel: PatchGuard. (2021). https://www.youtube.com/watch?v=wXRLnp2JoWU&ab_channel=RSAConference

[6] Feillyan Antariksa. 2019. Ransomware Attack using AES Encryption on ECB, CBC and CFB Mode. *Jurnal Ilmu Komputer* 12, 1 (2019), 8. https://doi.org/10.24843/JIK.2019.v12.i01.p06

[7] MITRE ATTACK. 2017. Access Token Manipulation. (2017). https://attack.mitre.org/techniques/T1134/

[8] Craig Beaman, Ashley Barkworth, Toluwalope David Akande, Saqib Hakak, and Muhammad Khurram Khan. 2021. Ransomware: Recent advances, analysis, challenges and future research directions. *Comput. Secur.* 111 (2021), 102490. https://doi.org/10.1016/j.cose.2021.102490

[9] Catalin Cimpanu. 2020. Ransomware installs Gigabyte driver to kill antivirus products. (2020). https://www.zdnet.com/article/ransomware-installs-gigabyte-driver-to-kill-antivirus-products/

[10] Cataldo Basile Bart Coppens Bjorn De Sutter Daniele Canavese, Leonardo Regano. 2022. Man-at-the-End Software Protection as a Risk Analysis Process. (2022). arXiv:arXiv:2011.07269v3

[11] Tim Fishe. 2023. What Is A Device Driver ? (2023). https://www.lifewire.com/what-is-a-device-driver-2625796

[12] Francesco Gadaleta, Raoul Strackx, Nick Nikiforakis, Frank Piessens, and Wouter Joosen. 2014. On the effectiveness of virtualization-based security. (05 2014).

[13] Steven Galbraith. Security levels in cryptography. https://www.math.auckland.ac.nz/~sgal018/ACISP.pdf

[14] Robert Gold. 2010. Control flow graphs and code coverage. *Int. J. Appl. Math. Comput. Sci.* 20, 4 (2010), 739–749. https://doi.org/10.2478/v10006-010-0056-9

[15] Serge Guelton, Adrien Guinet, Pierrick Brunet, Juan Manuel Martinez, Fabien Dagnat, and Nicolas Szlifierski. 2018. [Research Paper] Combining Obfuscation and Optimizations in the Real World. In *2018 IEEE 18th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. 24–33. https://doi.org/10.1109/SCAM.2018.00010

[16] Matt Hand. 2020. Mimidrv In Depth: Exploring Mimikatz's Kernel Driver. (2020). https://posts.specterops.io/mimidrv-in-depth-4d273d19e148

[17] Intel. 2010. Evolution of Integrity Checking with Intel® Trusted Execution Technology: an Intel IT Perspective. (2010). https://www.intel.ie/content/dam/doc/white-paper/intel-it-security-trusted-execution-technology-paper.pdf

[18] Luke Jennings. 2008. Security Implications of Windows Access Tokens - A Penetration Tester's Guide. (2008). https://www.exploit-db.com/docs/english/13054-security-implications-of-windows-access-tokens.pdf

[19] Pascal Junod, Julien Rinaldini, Johan Wehrli, and Julie Michielin. 2015. Obfuscator-LLVM - Software Protection for the Masses. In *1st IEEE/ACM International Workshop on Software Protection, SPRO 2015, Florence, Italy, May 19, 2015*, Paolo Falcarin and Brecht Wyseur (Eds.). IEEE Computer Society, 3–9. https://doi.org/10.1109/SPRO.2015.10

[20] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. 2020. Spectre attacks: exploiting speculative execution. *Commun. ACM* 63, 7 (2020), 93–101. https://doi.org/10.1145/3399742

[21] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, Mike Hamburg, and Raoul Strackx. 2020. Meltdown: reading kernel memory from user space. *Commun. ACM* 63, 6 (2020), 46–56. https://doi.org/10.1145/3357033

[22] Microsoft. 2021. CryptEncrypt function (wincrypt.h). (2021). https://learn.microsoft.com/en-us/windows/win32/api/wincrypt/nf-wincrypt-cryptencrypt

[23] Microsoft. 2021. Device Input and Output Control (IOCTL). (2021). https://learn.microsoft.com/en-us/windows/win32/devio/device-input-and-output-control-ioctl-

[24] Microsoft. 2021. Driver Signing. (2021). https://learn.microsoft.com/en-us/windows-hardware/drivers/install/driver-signing

[25] Microsoft. 2022. Configuring Additional LSA Protection. (2022). https://learn.microsoft.com/en-us/windows-server/security/credentials-protection-and-management/configuring-additional-lsa-protection

[26] Microsoft. 2022. DeviceIoControl function (ioapiset.h). (2022). https://learn.microsoft.com/en-us/windows/win32/api/ioapiset/nf-ioapiset-deviceiocontrol

[27] Microsoft. 2022. memcpy, wmemcpy. (2022). https://learn.microsoft.com/en-us/cpp/c-runtime-library/reference/memcpy-wmemcpy?view=msvc-170

[28] Microsoft. 2022. System Services. (2022). https://learn.microsoft.com/en-us/windows/win32/system-services

[29] Microsoft. 2023. Blue screen data. (2023). https://learn.microsoft.com/en-us/windows-hardware/drivers/debugger/blue-screen-data

[30] Microsoft. 2023. Debugging Tools for Windows. (2023). https://learn.microsoft.com/en-us/windows-hardware/drivers/debugger/debugger-download-tools

[31] Microsoft. 2023. MmAllocateContiguousMemory function (wdm.h). (2023). https://learn.microsoft.com/en-us/windows-hardware/drivers/ddi/wdm/nf-wdm-mmallocatecontiguousmemory

[32] Microsoft. 2023. OneDrive Personal Cloud Storage. (2023). https://www.microsoft.com/en-us/microsoft-365/onedrive/online-cloud-storage

[33] Microsoft. 2023. Setting Up Local Kernel Debugging of a Single Computer Manually. (2023). https://learn.microsoft.com/en-us/windows-hardware/drivers/debugger/setting-up-local-kernel-debugging-of-a-single-computer-manually

[34] Thomas Morris. 2011. Trusted Platform Module. In *Encyclopedia of Cryptography and Security, 2nd Ed*, Henk C. A. van Tilborg and Sushil Jajodia (Eds.). Springer, 1332–1335. https://doi.org/10.1007/978-1-4419-5906-5_796

[35] NSA. 2023. Ghidra. (2023). https://ghidra-sre.org/

[36] ntinternals. 2023. NtQueryIntervalProfile. (2023). http://undocumented.ntinternals.net/index.html?page=UserMode%2FUndocumented%20Functions%2FNT%20Objects%2FProfile%2FNtQueryIntervalProfile.html

[37] packt. 2023. Protection Rings. (2023). https://subscription.packtpub.com/book/security/9781789610789/8/ch08lvl1sec28/protection-rings

[38] Pierluigi Paganini. 2023. 'Bring your own vulnerable driver' attack technique is becoming popular among threat actors. (2023). https://cybernews.com/security/bring-your-own-vulnerable-driver-attack/

[39] Aurélien Palisse, Hélène Le Bouder, Jean-Louis Lanet, Colas Le Guernic, and Axel Legay. 2016. Ransomware and the Legacy Crypto API. In *The 11th International Conference on Risks and Security of Internet and Systems - CRiSIS 2016 (Risks and Security of Internet and Systems)*, Frédéric Cuppens, Nora Cuppens, Jean-Louis Lanet, and Axel Legay (Eds.), Vol. 10158. Springer, Roscoff, France, 11–28. https://doi.org/10.1007/978-3-319-54876-0_2

[40] ParrotSec. 2023. (2023). https://github.com/ParrotSec/mimikatz

[41] Aravind Prakash, Eknath Venkataramani, Heng Yin, and Zhiqiang Lin. 2013. Manipulating semantic values in kernel data structures: Attack assessments and implications. In *2013 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. 1–12. https://doi.org/10.1109/DSN.2013.6575344

[42] Vergilius Project. 2018. EPROCESS. (2018). https://www.vergiliusproject.com/kernels/x64/Windows%208.1%20|%202012R2/RTM/_EPROCESS

[43] rbmm. 2023. (2023). https://github.com/rbmm/DisableSvc

[44] Mark Russinovich, David A. Solomon, and Alex Ionescu. 2012. *Windows Internals, Part 1: Covering Windows Server 2008 R2 and Windows 7.* Microsoft Press, Redmond, WA.

[45] Mark Russinovich, David A. Solomon, and Alex Ionescu. 2012. *Windows Internals, Part 2: Covering Windows Server 2008 R2 and Windows 7.* Microsoft Press, Redmond, WA.

[46] Mohamed Sabt, Mohammed Achemlal, and Abdelmadjid Bouabdallah. 2015. Trusted Execution Environment: What It is, and What It is Not. In *2015 IEEE Trustcom/BigDataSE/ISPA*, Vol. 1. 57–64. https://doi.org/10.1109/Trustcom.2015.357

[47] Ruben Santamarta. 2007. Exploiting common flaws in drivers. (2007). https://shinnai.altervista.org/papers_videos/ECFID.pdf

[48] Sercan Sari. 2022. What are Rings in Operating Systems ? (2022). https://www.baeldung.com/cs/os-rings

[49] Camelia Simoiu, Joseph Bonneau, Christopher Gates, and Sharad Goel. 2019. "I was told to buy a software or lose my computer. I ignored it": A study of ransomware. In *Fifteenth Symposium on Usable Privacy and Security (SOUPS 2019)*. USENIX Association, Santa Clara, CA, 155–174. https://www.usenix.org/conference/soups2019/presentation/simoiu

[50] Emmanuel Stapf. 2022. *System Architecture Designs for Secure, Flexible and Openly-Accessible Enclave Computing.* Ph.D. Dissertation. Technical University

of Darmstadt, Germany. http://tuprints.ulb.tu-darmstadt.de/21487/

[51] Michael M. Swift, Anne Hopkins, Peter Brundrett, Cliff Van Dyke, Praerit Garg, Shannon Chan, Mario Goertzel, and Gregory Jensenworth. 2002. Improving the granularity of access control for Windows 2000. *ACM Trans. Inf. Syst. Secur.* 5, 4 (2002), 398–437. https://doi.org/10.1145/581271.581273

[52] Thibaut Frin Mouslim Bersanoukaev Lucas Giordani Thomas Boué, Théo Boulaire. 2023. PoC RobinHood Ransomware replication by using Aorus Gigabyte v1.34. (2023). https://github.com/cmd-theo/RobbinHood-attack

[53] touslesdrivers.com. 2018. Application AORUS Graphics Engine v1.3.4. https://www.touslesdrivers.com/index.php?v_page=23&v_code=57477. (2018).

[54] Shawn Wang. 2019. The difference in five modes in the AES encryption algorithm. (2019). https://www.highgo.ca/2019/08/08/the-difference-in-five-modes-in-the-aes-encryption-algorithm/

[55] Andreas Wolf, Dimitrios Simopoulos, Luca D'Avino, and Patrick Schwaiger. 2020. The PASTA threat model implementation in the IoT development life cycle. In *50. Jahrestagung der Gesellschaft für Informatik, INFORMATIK 2020 - Back to the Future, Karlsruhe, Germany, 28. September - 2. Oktober 2020 (LNI)*, Ralf H. Reussner, Anne Koziolek, and Robert Heinrich (Eds.), Vol. P-307. GI, 1195–1204. https://doi.org/10.18420/inf2020_111

A DRV2.SYS DISPATCHER FUNCTION'S DECOMPILED CODE

```
void UndefinedFunction_140007728
               (undefined8 param_1,undefined8 param_2,ulonglong param_3,ulonglong param_4,
               uint param_5)

{
  int iVar1;
  undefined8 uVar2;
  undefined8 uStackX_20;

  uStackX_20 = 0;
  if (param_4 == 0) {
    iVar1 = -0x3ffffff3;
    goto LAB_140007874;
  }
  if (param_5 < 0xc3502401) {
    if (param_5 == 0xc3502400) {
      iVar1 = FUN_1400074f0(param_2,param_3,param_4,&uStackX_20);
    }
    else if (param_5 == 0xc3502000) {
      iVar1 = FUN_140007450(param_2,param_3,param_4,&uStackX_20);
    }
    else if (param_5 == 0xc3502004) {
      iVar1 = FUN_1400072c0(param_1,param_2,param_3,param_4,&uStackX_20);
    }
    else if (param_5 == 0xc3502008) {
      iVar1 = FUN_14000761c(param_1,param_2,param_3,param_4,&uStackX_20);
    }
    else if (param_5 == 0xc350200c) {
      iVar1 = FUN_1400016c8(param_2,param_3,param_4,&uStackX_20);
    }
    else if (param_5 == 0xc3502010) {
      uVar2 = FUN_140001bc4(param_2,param_4,&uStackX_20);
      iVar1 = (int)uVar2;
    }
    else {
      if (param_5 != 0xc3502014) goto LAB_140007869;
      iVar1 = FUN_1400018a0(param_2,param_3,param_4,&uStackX_20);
    }
  }
  else if (param_5 == 0xc3502440) {
    iVar1 = FUN_140001a78(param_2,param_3,param_4,&uStackX_20);
  }
  else if (param_5 == 0xc3502580) {
    iVar1 = FUN_140001940(param_2,param_3,param_4,&uStackX_20);
  }
  else if (param_5 == 0xc3502800) {
    iVar1 = FUN_1400013d4(param_2,param_3,param_4,&uStackX_20);
  }
  else if (param_5 == 0xc3502804) {
    iVar1 = FUN_1400014e0(param_2,param_4,&uStackX_20);
  }
  else if (param_5 == 0xc3502808) {
    iVar1 = FUN_14000162c(param_2,param_4,&uStackX_20);
  }
  else {
```

```
    if (param_5 != 0xc350280c) {
LAB_140007869:
      iVar1 = -0x3ffffff0;
      goto LAB_140007874;
    }
    iVar1 = FUN_140001568(param_2,param_3,param_4,&uStackX_20);
  }
  if (-1 < iVar1) {
    (*DAT_140004908)(DAT_140004d48,param_2,0,uStackX_20);
    return;
  }
LAB_140007874:
  (*DAT_1400048f8)(DAT_140004d48,param_2,iVar1);
  return;
}
```