Chapter 1
Introduction

Most people, I think, don't even know what a rootkit is, so why should they care about it?

Thomas Hesse,
Former President of Sony's Global Digital Business

Many people associate the term *rootkit* to attacks on computer platforms. In fact, adversaries deploy rootkits to attack computer users. Rootkit-based attacks are used to conduct industrial espionage as well as political espionage, and cybercrime [see 16, pp. 22–25]. Adversaries conduct industrial espionage to steal intellectual property of competitors to slash the cost of technology development cycles. Political espionage differs from industrial espionage. In the case of political espionage the adversaries are interested in national secrets instead of novel technology. Cybercriminals use rootkits to steal internet banking credentials, passwords, and other sensitive data. Rootkits can also be utilized for conducting persistent surveillance of end users. Rootkits are also utilized by law enforcement, as well, to perform surveillance on suspects [see 16, p. 21]. But what exactly is a rootkit? Is it a *backdoor*? Is it a *Trojan horse*? In other words, what kind of malicious payload does a rootkit contain and how is the target computer infiltrated with the rootkit?

Several definitions for the term rootkit can be found in the literature such as Bill Blunden's *The Rootkit Arsenal: Escape And Evasion In The Dark Corners Of The System* [16]. His work also evaluates the rootkit definitions of Mark Russinovich (known from the *Windows Internals* series [106]) and Greg Hoglund (author of *Rootkits: Subverting the Windows Kernel* [60]). Finally, Bill Blunden came up with his own definition [see 16, p. 12]:

> A rootkit establishes a remote interface on a machine that allows the system to be manipulated [...] and data to be collected (e. g., surveillance) in a manner that is difficult to observe (e. g., concealment).

All these definitions imply an important property exhibited by rootkits in general, namely the capability of operating stealthily. Attackers deploy rootkits to camouflage the malicious code that attacks the target computer. This answers the question about the malicious payload of a rootkit.The payload can be anything that implements

P. Stewin, *Detecting Peripheral-based Attacks on the Host Memory*,
T-Labs Series in Telecommunication Services, DOI 10.1007/978-3-319-13515-1_1

malicious behavior from the user's perspective. This malicious behavior can also be a backdoor. A backdoor is used to bypass security mechanisms such as authentication requests to gain access to a computer system. A backdoor can also provide an attacker with remote access to a computer. From the attacker's point of view it makes sense to hide the backdoor. The backdoor should be used without the knowledge of the computer user. Hence, a backdoor can benefit from rootkit mechanisms. Another example for rootkit payload is a surveillance program that activates the microphone and camera of the target computer to stealthily monitor the computer user. A keystroke code logger that captures all keystrokes that are entered by the computer user is also a popular example for malicious payload.

However, the challenge for the attacker is the infiltration of the target computer platform. The attacker has to implement some kind of rootkit installer. A rootkit installer is commonly referred to as *dropper* [see 16, p. 9, 33]. Such a dropper can be based on one of the most popular infiltration mechanism, a Trojan horse or Trojan in short. The goal of a Trojan is to mislead the target computer in installing a desired program, feature or function. Instead, the user installs malicious payload such as a keystroke code logger or a backdoor. Such a payload is generally deployed in a highly privileged environment and camouflaged using rootkit techniques. Another popular infiltration approach is the exploitation of a security vulnerability. The rootkit installer could implement a so-called *exploit*. An exploit is attack code that utilizes a security vulnerability. So-called *zero-day exploits* are more threatening than non-zero-day exploits. A zero-day exploit utilizes a previously unknown security vulnerability, which can be advantageous for the attacker. It enables the attacker to conduct a stealthy infiltration of the target computer.

Another key rootkit property is that the rootkit code runs with the highest privileges as possible. The goal is to gain at least higher privileges than any potential detection mechanism. This allows the rootkit to control and modify the detection mechanism. At a certain point the detection mechanism will fail to detect the rootkit or the malicious payload that is camouflaged by the rootkit. This is the reason why attackers seek new and more powerful attack vectors. The more privileges the attacker has the more control of the target computer the attacker gains.

The goal of the attacker is to gain absolute control of the target computer. The *rootkit evolution* documents the arms race between attackers and the anti-malware community. Rootkits moved to more privileged execution environments compared to the original rootkit. In recent years [35, 36, 47, 134, 135], the rootkit evolution reached a new level. Attackers started to exploit the isolated execution environments of platform peripherals. Peripherals with a dedicated processor, dedicated memory, and a hardware feature to directly access the runtime memory of the host are able to camouflage malicious payload that attacks the target computer. Such attacks are supposed to be stealthy. No modern anti-virus like software that is available on the market considers the peripheral-based execution environments. Such software is executed on the host processor and usually only considers the harddisk and the main memory for storing malicious code.

1.1 Problem Statement

Malware is a threat for the confidentiality, integrity, and also for the availability of data. In the case of peripheral-based malware, the attacker can exploit the stealth potential of peripherals. Malware hidden in platform peripherals is not considered by anti-virus software. Depending on the peripheral, security software can not even access the inner workings of the device. For example, certain management controller have access to the whole host memory and offer remote administration features. To prevent abuse, the manufacturer applies protection mechanisms that thwart access to the inner workings of this execution environment.

The mechanism, which is exploited by peripheral-based malware to attack the host, is called direct memory access or DMA. In this work, we will introduce the term *DMA malware* for such classes of attacks. DMA malware has similar characteristics to rootkits. Current countermeasure approaches are unable to deal with the challenge of DMA malware. For example, mechanisms such as load-time integrity checks of the code intended to run on the peripheral does not prevent runtime attacks. The same is true for digitally signed firmware images. Another approach is latency-based attestation. This kind of attestation requires that a checksum be computed within a certain timeframe. Unfortunately, it also requires the modification of the peripheral's firmware and does not prevent transient attacks. Further approaches such as special monitoring and memory bus snooping are based on special hardware or hardware features. Preventing sensitive data from being present in the main memory also does not help. Such data can be dumped into the main memory via a DMA attack.[1]

A proposed countermeasure approach against DMA attacks is the utilization of a so-called *Input/Output Memory Management Unit* (I/OMMU). Such a management unit can restrict the access of peripherals to parts of the host main memory. Unfortunately, this technology has significant deficiencies. It was demonstrated that I/OMMUs can be attacked and circumvented [111, 146–148]. Hence, the I/OMMU is not necessarily trustworthy. Some operating systems such as Windows do not provide a device driver to support the I/OMMU. Additionally, not every chipset provides an I/OMMU. Furthermore, memory access policy conflicts cannot be handled by an I/OMMU. For example, Bulygin [25] demonstrated how to use a peripheral to reveal malware present in the host runtime memory. We use the same execution environment for our attack study in Chap. 4. If the I/OMMU is configured to allow the peripheral to scan the whole host runtime memory to reveal rootkits, then our attack code can also access the whole runtime memory to steal sensitive data, for example. Hence, this work does not rely on I/OMMUs as a countermeasure. Furthermore, I/OMMUs can introduce significant performance overhead [13, 150], which makes I/OMMUs undesirable in certain scenarios. Due to these considerations,

[1] Details can be found in Sect. 3.2 "Related Work–Countermeasure Approaches".

we believe that a runtime monitor that can detect malicious memory access with negligible performance overhead is missing. The absence of a runtime monitor is one of the major motivations for this work.

1.2 Research Question and Methodology

Our research interest is based on the stealth capabilities of modern x86 platforms. These capabilities are exploited by adversaries to hide malicious code as documented by the rootkit evolution, see also Sect. 2.1. This raises the question whether or not undetectable software can exist at all. To examine this question we consider the next logical step in the evolution of rootkits, i.e., exploiting platform peripherals to attack the host runtime memory.

We developed a malware *Proof of Concept (PoC)* that is executed on an isolated peripheral. The hardware of this peripheral provides access to the host runtime memory. We implemented an attack in the form of a keystroke code logger. This means that our malware searches for the keyboard buffer of the host operating system and monitors that buffer to capture keystroke codes. The evaluation of the keystroke logger led us to a follow-up research question, i.e., is the host system able to defend itself against peripheral-based host main memory attacks? To answer this question, we implemented a runtime monitor that is executed on the host CPU. With this monitor we want to demonstrate that additional (malicious) accesses to the host main memory that originate from platform peripherals can in fact be detected. We require that the host CPU-based monitor detects malicious accesses even if it is unable to access the isolated execution environment of the malicious peripheral.

We used our malware example to derive typical properties of this class of malware. Afterwards, we exploited these properties to detect memory accesses conducted by the malware. We identified a property that every peripheral-based malware that attacks the host memory exhibits. Because of this, we consider our malware proof of concept as typical for this malware class. We implemented the host CPU-based detector to reveal illegitimate memory accesses conducted by platform peripherals via direct memory access. The goal was to implement a runtime monitor that does not only cause minimal performance overhead for the host CPU, but also prevents transient attacks.

We also consider the network interface card in the last part of our research. The network interface card could also host malware. Especially in enterprise environments it is required that a computer platform reports its status to a central administrator platform. Such a status report can be modified by malware that is executed on the network interface card. Hence, we developed an authentic reporting channel. This channel helps to reveal attacks on such a status report.

Experimental Research Environment Our experimental environment is based on Intel x86 hardware. The isolated peripheral that we use for our peripheral-based malware is *Intel's Manageability Engine* (Intel ME [79]). The Intel ME is a special micro-controller that runs a powerful platform management firmware. An administrator can use the management firmware to remotely reinstall the operating system even if the operating system is not bootable and the platform is not reachable via the operating system's network stack. The ME also works when the platform is in standby or powered off. Due to these features the manufacturer Intel established protection mechanisms that cannot be circumvented without significant effort. The ME is isolated from the host system. The Intel ME environment is completely isolated from the host, whereas other peripherals can be accessed via debug registers and other mechanisms.

From a detector's point of view the ME is the worst case execution environment for hosting peripheral-based malware. The host CPU is unable to access the ME environment. We use this worst case environment for our research. We infiltrate the ME environment by applying an exploit that only works with a certain chipset.[2] Please note, this work does not aim to find undiscovered security vulnerabilities. We reused a known security vulnerability to set up our experimental environment due to the lack of an appropriate Intel developer board.

1.3 Impact of Thesis Contributions

To conduct industrial espionage or steal online banking credentials, for instance, attackers demand stealthily operating malware. Peripheral-based malware ensures that the attack remains to be undetectable. Peripherals that fulfill the requirements for stealthy malware operation are present in almost every modern computer platform. Peripherals such as video cards, network interface cards, and management controllers are part of desktop computers, server systems, and other computer terminals. Mobile phones and tablet computers also have peripherals with a dedicated processor, memory, and direct access to the host runtime memory. This means that all modern platforms are susceptible to peripheral-based malware attacks. Such malware is executed in an isolated execution environment and outside the scope of anti-virus software and security mechanisms set up by the operating system kernel. Due to the lack of a detector for peripheral-based malware and the lack of similar functionality in anti-virus software, the contributions of this thesis can have impact on the mentioned computer devices and their users. We summarize the main contributions of this thesis in the following:

- **DMA malware study**: We define DMA malware to be able to distinguish different DMA code. Such malware is executed on a peripheral and able to attack the host via direct memory access. We develop a proof of concept DMA malware implementation that is able to conduct a stealthy attack using an isolated peripheral. Our proof of concept is called DAGGER, which is derived from *DmA-based keystroke loGGER*. DAGGER can attack different host operating systems. DAGGER

[2] The exploit is only applicable to Intel's Q35 chipset with a certain BIOS version in place. Intel closed the corresponding security vulnerability by providing a BIOS update.

highlights how efficient and effective DMA malware is in practice. We identify the core properties of DMA malware to learn the properties of such malicious software. These properties are the basis for a DMA malware detector. In a first experiment we provide evidence that DMA side effects exist. We demonstrate how such an effect can be measured using common host CPU features. This is a first step for the development of a DMA malware detector. (see Chap. 4)

- **Detecting DMA malware**: We developed a monitor that detects DMA malware by comparing actual memory bus activity with expected memory bus activity. Our method is able to determine and compare actual bus activity without any firmware or hardware modification. The detector is based on a feature that implements permanent runtime monitoring and runs on the host CPU. We implemented and evaluated a PoC that we call *Bus Agent Runtime Monitor* (BARM). Our monitor implements a monitoring strategy that considers transient attacks. It does only cause negligible performance overhead. BARM can detect and halt DMA malware immediately. (see Chap. 5)
- **Authentic platform state reporting that excludes DMA malware**: We demonstrate that our detection method is also suitable in scenarios where a computer platform has to report its status to a central administrator platform. We establish an authentic reporting channel that reveals attacks conducted by malware executed on the network interface card. This means that we enhance BARM to reveal *Man-in-the-Middle* (MitM) attacks and to prevent relay attacks conducted by the network interface card. We implemented a channel to securely transmit the platform state information to an external computer. The platform state information enables a remote party to evaluate BARM measurement results. This means that the remote party can determine if its counterpart has been attacked by DMA malware. Our channel considers the host CPU as the channel endpoint and not the complete target platform. This excludes the network interface card from being part of the endpoint. We enhance BARM to account for memory bus activity that is caused by the network interface card. The enhanced BARM utilizes *OpenSSL* to implement the authentic reporting channel. We also modify the TLS handshake protocol to already account for platform state information in the very beginning of the communication session. Our modifications are still compliant to the TLS specification. (see Chap. 6)

A more detailed elaboration can be found in the corresponding chapters.

1.4 Structure of the Thesis

According to our methodology we structured this thesis as follows. In the next chapter we will introduce the required technical background, preliminaries as well as assumptions. The target platform for our evaluation is a modern Intel x86 based system, see Sects. 2.2, 2.3, 2.4, 2.5, and 2.6. These sections introduce the most important terms regarding the target platform, especially the host CPU, *Direct Memory Access*

(DMA) as well as bus master, and *Input/Output Memory Management Unit*. We also introduce our assumptions and the resulting trust and adversary model in Sect. 2.7. Chapter 3 covers related work. Since we consider both, the attack as well as attack detection and protection, we have to elaborate related work in both areas. Related works regarding DMA attacks are described in Sect. 3.1. Section 3.2 presents previous works that consider countermeasure approaches. Furthermore, we want to enable our target platform to report its status regarding DMA-based malware to an external platform. To do so, we require a communication channel that reveals MitM attacks of the network interface card. This is necessary, since we also consider network interface cards as dedicated hardware that can hide DMA attack code.

We conducted a study of DMA malware and present the results in Chap. 4. A definition for DMA malware is given in Sect. 4.1. In Sect. 4.2 we present DMA malware core functionality. The design and implementation of our DMA malware is presented in Sect. 4.3. Section 4.4 describes the evaluation of DAGGER, Sect 4.5 considers countermeasures and discusses in particular I/OMMU issues. In the same section is demonstrated how we were able to exploit these properties to demonstrate first DMA side effects. Since the host CPU is unable to directly realize illegitimate memory accesses conducted by compromised peripherals we try to provoke a side effect that occurs when a peripheral accesses the main memory.

The evidence of DMA side effects presented in Chap. 4 is the motivation for the runtime monitor that we introduce in Chap. 5. In Chap. 5 "A Primitive for Detecting DMA Malware" we demonstrate how DMA side effects can be exploited to develop a detection tool. We define a general detection model that helps us to build a detection tool, see Sect. 5.1. Afterwards we present a PoC implementation based on the popular Intel x86 platform in Sect. 5.2. We evaluate our implementation in Sect.5.3. We also test BARM with the DMA malware that we developed in Chap. 4. Finally, BARM exploits the fact that our DMA malware has to search for valuable data that causes a certain amount of bus transactions.

In Chap. 6 we enhance our detection tool to implement an authentic state reporting application. The application sends BARM measurements to an external platform. The goal is a secure communication channel that excludes malware, which runs on the network interface card, from conducting a MitM attack. In Sect. 6.1 we present a model to negotiate an authentic reporting channel. We require a secure channel such as TLS that is bound to the actual communication endpoint, i.e., to the host CPU. Our PoC implementation of our authentic reporting application is based on OpenSSL, see Sect. 6.2. This implementation section also describes the BARM enhancements that are required to consider the network interface card. The evaluation of our implementation is presented in Sect. 6.3. We also test our network related BARM enhancements with our DMA malware DAGGER. Authentic reporting channel security considerations are discussed in Sect. 6.4. Our conclusions of this thesis as well as future work are presented in the last chapter, Chap. 7.