A Practical Approach for Generic Bootkit Detection and Prevention

Bernhard Grill, Christian Platzer
Secure Systems Lab
Vienna University of Technology
{bgrill,cplatzer}@seclab.tuwien.ac.at

Jürgen Eckel
IKARUS Security Software GmbH
Vienna, Austria
eckel.j@ikarus.at

ABSTRACT

Bootkits are still the most powerful tool for attackers to stealthily infiltrate computer systems. In this paper we present a novel approach to detect and prevent bootkit attacks during the infection phase. Our approach relies on emulation and monitoring of the system's boot process. We present results of a preliminary evaluation on our approach using a Windows system and the leaked Carberp bootkit.

Categories and Subject Descriptors

D.4.6 [**Security and Protection**]: Invasive software

General Terms

Security, Design

Keywords

bootkit detection and prevention, dynamic malware analysis, x86 emulation

1. INTRODUCTION

Bootkits (BK) are a sophisticated type of malware designed to interfere with the boot process of an infected system by executing malicious code even before the OS kernel takes control. Having been around for years, bootkits are still very common as the Careto APT infection shows, which is a recent example for such an attack [6]. Typical scenarios include compromising and patching the kernel itself. Those infections are extraordinarily difficult to detect as they have full system access and can hide deeply within the OS.
In this paper we aim to thwart primary infection by implementing a detection engine that emulates and monitors the system's boot process. Virtual Machine Introspection (VMI) is used to detect malicious behavior during emulated startup in order to identify bootkit activity.

In this paper we make the following contributions:

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
EuroSec'14 April 13 - 16 2014, Amsterdam, Netherlands
Copyright 2014 ACM 78-1-4503-2715-2/14/04 ...$15.00.
http://dx.doi.org/10.1145/2592791.2592795

I. We propose techniques to separate malicious BK behavior during system boot from benign boot processes.

II. We design a system to detect and prevent bootkit attacks, and

III. we present results of a preliminary evaluation based on a prototypical implementation of our system for Windows.

2. RELATED WORK

A number of papers discuss malware behavior [20, 23], their distribution [16, 18] and detection [13]. There are several contributions dealing with automatic malware analysis like [9, 25] but few are concerned with bootkits as such [14]. In [14], the authors give an overview on offensive techniques used by bootkits and perform a classification on BKs according to the infected boot process stage. Typical malware analysis sandboxes such as e.g. Anubis [9], do not deploy any bootkit detection technology either. Other approaches utilize native systems to analyze hard disk sectors post-infection [26]. Idika et al. [12] show a survey on malware detection techniques, while [11] gives an overview on automated dynamic malware analysis. However, both papers don't deal with bootkits in particular. Virtual machine introspection (VMI) is a widely deployed technology nowadays [19].

3. BACKGROUND

Before discussing our approach in detail, we provide a brief introduction on boot procedures and how they can be exploited by bootkits.

3.1 Boot process

The boot process on x86 BIOS (Basic Input Output System) systems using MBR (Master Boot Record) is outlined in Figure 1 and can be summarized as follows:
The CPU starts in *real mode* and the BIOS locates and executes the MBR, which is located in the first 512 bytes of the first hard disk of the system. The MBR code parses the partition table (PT) to determine which partition is marked as "bootable", containing the operating system to start. The first 512 bytes of this bootable partition are called VBR (Volume Boot Record), sometimes also referred to as PBR (Partition Boot Record). The VBR contains information on the used file system and the location of the bootloader (typically only the first part of the code). The bootloader (BL) is typically located right after the VBR on the hard disk. The

BL loads further code from the disk, switches to *protected mode*, loads the kernel and finally hands over control to the kernel [7, 8].

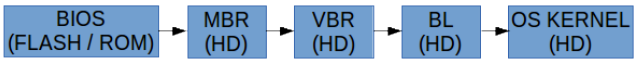

Figure 1: A boot process overview.

MBR partition tables support only four partitions limiting the partition size field to 32 bits. Since blocks of 512 Bytes are addressed, the maximum addressable size of the disk is limited to the well-known 2 TiB. Figure 2 outlines the MBR structure. The executable code is marked green, whereas the error messages are displayed in blue. The partition table shows all four partition table entries in red, starting at 0x1BE. The MBR concludes with the MBR signature 0x55AA marked in yellow.

```
Absolute Sector 0 (Cylinder 0, Head 0, Sector 1)

       0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F
0000   33 C0 8E D0 BC 00 7C FB 50 07 50 1F FC BE 1B 7C   3.....|.P.P....|
0010   BF 1B 06 50 57 B9 E5 01 F3 A4 CB BD BE 07 B1 04   ...PW...........
0020   38 6E 00 7C 09 75 13 83 C5 10 E2 F4 CD 18 8B F5   8n.|.u..........
0030   83 C6 10 49 74 19 38 2C 74 F6 A0 B5 07 B4 07 8B   ...It.8,t.......
0040   F0 AC 3C 00 74 FC BB 07 00 B4 0E CD 10 EB F2 88   ..<.t...........
0050   4E 10 E8 46 00 73 2A FE 46 10 80 7E 04 0B 74 0B   N..F.s*.F..~..t.
0060   80 7E 04 0C 74 05 A0 B6 07 75 D2 80 46 02 06 83   .~..t....u..F...
0070   46 08 06 83 56 0A 00 E8 21 00 73 05 A0 B6 07 EB   F...V...!.s.....
0080   BC 81 3E FE 7D 55 AA 74 0B 80 7E 10 00 74 C8 A0   ..>.}U.t..~..t..
0090   B7 07 EB A9 8B FC 1E 57 8B F5 CB BF 05 00 8A 56   .......W.......V
00A0   00 B4 08 CD 13 72 23 8A C1 24 3F 98 8A DE 8A FC   .....r#..$?.....
00B0   43 F7 E3 8B D1 86 D6 B1 06 D2 EE 42 F7 E2 39 56   C..........B..9V
00C0   0A 77 23 72 05 39 46 08 73 1C B8 01 02 BB 00 7C   .w#r.9F.s......|
00D0   8B 4E 02 8B 56 00 CD 13 73 51 4F 74 4E 32 E4 8A   .N..V...sQOtN2..
00E0   56 00 CD 13 EB E4 8A 56 00 60 BB AA 55 B4 41 CD   V......V.`..U.A.
00F0   13 72 36 81 FB 55 AA 75 30 F6 C1 01 74 2B 61 60   .r6..U.u0...t+a`
0100   6A 00 6A 00 FF 76 0A FF 76 08 6A 00 68 00 7C 6A   j.j..v..v.j.h.|j
0110   01 6A 10 B4 42 8B F4 CD 13 61 61 73 0E 4F 74 0B   .j..B....aas.Ot.
0120   32 E4 8A 56 00 CD 13 EB D6 61 F9 C3 49 6E 76 61   2..V.....a..Inva
0130   6C 69 64 20 70 61 72 74 69 74 69 6F 6E 20 74 61   lid partition ta
0140   62 6C 65 00 45 72 72 6F 72 20 6C 6F 61 64 69 6E   ble.Error loadin
0150   67 20 6F 70 65 72 61 74 69 6E 67 20 73 79 73 74   g operating syst
0160   65 6D 00 4D 69 73 73 69 6E 67 20 6F 70 65 72 61   em.Missing opera
0170   74 69 6E 67 20 73 79 73 74 65 6D 00 00 00 00 00   ting system.....
0180   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
0190   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
01A0   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
01B0   00 00 00 00 00 2C 44 63 A8 E1 A8 E1 00 00 80 01   .....,Dc........
01C0   01 00 07 7F BF FD 3F 00 00 00 C1 40 5E 00 00 00   ......?....@^...
01D0   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
01E0   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
01F0   00 00 00 00 00 00 00 00 00 00 00 00 00 00 55 AA   ..............U.
       0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F
```

Figure 2: MBR structure overview, taken from [2].

The difference between GPT (GUID Partition Table) and MBR is the size and quantity of supported partitions [2]. GPT maintains an arbitrary number of partitions and applies 64 bit partition size fields increasing the maximum addressable size to 2 ZiB [5]. However, an MBR is still present, it is simply followed by the GPT and features a slightly different MBR code.

3.2 Bootkits

The term bootkit is a combination of the terms "boot" and "rootkit". Bootkits interfere with the system's startup process before the OS kernel is started. Therefore, malware has to execute malicious code in any stage before the kernel is started to gain control over the system and hence interfere with the kernel boot process [14]. The stages are depicted in Figure 1.

A typical attack vector is passing arbitrary kernel parameters on OS startup to deactivate security features. More advanced attack scenarios include patching and injecting code into the kernel itself. Advanced BKs even deploy sophisticated self-protection and hiding techniques and are difficult to detect in post-infection scenarios. Figure 3 shows the boot process on an infected system with malicious code marked red. According to [14] four different types of BKs exist: BIOS-, MBR-, NTLDR- and Other-Technology-based bootkits. BIOS-based BKs write their malicious code directly into the BIOS FLASH/ROM, e.g. the research prototype IceLord [1], while MBR-based infect the MBR or VBR as TDL4, Rovnix or Gapz does [3]. NTLDR-based BKs infect the bootloader, e.g. Carberp [4], whereas some rely on other technologies, like boot.ini-based or hive-based BKs.

3.3 Kernel-level infections

x86 based Windows malware often uses drivers to infect the OS at the kernel level. It can hide deeply inside the kernel, has full system access and is therefore extremely difficult to detect [24]. Modern operating systems include advanced security features to defend against such malware. Windows, e.g. introduced kernel patch protection (PatchGuard) and driver signature enforcement policy on all x64 Windows OS to reduce the risk of kernel-level malware infections [22]. As these technologies to prevent malware from entering the kernel level became increasingly common, malware authors devised new ways to compromise the OS [22]. Recent malware uses BK technology to circumvent those defense strategies.

3.4 Int 13

Int13 is an x86 assembler instruction calling interrupt 13, which is responsible for disk control in x86 BIOS based systems. Depending on registers and memory content, the call handles hard disk access attempts (e.g. different modes of read and write requests to the disk, or getting information on the installed drives). Malware authors hook this call to transfer control flow back and forth between malicious code and the original OS.

4. DESIGN CONSIDERATIONS

With this knowledge about startup procedures and basic bootkit behavior, we finally want to design our system.

4.1 Objectives

Our goal is to detect and prevent bootkit infections during run time within the system and to detect infections from outside the system after infection. Therefore, we consider malware that tries to install a bootkit on the local hard disk to areas responsible for system startup. Furthermore, our detection technique should be deployed in a pre-existing anti-malware solution and thus, should provide the following features:

I. **Integration:** The system should not interfere with other widely used security measures such as anti-virus software, intrusion detection systems, DEP, ASLR, etc.

II. **Generic:** Our system must be capable of detecting unknown (0-day) bootkits like the recently discovered Careto BK [6].

III. **Performance:** Since our approach will be integrated in a commercial product, the overall performance impact to the hosting system must not exceed 3%.

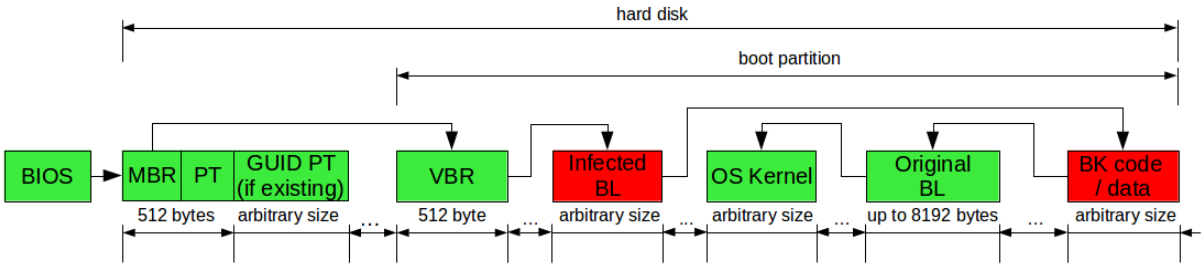

Figure 3: An example execution flow for an infected boot loader.

To meet these objectives, the following assumptions have to be made: To prevent and detect infections, the system has to be installed on the host to be defended. Furthermore, we assume a clean hosting system without any malware infections. Otherwise the malware could deploy self-protection techniques, tamper with the environment and fool the detection system. Furthermore, we assume an x86 BIOS based system using MBR boot process as described in Section 3.1 and we do not support UEFI (Unified Extensible Firmware Interface) systems.

4.2 Bootkit Behavior & Techniques

On a typical infected system, the original startup code is still executed first. Afterwards, the BKs code replacing parts of the original boot code is loaded from the disk and executed. If implemented, the BK code performs self-decryption. Subsequently, the configuration is loaded and the malware installs an Int13 hook. The hook is responsible for transferring control back to the BK after the original code has loaded the kernel. Then the original boot code responsible for loading the OS kernel is invoked. After loading the kernel into memory, the hook performs a control transition back to the malicious code. Finally, the malware alters kernel behavior and starts the OS. Due to their design, BKs have some special properties:

I. **Persistence:** They have to persist themselves on disk sectors responsible for system startup to execute prior to the kernel.

II. **Backup:** BKs have to back up the original boot code preserving the ability to finally load and start the kernel. This code is stored together with the BK configuration in a hidden section, usually at the very end of the hard disk. Since this hidden section is unknown to the OS file system it is preferably stored at the infrequently used hard disk's end and is additionally often protected by the BK after installation from accidental write request by the OS.

III. **Polymorphism:** Although polymorphism is not strictly required for BK malware, it is used in most recent samples such as Carberp, Rovnix [15], Wapomi [10], etc.

4.3 Indicators for Infection

Based on these techniques and their characteristics we define the following indicators to detect bootkit attacks:

I. **Modifying boot sectors:** Since boot sector modifications are uncommon during normal operation, we define this event as a trigger condition for our detection engine.

II. **Disk access indicator:** We monitor disk access during startup by intercepting every disk read attempt. As BKs load their configuration and the original boot code from the end of the disk, we consider loading content from the hard disk's end at boot time as malicious.

III. **Self-modifying code indicator:** Since self-modifying code should not appear in legitimate boot processes we define every attempt on code modification during startup as malicious. We define self-modification as memory areas which are written by code and executed afterwards.

IV. **Decryption routine indicator:** Because space, both on disk and in memory, is very limited in boot processes, BKs typically perform only trivial decryption routines, like simple XOR based decryption (e.g. Carberp, Rovnix [15]) or ROR (Rotate Right) based (e.g. TDL4 [10]). These decryption loops are passed several hundreds to thousands of times. Hence, we specify loops having large iteration counts and performing certain instructions as prohibited.

V. **Int13 hook indicator:** To the best of our knowledge, every bootkit performs Int13 hooking, as this is the only way to regain control after executing the legitimate boot code. Therefore we observe alteration on the Interrupt Vector Table's (IVT) target address of interrupt 13 to detect Int13 hooks during system boot.

5. SYSTEM OVERVIEW

In this section we outline our approach. Depending on the application scenario it consists of one or two major components: a kernel level driver and a detection engine. The applications scenarios are also described in this section.

5.1 Driver Component

On startup the driver scans the partition table to locate disk areas containing boot code. It observes write attempts on those areas and notifies the detection engine on write requests to these sections. Write attempts on non-monitored areas and read requests are passed through without any further processing.

5.2 Detection Engine

The engine detects bootkit attacks based on the indicators for infection proposed in Section 4.3. It uses an emulator to emulate the system's boot process using the current boot configuration and code stored on disk. The engine retrieves the initial boot code from the system's MBR and starts the

emulation. During MBR code execution, the emulator will try to load the next stage (VBR) from the hard disk performing an Int13 call. Those interrupts are intercepted by the emulator and analyzed. Write requests are handled on the emulated disk but read request are redirected to the host's physical drive getting the authentic hard disk's content. Thus, the emulator fetches physical disk content to perform emulation. Write requests in the emulator on the other hand, are not forwarded to the physical disk.
Using this approach the emulated system can access potentially malicious code stored by the BK on the host's system (e.g. further code, configuration) and fully trace the malware's execution path.
The boot process monitoring is based on VMI. It observes startup behavior and collects information relevant for compromise detection. The gathered information is returned to the engine, which decides whether or not an infection attempt has occurred. In this case, the engine restores the original boot sectors.

5.3 Application Scenarios

The system can be used in two different application scenarios: deploying the engine within the protected system to detect and prevent attacks or from outside of the system to detect bootkit infections.
Deploying the detection engine within the protected system requires a clean environment without any malware infections. This scenario uses both system components, the driver and the engine. The driver intercepts write requests to boot sectors, while the engine detects the infection attempts itself. In this scenario the system is able to detect and prevent bootkit installations.
The engine may also be used from outside a potentially infected system (e.g. a stopped virtual machine environment) to determine whether or not it is infected. In this scenario only the engine is used. The engine extracts the initial boot code from the investigation target's disk and redirects read attempts to this hard disk. As the detection engine is running outside the potential infected machine it can be used in post-infection scenarios.

6. IMPLEMENTATION

As this project is still ongoing work, we have not yet implemented all parts of the system. Furthermore, we consider integration of the system into industrial anti virus software utilizing its emulator and virtual machine environment.
We implemented a fully functional Windows driver capable of write request interception. The engine is a user-land application developed in C++. It is engineered to run on multiple platforms, but as the driver is restricted to Windows and BKs are most widely spread on this OS, it is our major testing platform. The engine emulates the boot process but does not perform an emulation of the running OS. It uses different *filters* to detect an infection. Every filter represents an implementation of the indicators for infection proposed in Section 4.3. If a single filter reports the boot process as malicious, the engine reports an infection. Consequently, the BK only has to trigger one filter condition to get detected. The engine applies the template design pattern for filter implementation to support fast and easy engine modifications. So far we have implemented the disk-access-monitoring filter and the decryption-routine filter. The loop detection algorithm is based on [17], including adjustments to facilitate instruction tracing.
Before applying filters, the engine performs a white-listing approach determining whether the executed code is a known non-malicious one to avoid false-positives. To this end, we gathered eleven distinct boot processes (MBR, VBR and BL) from uninfected Windows systems (XP to Win7 and Server 2003 to Server 2008, x86 and x64 mixed). This approach proved to be quite effective, as boot code changes very infrequently e.g. XP SP0-3 all have the same boot code. Since white-listing existing operating systems is not generic though, we use the proposed filters to detect infections. If the executed code is not white-listed, the engine proceeds detection by applying the filters.
On every scan request, the engine initializes the emulator with the initial boot code and starts the emulation. On every hard disk read, the emulator instructs the engine to load the corresponding hard disk content. During runtime, the emulator collects information on the executed code via VMI. For now we use the emulated instruction counter (e.g. 500.000) as exit criterion for the emulation. After boot simulation the emulator delivers the collected information to the engine. The engine distributes the needed information to every filter, each of which decides whether or not malicious behavior has been detected.

7. EVALUATION

To evaluate our approach, we used a Virtual Machine with Windows XP SP3 and 512 MiB RAM running on a Win7 SP1, Intel Core2 Duo E6550 @ 2.30 GHz using VMware Workstation 10. We installed the driver and the engine in the virtual machine to measure the performance impact of the driver and evaluate the engine.

7.1 Driver Performance Measurement

To evaluate the performance impact of our on-write-access-driver we copied 5.06 GiB of data (31,508 files in 4,802 directories) with and without the driver. We performed the copy procedure five times and resetted the virtual machine after every copy. Table 1 outlines the results. We measured an average copy time of 19 min 57 sec without the driver and 20 min 09 sec with the driver installed, resulting in an average performance overhead of 1.0% during full load. The copy resulted in 128,724 handled write requests and 140,511 handled read requests by the driver. For comparison, we also measured the average handled read and write requests during 20 minutes runtime in an idle system. Without any user interaction, the driver handled a negligible amount of 59 read requests and 409 write requests.

5.06 GiB copy time without driver	19:57
5.06 GiB copy time with driver	20:09
Performance overhead	**1.0%**
Handled read requests (copy)	140511
Handled write requests (copy)	128724
Handled read requests (IDLE)	59
Handled write requests (IDLE)	409

Table 1: Overview on the performance measurement results for the driver.

7.2 Engine Evaluation

For the preliminary engine evaluation, we use the leaked Carberp bootkit from [4]. This choice was motivated by the availability of the full malware's source code.

We infected the virtual machine with the malware and used the second application scenario described in Section 5.3, hence detecting the BK in a post-infection scenario. Both implemented filters detected the installed bootkit, while none of the eleven captured benign boot processes were flagged malicious. Figure 4 outlines the output of the decryption loop filter, detecting the self decryption performed by Carberp. The self decryption loop's entry point is located at 0xd000008c9, which was iterated 1,217 times performing XOR as decryption instruction. The second alert was caused by Carberp's persistence method, which places bootloader code at the end of the disk and therefore triggers our disk read access filter.

```
=========== printing potential decryption loop info ===========
loop entry point: 0xd00008c9
loop exit point: 0xd0008ea

printing loop iteration information:

loop iteration counter: 1217
instruction count of loop iteration: 7

printing instructions:
0xd00d8c9: 33c2      xor         AX, DX
0xd00d8cb: 268905    mov         [ES:DI], AX ; 9f00:00ca = 0
0xd00d8ce: 83c602    add         SI, 02
0xd00d8d1: 83c702    add         DI, 02
0xd00d8d4: e212      loop        d8e8
0xd00d8e8: 8b04      mov         AX, [DS:SI] ; 0d00:0344 = cc9c
0xd00d8ea: ebdd      jmp         d8c9
===========    potential decryption loop info end    ===========
```

Figure 4: Output of the decryption loop information printed by the corresponding filter.

8. LIMITATIONS & DISCUSSION

Naturally, our approach comes with a set of restrictions. We already mentioned that we do not support UEFI as it fundamentally differs from BIOS/MBR based boot processes. We also don't support GPT yet but since it uses the same concept as MBR PT, we will include it at a later point. The driver component which handles disk accesses is currently available for Windows only, restricting the modus operandi for other operating systems to the post-infection use case. Furthermore, our approach is only effective for MBR/VBR and NTLDR-based BKs, which are also the most widely spread BK types.

8.1 Evasion Techniques

There are several techniques to detect a system emulator [21]. In our case, there are two possibilities: Either the emulator is detected before the malware infects the target host, or a well-crafted bootkit detects the emulator while being executed. The first case is undoubtedly preferable since it results in an uninfected, protected system. In the following iteration we discuss possibilities on how to achieve this:

- **Driver/engine detection:** Before infecting the boot process, malware could try to detect our system by probing for the engine e.g. with a full disk search or detecting the presence of the on-write-access-driver.
- **Modifying boot sectors trigger:** As bootkits have to modify data on the hard disk responsible for booting, to interfere with the start-up process, malware would have to remove the on-write-access-driver to bypass the triggering condition for the detection engine. This scenario assumes sufficient permissions by the attacker to remove the driver and the malware would have to restart the system, probably alerting the user.

The second case assumes that an infected bootloader behaves differently under emulation compared to native execution. It is important to note here, that capabilities during boot time are limited, making a successful implementation of the following techniques very difficult.

- **Environment detection:** During boot process emulation, bootkits could try to detect the emulator by fingerprinting the environment , e.g. by detecting the used CPU in combination with other peripherals.
- **Instruction counter exhaustion:** Due to performance reason we limit the amount of executed instructions in the emulator. The bootkit could exhaust the instructions counter before performing malicious activities.
- **Disk read access filter:** The malware could store its data in unsuspicious disk sectors to evade the filter, inducing more risk for the BK to accidentally be overwrite by the OS, because it is unaware of the disk sectors allocated by the bootkit. The BK could also store its data in the file system's slack space, but using this approach would introduce several problems and certain complexity to the malware, e.g. the BK would have to be aware of the file system's implementation, know the location of every single utilized file and its slack space and observe copy, move, update and delete operations on all utilized files to keep the data consistent.
- **Self-modifying code filter:** The bootkit could simply refrain from using self modifying code and decryption routines, but would consequently be susceptible to common pattern-based detection.
- **Int13 hook filter:** To the best of our knowledge every bootkit performs Int13 hooking, as this is the only way to regain control after executing the legitimate boot code. Therefore we do not know any way to circumvent this filter, though this conjecture is part of future research.

9. FUTURE WORK

Our main objective is to implement our approach in a commercial scan engine. To this end, we will perform an exhaustive evaluation with existing bootkits and operating systems. This set of malicious and benign boot code must not produce false-positives. Furthermore, we are confident to be able to detect all currently available bootkits since none of them exhibit sophisticated evasion techniques to the best of our knowledge. Another improvement is planned for the stop criterion of the emulator. In the current implementation it executes instructions even after the bootloader handed control over to the kernel. By introducing stopping emulation after this point, we can further increase the system's overall performance. Finally, we will check whether

every bootkit relies on Int13 hooking and if any benign boot process is capable of triggering false positives. If not, we could remove the white-listing functionality altogether.

10. CONCLUSION

In this paper we presented a novel bootkit detection and prevention system. We presented several techniques to separate normal from malicious behavior during system startup. The approach is based on VMI for boot process emulation and monitoring to detect bootkit infections. The systems consists of two major components: an on-write disk access driver, protecting disk sectors responsible for booting and the detection engine evaluating if the system is infected or not. The detection engine may either be installed within the system to detect and prevent attacks or from outside of the system detecting malware in post infection scenarios. Our first evaluations were carried out on Windows using the leaked Carberp bootkit.

Acknowledgments

We would like to thank IKARUS Security Software GmbH for supporting this research project. Additionally, our thanks go to Christian Ondracek for providing his knowledge on Windows driver development and implementing the driver. Furthermore, we would like to thank the reviewers for their excellent input. The research leading to these results has received funding from the European Union Seventh Framework Programme under grant agreement n° 257007 (SysSec).

11. REFERENCES

[1] BIOS Rootkit: Welcome home, my Lord! `http://blog.csdn.net/icelord/article/details/1604884`, 2007, Last Accessed: 2014-03-11.

[2] Win2k Master Boot Record (MBR) revealed! `http://thestarman.narod.ru/asm/mbr/Win2kmbr.htm`, 2010, Last Accessed: 2014-03-11.

[3] Advanced Evasion Techniques by Win32/Gapz. `http://www.welivesecurity.com/wp-content/uploads/2013/05/CARO_2013.pdf`, 2013, Last Accessed: 2014-03-11.

[4] krab.rar - leaked Carberp malware source code. `https://mega.co.nz/#!0YsXWBRD!CMqd9nrm1d0XABKlifI9vmxprpQ6RnfsdhBHeKrDXao`, 2013, Last Accessed: 2014-03-11.

[5] GUID Partition Table. `https://wiki.archlinux.org/index.php/GUID_Partition_Table`, 2014, Last Accessed: 2014-03-11.

[6] Kaspersky Lab - Unveiling "Careto" - The Masked APT. `https://www.securelist.com/en/downloads/vlpdfs/unveilingthemask_v1.0.pdf`, 2014, Last Accessed: 2014-03-11.

[7] System Initialization (x86). `http://wiki.osdev.org/System_Initialization_%28x86%29`, Last Accessed: 2014-03-11.

[8] x86 real mode. `https://www.princeton.edu/~achaney/tmve/wiki100k/docs/Real_mode.html`, Last Accessed: 2014-03-11.

[9] U. Bayer, C. Kruegel, and E. Kirda. Ttanalyze: A tool for analyzing malware. In *15th European Institute for Computer Antivirus Research (EICAR 2006) Annual Conference*, 2006.

[10] H. Blinka. AVG - An Int 13 trick from the new Wapomi sample. `http://blogs.avg.com/news-threats/int-13-trick-wapomi-sample`, 2012, Last Accessed: 2014-03-11.

[11] M. Egele, T. Scholte, E. Kirda, and C. Kruegel. A survey on automated dynamic malware-analysis techniques and tools. *ACM Computing Surveys (CSUR)*, 44(2):6, 2012.

[12] N. Idika and A. P. Mathur. A survey of malware detection techniques. *Purdue University*, 2007.

[13] C. Kolbitsch, P. M. Comparetti, C. Kruegel, E. Kirda, X.-y. Zhou, and X. Wang. Effective and efficient malware detection at the end host. In *USENIX Security Symposium*, pages 351–366, 2009.

[14] X. Li, Y. Wen, M. Huang, and Q. Liu. An overview of bootkit attacking approaches. In *Seventh International Conference on Mobile Ad-hoc and Sensor Networks (MSN), 2011*, pages 428–431. IEEE, 2011.

[15] A. Matrosov. ESET - Rovnix bootkit framework updated. `http://www.welivesecurity.com/2012/07/13/rovnix-bootkit-framework-updated`, 2012, Last Accessed: 2014-03-11.

[16] N. P. P. Mavrommatis and M. A. Monrose. All your iframes point to us. *USENIX Security Symposium*, 2008.

[17] T. Moseley, D. Grunwald, D. A. Connors, R. Ramanujam, V. Tovinkere, and R. Peri. Loopprof: Dynamic techniques for loop detection and profiling. In *Proceedings of the 2006 Workshop on Binary Instrumentation and Applications (WBIA)*, 2006.

[18] A. Moshchuk, T. Bragin, S. D. Gribble, and H. M. Levy. A crawler-based study of spyware in the web. In *NDSS*, 2006.

[19] K. Nance, B. Hay, and M. Bishop. Virtual machine introspection. *IEEE Computer Society*, 2008.

[20] M. Polychronakis and N. Provos. Ghost turns zombie: Exploring the life cycle of web-based malware. *LEET*, 8:1–8, 2008.

[21] T. Raffetseder, C. Kruegel, and E. Kirda. Detecting system emulators. In *Information Security*, pages 1–18. Springer, 2007.

[22] J. Rutkowska. Rootkit hunting vs. compromise detection. *Black Hat Federal*, 2006.

[23] S. Small, J. Mason, F. Monrose, N. Provos, and A. Stubblefield. To catch a predator: A natural language approach for eliciting malicious payloads. In *USENIX Security Symposium*, pages 171–184, 2008.

[24] J. Wilhelm and T.-c. Chiueh. A forced sampled execution approach to kernel rootkit identification. In *Recent Advances in Intrusion Detection*, pages 219–235. Springer, 2007.

[25] C. Willems, T. Holz, and F. Freiling. Toward automated dynamic malware analysis using cwsandbox. *IEEE Security and Privacy*, 5(2):32–39, 2007.

[26] Y. Zhu, S. L. Liu, H. Lu, and W. Tang. Research on the detection technique of bootkit. In *International Conference on Graphic and Image Processing 2012*, 2013.