FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Bootkits Revisited - Detecting, Analysing and Mitigating Bootkit Threats

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering & Internet Computing

eingereicht von

Bernhard Grill
Matrikelnummer 1028282

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung: Ao. Univ. Prof. Dr. Wolfgang Kastner
Mitwirkung: Dr. Christian Platzer

Wien, 26.09.2016

Bernhard Grill
(Unterschrift Verfasser)

(Unterschrift Betreuung)

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.ac.at

Erklärung zur Verfassung der Arbeit

Bernhard Grill
Werdstrasse 121, 8003 Zurich, Schwitzerland

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit - einschließlich Tabellen, Karten und Abbildungen -, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

WIEN, 26.09.2016 Bernhard Grill

(Ort, Datum) (Unterschrift Verfasser)

Acknowledgements

I would like to thank Ikarus Security Software GmbH, for helping me finding the initial topic and letting me perform the first evaluations for the thesis. Furthermore, I would like to express my gratitude for the *Researcher Mobility Program for Systems Security* from the *SysSec* project, which enabled me to perform a student exchange to VU University Amsterdam. Thereby, I would like to thank Prof. Herbert Bos and his amazing group at VU University for hosting, providing me great input & support, their sincere inclusion in their group and the fun moments. I would also like to thank all the reviewers for their feedback, including the great input we received from anonymous conference reviewers. In addition, I would like to thank Prof. Wolfgang Kastner and Dr. Christian Platzer for their supervision, feedback and patience. Last but not least, I would like to thank Andrei Bacs, Jürgen Eckel and Christian Ondracek for the awesome collaboration and their help.

Abstract

Bootkits (short for boot + rootkit) are one of the most advanced, powerful and persistent threats exploited in sophisticated malware attacks. In order to investigate this attack vector in greater detail, we performed a large-scale bootkit malware analysis utilizing Windows XP and Windows 7. The data set consists of samples spanning a timer period over 8 years to gain detailed insights into the bootkit malware economy. Grounded on the analysis results of 26,378 malware samples, novel techniques to detect and stop bootkit attacks have been developed. The proposed detection techniques are based on anomaly identification utilizing dynamic analysis during the system's boot phase. Furthermore, two novel techniques to prevent bootkit infections have been developed. The first one, is grounded on strictly blocking the malware's initial persistence and infection vector, while the second approach relies on emulation and monitoring the impact of disk modifications targeting the system's boot process. Furthermore, this work showcases the historic evolution of bootkit techniques starting in 2006 and presents an outlook on their potential future evolution.

Kurzfassung

Bootkits (kurz für boot + rootkit) sind eine der fortschritlichsten, mächtigsten und langlebigsten Techniken, die in moderner Schadsoftware angewendet werden. Um dieses Bedrohungsszenario im Detail zu analysieren, wurde eine großangelegte Bootkit-Schadsoftware-Analyse basierend auf Windows XP und Win7 durchgeführt. Der Versuchsdatensatz besteht aus Schadsoftware, die eine Periode von über 8 Jahren umfasst, um eine detaillierte Einsicht in die Bootkit-Szene zu erhalten. Basierend auf der Analyse von 26.378 Schadsoftware-Proben, wurden neuartige Techniken zur Erkennung und Prävention von Bootkitangriffen entwickelt. Die vorgeschlagenen Erkennungstechniken basieren auf Anomalieerkennung und dynamischer Analyse während des Bootprozesses. Weiters wurden zwei neue Mechanismen zur Verhinderung von Bootkit-Infektionen entwickelt. Erstere basiert darauf, die Infektion durch Blockierung von Schreibzugriffen auf Festplatten-Bereiche, welche für den Bootvorgang relevant sind, zu verhindern, während der Zweite auf der Emulation und Überwachung der Auswirkungen von Festplattenänderungen, welche für den Bootvorgang wichtig sind, beruht. Weiters gibt diese Arbeit einen Einblick in die historische Entwicklung von Bootkit-Technolgie und bietet einen Ausblick auf mögliche zukünftige Entwicklungen.

Contents

CHAPTER 1

Introduction

Bootkits are a special type of malware constructed to attack an operating system's (OS) boot process. They are able to execute malicious code very early during the boot process enabling it to keep control throughout the execution of the infected system's boot phase. In most cases, the boot sequence is diverted before the bootloader is even executed, rendering any anti-virus-related (AV) protection virtually useless. Combined with the fact that the startup code is rarely modified on a typical system, this allows bootkits to survive for a significant time once the target system is infected and furthermore, makes it significantly more difficult to detect those kinds of malware infections. They were highly popular in the 80s and 90s [87]. In the following years, they faded into oblivion but returned with a vengeance starting in 2006. Some of the most sophisticated and hard to remove malware families nowadays exploit bootkit technology.

Recently introduced Windows security mechanisms such as driver signing policy [63] or kernel patch protection [60] forced malware writers to develop new techniques in order to gain kernel level privileges again. An increasing number of attacks are using bootkits to go undetected by deployed countermeasures such as anti-virus (AV) software and protection techniques provided by the operating system itself, e.g. the Windows PatchGuard. Those techniques can be circumvented utilizing bootkit technology as shown by malware families such as TDL4 and XPAJ [15], [19]. A well known and nowadays highly popular mass malware bootkit family is *TDSS* [81]. Additionally, federal *'Remote Surveillance and Forensic Solutions'* offered by e.g. Hacking Team [9] or FinFisher [6] utilize bootkits in their surveillance software [7]. Furthermore, GrayFish, a highly sophisticated malware recently used by the infamous *Equation group*, was discovered by Kaspersky Labs [47]. The *Equation group* is suspected to be directly operated or highly connected to the NSA (American National Security Agency). Hence, also state sponsored or federal secret services are utilizing bootkit technology.

Currently, almost all protection techniques share a weakness in common. Those tools are either implemented in the operating system itself or rely on the underlying OS, as it is the case with anti-virus solutions. Therefore, malware managing to subvert the kernel e.g. by infecting the OS's boot process is capable to run undetected. Most protection mechanisms become active too late during the startup process to catch active bootkits, e.g. AV solutions or even Microsoft's ELAM module (Early Launch Anti-Malware) [59], a circumstance which is partially owed to the increasing complexity of today's operating systems. This circumstance ultimately enhances the window for an attacker to execute malicious code undetected during boot time. As a consequence, malware authors show renewed interest in bootkits and their techniques.

1.1 Goals and Objectives

The following objectives have been defined for the master thesis:

- Techniques should be developed in order to (i) detect, (ii) analyse and (iii) prevent bootkit attacks.
- Two different scenarios must be supported. To detect if (i) a given binary installs a bootkit during execution and (ii) a given hard disk image is infected by a bootkit. Furthermore, it must be possible to analyse the attack in further details in both cases.
- Tools should be developed which help a human to detect and analyse bootkit malware and quickly get a basic understanding of its behaviour. It should be able to deal with both previously described scenarios.
- The approach must be able to detect unknown (0-day) threats, while the prevention technique should not interfere with widely used security measures as ASLR, DEP or anti-virus systems.
- The thesis targets Windows binaries as Windows is the dominating OS for both x86 systems and malware.
- The work focuses on BIOS based computer systems as virtually all bootkit malware samples in the wild target this configuration. [40]
- Utilizing the previously mentioned detection & analysis system, we want to perform a large-scale and long-term bootkit malware analysis to get a detailed insight into the current state of bootkit technology.

1.2 Scientific Outcome

Parts of this work have already been published while writing the master thesis to get feedback from reviewers and additional input during scientific conferences. The publications are available under [41], [40] and [21]. Those publications contain sub-elements of this work which are extended, connected and presented in more detail in this master thesis.

1.3 Structure of this Thesis

This thesis is structured as follows: Chapter 2 provides necessary background information such as: x86 real and protected mode, interrupts and their processing, the boot process for an OS, an introduction into bootkit techniques, slack space and QEMU. Chapter 3 describes our approach and heuristics to detect bootkit infections with dynamic analysis, introduces into dynamic bootkit malware analysis and shows a preventive approach for bootkit attacks. Chapter 4 pictures two techniques to prevent bootkit infections. One is grounded on strictly blocking the malware's initial persistence and infection vector, while the second approach relies on emulation and monitoring the impact of disk modifications targeting the system's boot process. Chapter 5 outlines the system overview, its architecture and gives insights into the system's implementation, e.g. the utilized technologies, database model and user interface, while Chapter 6 outlines a case study by analysing a TDL4 malware sample utilizing the proposed system. Chapter 7 presents the experimental setup for the evaluation such as the utilized data set and containment policies. Chapter 8 summarizes the results for the large-scale bootkit malware analysis, the historic evolution of bootkits, and the performance of the proposed detection and prevention techniques. Chapter 10 provides the related work, Chapter 9 discusses limitations, evasion techniques and misdetections of our approach and provides an outlook into potential future bootkit evolutions, while finally Chapter 11 outlines the future work and concludes the thesis.

CHAPTER 2

Background

This chapter provides necessary background information on x86's real and protected mode, interrupt processing, the Interrupt Vector Table (IVT), hooking this table and interrupt 13 which is elementary for booting and also highly relevant for bootkits. Furthermore, it explains the system's boot process, and slack space, and introduces the concept of *dark regions* which are a particular type of slack space. Finally, the chapter provides an introduction into bootkit techniques and gives an overview on QEMU.

2.1 Real and Protected Mode of x86 processors

In the following, we discuss the two modes of operation for x86, the real and the protected mode.

Real Mode

Real mode is an operation mode of the x86 processor family. When an x86 processor is powered on, it starts in real mode. In this mode, the processor behaves like a very fast 8086 processor. It does only support a basic set of instructions and the memory space is restricted to 1 MiB as the address bus is limited to 20 bit. Furthermore, it provides unlimited direct access to all memory, I/O addresses and peripheral hardware. Real mode does not provide any memory protection, virtual memory, multitasking or code privilege levels, i.e. differentiation between kernel- and user-land. In real mode, the processor offers 16 bit registers. This mode basically provides backwards-compatibility for the old 8086 processor family. [17], [18]

Protected Mode

In *protected mode*, the processor can address 4 GB of memory as it is using a 32 bit address bus. In this mode, memory protection for pages and segments, virtual memory, multitasking and privilege levels (ring 0 to ring 3), i.e. differentiation between kernel- and user-land, are available. The processor offers 32 bit registers, hence this modes is sometimes referred as “32 bit mode”.

The protected mode and its features are controlled by the control registers (CR0, CR2, CR3, CR4).

Almost all modern x86 processors support *long mode* in addition. This mode is also known as "64 bit mode" or x86_64. It allows the CPU to access 64 bit registers and 64 bit memory space. In general, long mode is rather similar to protected mode except for 64 bit support and some additional instructions. An x86_64 processor behaves in real mode exactly like a 16 bit x86 CPU, and in protected mode exactly like a 32 bit x86 processor. In the following, we will mainly deal with real and protected mode, hence we will not go into further details for long mode. [17], [16]

2.2 Hooking the Interrupt Vector Table

This section discusses interrupts, the Interrupt Vector Table (IVT), hooking and how to exploit this knowledge in order to hook the IVT.

Interrupts and the Interrupt Vector Table

An interrupt is an event triggering the execution of a particular type of function named the interrupt service routine (ISR) or interrupt handler. Each interrupt is associated with a certain number and a corresponding ISR. The details how interrupts are processed depend whether the processor is in real or protected mode but in general they follow the same procedure. We will have a closer look on interrupt handling in real mode now. [30]

In real mode, the first 1024 byte of memory (`0x00000` - `0x003FF`) are reserved for the so called *Interrupt Vector Table* (IVT). In protected mode, the IVT is named Interrupt Descriptor Table (IDT) which has a slightly different layout but having the same basic purpose. Both map interrupts and their corresponding interrupt numbers to ISRs, which finally handle them.
Basically, the IVT stores the start addresses of the ISRs' locations in memory. Every IVT entry (the ISR's start address) consists of a 4 byte address. Hence, the data structure can contain 256 interrupts as its overall size is 1024 bytes. The target address for interrupt 0 is stored at address `0x00000` - `0x00003`, for interrupt 1 from `0x00004` - `0x00007`, and so on. If an interrupt is raised, the CPU utilizes the address at the corresponding position in the IVT to locate the fitting ISR and finally executes the routine. Interrupts are called using the `int` assembler instruction, e.g. `int 4` calls interrupt number 4, i.e. the fourth entry in the IVT. Figure 2.1 outlines the previously described interrupt handling process with the main program, the IVT and the ISR. [30]

`Int 13h` is an x86 assembler instruction calling interrupt 13. This interrupt is responsible for hard disk control in real mode for BIOS based systems. Depending on memory content and registers, the interrupt handles hard disk operations (e.g. read and write requests to the disk or getting information on the installed drives). `Int 15h` provides an interface to miscellaneous system services such as getting the memory size or switching to protected mode. Both interrupts are highly interesting for bootkit authors, as discussed in Section 2.5. [41]

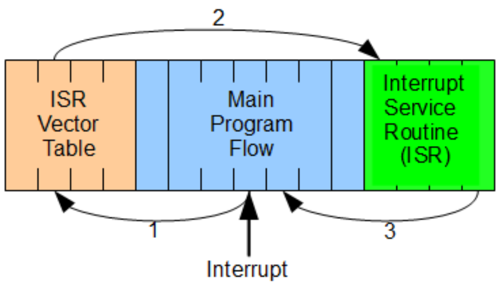


Figure 2.1: The interrupt handling process utilizing IVT and ISR [85]

Hooking the Interrupt Vector Table

Hooking is a very powerful technique to modify the behaviour of a program by intercepting function calls. Code artefacts handling those intercepted function calls are named “hooks”. Figure 2.2 provides an introduction into the hooking technique which we will examine in further detail now.

Let’s consider a table with function addresses (call table), e.g. the third entry in the figure points to the start of the function “LegitimateRoutine” at address `0x7c820011`. Furthermore, let’s assume there is a second function called “ImposterRoutine” at address 0xc01dbeef. Now, one could replace the original address with the address of the second function in the call table. So whenever the first function should be called, the second (the so called hook or hooking function) is actually executed. The hook function will process the function call and may or may not forward execution to the original procedure. Hooking can be used for benign purpose, e.g. to perform additional security checks prior to a function call, or for malicious reasons, e.g. by rootkits to hide certain files from the user.

Of course, the IVT is also hookable by overwriting the target address in the IVT with the address of a different function. Hence, whenever the interrupt is triggered, the hooking ISR is invoked instead of the original one. In the IVT case, the hooking function typically calls the original ISR in order to perform the intended functionality, e.g. load content from the hard disk into main memory. `Interrupt 13` is a very popular hooking target for bootkits, as it is used to load further code into memory, e.g. by the bootloader or the OS kernel. Furthermore, it is an easy technique to transfer control flow back and forth between malicious code and the original OS (we will examine this feature in further detail in Section 2.5).

2.3 The Boot Process

In this section, we will discuss how the boot process for BIOS based x86 system works, which is outlined in Figure 2.3. After a power-on self-test (POST), the processor starts in real mode and runs the BIOS (Basic Input/Output System). The BIOS locates and loads the Master Boot Record (MBR) at address `0000h:7C00h` and hands over execution to the MBR. Be aware of the address `0000h:7C00h`, as it will get important again for bootkit detection in Section

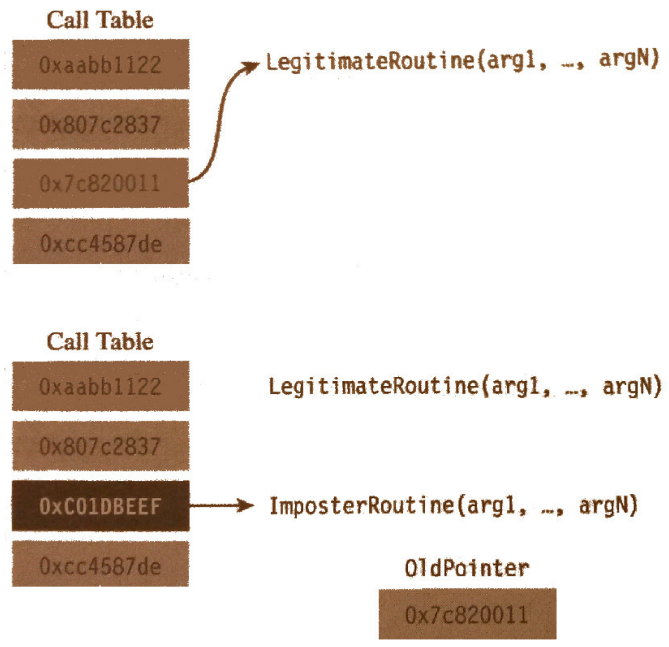


Figure 2.2: Normal vs. hooking scenario [30]

3.4. The MBR are the first 512 bytes of the system's first hard disk. It consists of two parts: a partition table and code. The code parses the partition table (PT), locates the partition marked as "bootable", i.e. the partition containing the OS, and redirects execution to the Volume Boot Record (VBR).
The VBR are the first 512 bytes of the bootable partition and is loaded by the MBR again at the address `0000h:7C00h`. The VBR also consists of code and parameters relevant for the further boot process, e.g. the first stage bootloader's location, the partition's file system, the amount of hard disk blocks (smallest managed section of a hard drive) forming a cluster. The VBR code parses the relevant information, locates the bootloader's (BL) first stage on disk and loads the BL into memory.
Typically, the bootloader consists of several stages, whereas every stage is loaded by the previous stage and gets more powerful (e.g. the first stage does not understand, the second partially and the third fully understand the file system's structure). Finally, the bootloader prepares the environment, switches to protected mode, loads the kernel and the first low level drivers, and hands over control to the OS kernel. [41], [40], [14], [17]

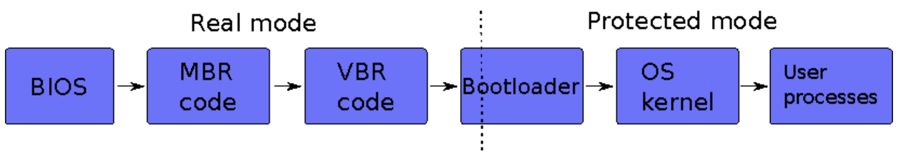
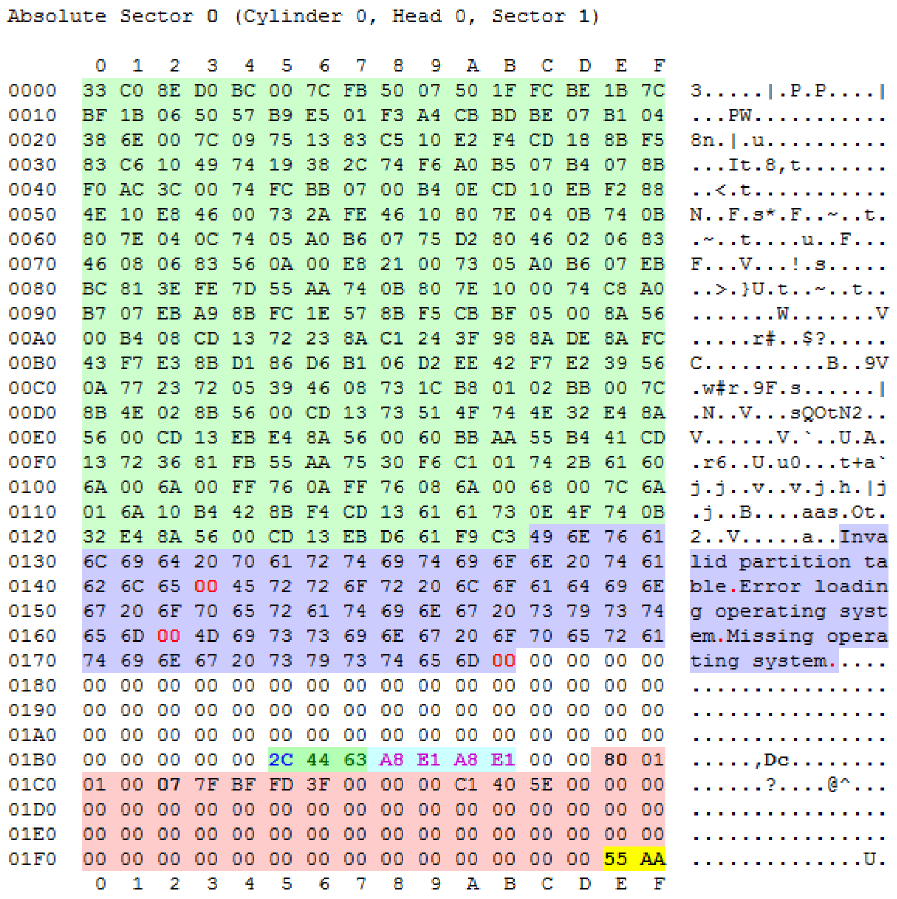


Figure 2.3: Boot sequence for BIOS based systems [40]

MBR partition tables do only support up to 4 partitions, whereas each partition's size field is limited to 32 bits. Considering hard disk block sizes of 512 bytes, this results in the well known 2 TiB maximum hard disk size for BIOS based MBR systems. Figure 2.4 shows the structure of the MBR. The executable code (the first 446 bytes are colored in green), the error messages are blue. The partition table is marked in red, whereas one can recognize the four partition table entries, each consisting of 16 bytes. Both the MBR and the VBR conclude with magic number signature 0x55aa, outlined in yellow. [41], [40]



```
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 2.4: MBR structure overview (code in green, configuration in blue, partition table in red, magic boot number in yellow) [2]

In addition to the MBR partition table schema, there is also the GUID Partition Table (GPT) style. The difference between GPT and MBR partition table is the size and quantity of supported partitions. GPT uses 64 bit partition size fields increasing the maximum addressable size to 2 ZiB and supports an arbitrary number of partitions. The GPT partition table is located in the disk blocks succeeding the MBR, starting with byte 513 of the system's first hard disk. However, an MBR is still present, it is simply followed by the GPT and features a slightly different MBR code. We will not deal with the GPT partition scheme as they are rather uncommon for BIOS based systems and bootkits in the wild are not applicable to GPT disks, yet [12]. Anyhow, the proposed techniques would also work for GPT with slightly increased implementation effort. [8], [41]

2.4 Slack Space and Dark Regions

Operating systems store files on the hard drive in blocks (the smallest managed section on a hard drive). If the file size is not a multiple of the block size, then some space is unused. This unused space is referred to as slack space, i.e. slack space is the storage area between the end of the file and the end of the last used block. From the OS's point of view, slack space is occupied, although it is practically not used.

In the following, we will introduce *dark regions* (DR) which are related to slack space. We define a dark region as a contiguous physical hard disk area which is not part of the file system. As dark regions are not part of a file system, they are not accessible during normal system operation by the user. Examples for DRs are: the MBR, the area between the MBR and VBR, the bootloader, the inter-partition gaps and the sectors after the last partition (these gaps size up to several MB on typical systems). Figure 2.5 shows a common hard disk structure and its corresponding DRs. The DRs are colored in gray. [40]

The OS's initialization may use certain DRs, e.g. the MBR, VBR and bootloader are used during the boot phase while inter-partition gaps are not. According to our definition of a DR, all boot code up to the bootloader is located within dark regions. Only very rare events like major OS updates or re-installs could ever modify dark regions, which makes them very interesting for persistence reasons. We do not refer to DRs for any hidden areas within file system.

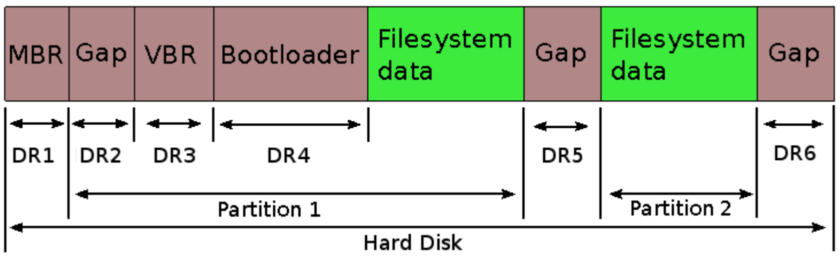


Figure 2.5: Common dark region (DR) structure - DRs in gray, file system in green [40]

2.5 Introduction to Bootkit Techniques

The name *bootkit* refers to "boot" and "rootkit". As already discussed, bootkits are hijacking an operating system's boot process before the OS kernel is executed. Therefore, malware has to execute code in any stage prior kernel execution to gain control over the system.
Simple attack scenarios result in passing additional kernel parameters during the boot process in order to disable OS security mechanisms, such as Windows PatchGuard and kernel patch protection [60]. Advanced attack vectors include injecting code into the kernel or patching it directly in memory, e.g. to disable security features or provide kernel rootkit capabilities.

Furthermore, sophisticated bootkits deploy advanced self protection features which make them especially difficult to detect in post-infection scenarios. For example, Carberp installs a self protection driver filtering hard disk reads which target the infected MBR respectively the hidden bootkit data storage outside of the partition by returning the original (not infected) MBR or empty data when trying to read those sections. In addition, Carberp also prevents those disk sectors against being overwritten by malware removal tools [84]. Modern BKs often utilize encryption and self-modifying code to hide the secret storage's content. The original (benign) boot code provided by the OS is normally not modify during execution and is not encrypted. This property is exploited by one of the proposed BK detection techniques discussed in Section 3.6. [57], [29]

Windows PatchGuard and Kernel Patch Protection

In 2006, bootkits were rediscovered as Microsoft introduced Windows PatchGuard and kernel patch protection (KPP) in their x64 operating systems [82]. Prior to that, malware writers could enter kernel level, which is mandatory for sophisticated and powerful malware, by simply loading a malicious kernel driver at runtime. But PatchGuard not only prevents loading drivers which were not signed my Microsoft, the kernel patch protection also prohibits unauthorized modification (e.g. hooking) of certain kernel data structures, such as the IDT (Interrupt Vector Table), GDT (Global Descriptor Table), SSDT (System Service Descriptor Table). Hence, bootkits became very popular again for advanced malware. [60], [63]

Bootkit Types

According to Li et al. [54], there are 4 different types of bootkits:

- **BIOS-based:** They write their malicious payload directly into the BIOS, such as the research prototype IceLord does. [1] Those bootkits are very rare and most of them are proof-of-concepts as they typically have to target particular BIOS and hardware versions and must be customized therefore. Only a few such BK type samples have been spotted in the wild. [39] [86]

- **MBR-based:** Those bootkits infect the MBR or VBR such as the malware families TDL4/TDSS, Rovnix, Sinowal, Gapz or Pihar do. [4] [3]

- **NTLDR-based:** NTLDR-based BKs infect the bootloader stage, e.g. Cidox/Carberp exploit this infection vector. [5]

- **Other Technologies:** There are also some other techniques as boot.ini-based or hive-based bootkits. This type is also hardly seen in the wild.

As MBR and NTLDR-based BKs are the dominant types (and others are hardly spotted in the wild), we will focus on those two types for this work.

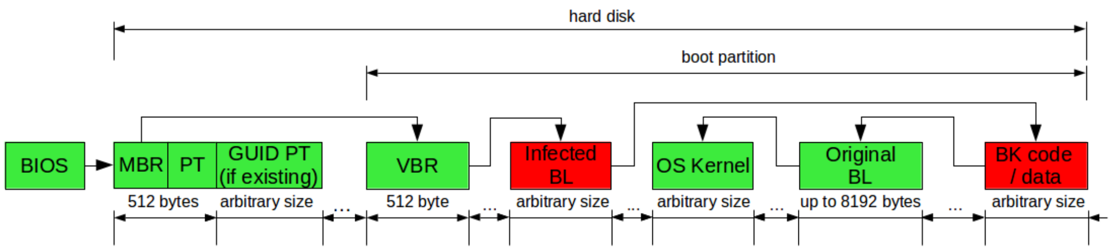


Figure 2.6: Execution flow for a bootkit utilizing the bootloader as initial infection vector (benign areas in green, infected in red) [41]

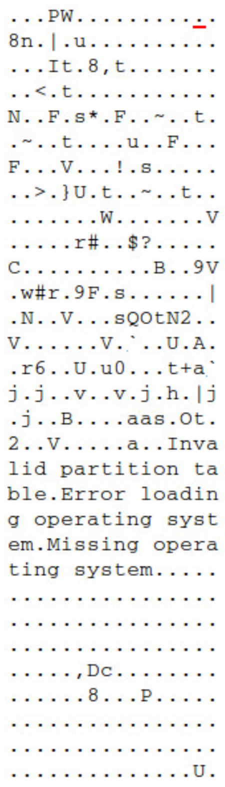
Bootkit Infections

Figure 2.6 outlines an example boot process for a bootkit exploiting an infected bootloader stage. The BIOS starts the system and hands over execution to the MBR which loads and redirects execution to the VBR. The code in the VBR loads the bootloader (BL) infected by the bootkit. In this example, the BL represents the initial infection vector. The malicious BL loads and executes further code from the system's disk (the dark region at the end of the drive). This code performs additional malicious activity and setup tasks, such as hooking particular interrupts in order to regain control before the kernel is executed. Afterwards, the original BL is executed to load the kernel into memory. Then the BK regains control by utilizing a hooked interrupt and finally infects the kernel in memory. In the end, the system proceeds initializing and executing the (now infected) system. Figure 2.7 outlines an MBR infected by TDL4 and compares it to a benign Windows XP SP2 MBR.

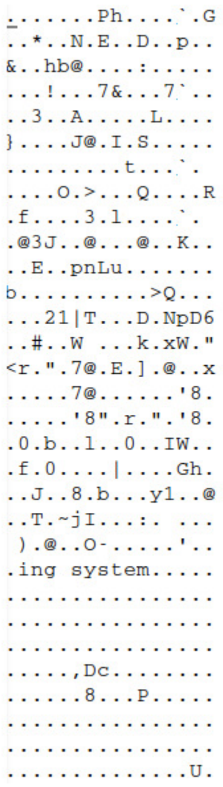
If bootkits target to modify the kernel in memory but do not want to implement the bootloader on their own, they have to regain control *after* the kernel is loaded into memory but *before* it is executed. During this time frame they can infect the kernel as it is already in memory but no protection techniques are in place, yet. Therefore, BKs typically hook the system's Interrupt Vector Table (IVT) (cf. Section 2.2). Using interrupt hooking, the malware first has to setup the hooks, rerun benign code, and regain control via the interrupt hook at the desired stage during the boot process to perform further malicious activity. `Interrupt 13` and `15` are often exploited for this purpose. `Interrupt 13` is a great fit, as by hooking this interrupt the malware can exactly determine when the bootloader loads the kernel into memory and therefore infecting it the very same time. On the other hand `interrupt 15`, is executed several times during the Windows boot process after the kernel is loaded into memory but before it is executed, making the interrupt also a highly fitting target.

2.6 QEMU

QEMU is an open source machine emulator and virtualizer. It can emulate a full computer system, including a CPU, motherboard, hard disk and various peripherals. QEMU can also emulate several different processors. It is capable of emulating processors (target processor) different than the actual physical processor (host processor). It is using so-called translation



```
...PW...........
8n.|.u..........
...It.8,t.......
..<.t...........
N..F.s*.F..~..t.
.~..t....u..F...
F...V...!.s.....
..>.}U.t..~..t..
.......W.......V
.....r#..$?.....
C..........B..9V
.w#r.9F.s......|
.N..V...sQOtN2..
V......V.`..U.A.
.r6..U.u0...t+a`
j.j..v..v.j.h.|j
.j..B....aas.Ot.
2..V.....a..Inva
lid partition ta
ble.Error loadin
g operating syst
em.Missing opera
ting system.....
................
................
................
.....,Dc........
......8...P.....
................
................
..............U.
```



```
.......Ph....`.G
..*..N.E..D..p..
&..hb@....:.....
...!...7&...7`..
..3..A.....L....
}....J@.I.S.....
..........t...`.
....O.>...Q....R
.f....3.l....`.
.@3J..@...@..K..
..E..pnLu.......
b..........>Q...
...21|T...D.NpD6
..#..W ...k.xW."
<r.".7@.E.].@..x
.....7@......'8.
.....'8".r.".'8.
.0.b..l..0..IW..
.f.0....|....Gh.
..J..8.b...y1..@
..T.~jI...:. ...
 ).@..O-.....'..
.ing system.....
................
................
................
.....,Dc........
......8...P.....
................
................
..............U.
```

Figure 2.7: Benign MBR of a Windows XP SP2 (left), MBR infected by TDL4 (right)

engines to convert target processor instructions to instructions of the host processor. Therefore, it is able to run OSs and programs dedicated for a particular CPU (e.g. ARM) on a different machine (e.g. x86). The presented solution is based upon QEMU, and will be described in further detail in the following chapters.

CHAPTER 3

Detecting Bootkit Infections

The section describes the following bootkit detection heuristics: *dark region modification, dark region access, interrupt hooking, executing address 0000h:7C00h, early boot network traffic and decryption loop detection*. The first three heuristics have already been described in my paper [40]. The *executing address 0000h:7C00h* heuristic was used by Haukli [42] to detect startup anomalies. The *decryption loop detection* heuristic was shortly outlined in my second paper [41].

Bootkits need to be installed on the target system before they become active. This can be achieved (a) when the target system is off and the physical disk is modified via an evil maid like attack [83], or (b) during system operation by a malware dropper. While the latter one is the common case for mass-malware infections, the first mode is preferred by targeted attacks, e.g. by secret services. This work is not interested in detecting how the dropper reaches the target system or how it starts to execute, but focuses on its actions to install the bootkit payload.
Given the characteristics described in Chapter 2, this work proposes techniques in order to detect bootkit attacks either during installation but also after the system is already infected. The dark region modification heuristic is the only exception, as it only detects bootkits during installation. All the other methods detect them both during, but also after infection (in a so called post infection scenario).

3.1 Dark Region Modification

Bootkits have to modify the target system's hard disk in order to survive system restarts and get active before any OS defensive technique. They write their code into dark regions as those areas are typically not used under normal system operation. As a result, the malicious code is not detected by file system based protection systems or in danger of being overwritten by the OS coincidentally. Furthermore, the code to access dark regions can be simpler and smaller. As previously discussed, the bootstrap code is located in dark regions (MBR, VBR, bootloader) at well known locations. Infections for MBR, VBR and BL-based BKs cannot avoid modifying at least one of those DRs. Therefore, a dark region modification (i.e. a disk write targeting a dark region) is a great indicator for compromise.

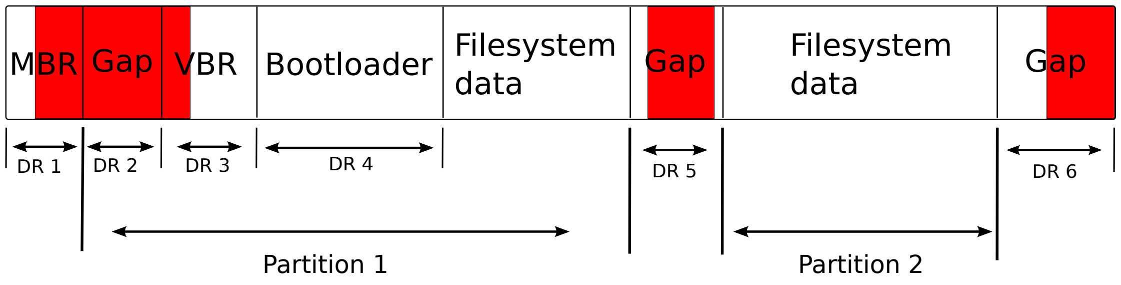


Figure 3.1: Typical hard disk layout of an infected system (indicating the utilized space by the bootkit in red)

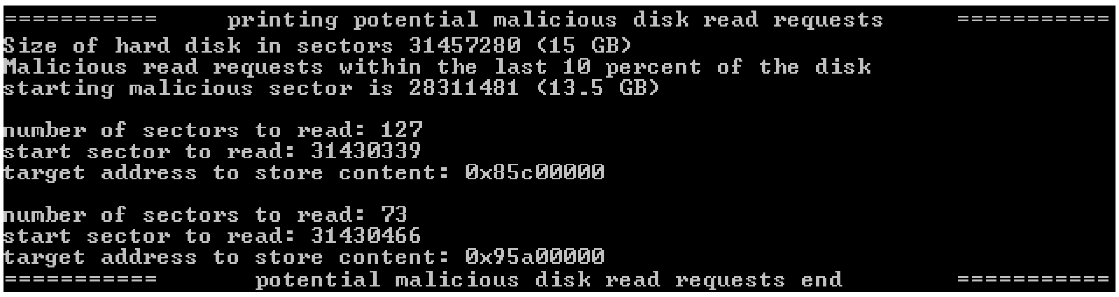
Figure 3.1 outlines a hard disk layout of an infected system with two partitions. In this case, the bootkit modified the MBR, the VBR, the gap between the MBR and the VBR, the gap between the two partitions and the gap beyond the last partition.

As long as the startup code is located on the system's disk, bootkits have to modify those areas and, hence, there is neither a way for them to escape nor to leave traces, when monitoring dark region modifications. In this scenario, false positives occur in case of legitimate DR updates, for instance during a complete system re-installation, a major system upgrade or partitioning. Modern, sophisticated bootkits need more space for their code. Therefore, they utilize further and bigger DRs. For example, they use the inter-partition gaps and the gap at the end of the disk to store additional content, e.g. the malware's config, DLLs to inject. The dark regions can also be written by a keylogger that runs within the target system and uses it as hidden storage; this is another example of a false positive induced by this heuristic (although the false positive would still point to a real infection). However, this behaviour also triggers the dark region modification heuristics.

3.2 Dark Region Read Access

The bootkit code is executed during the boot process of an infected system. During its execution, it loads additional code from its hidden storage typically located in DRs, e.g. at the end of the disk and / or between partitions. Hence, a read operation targeting a DR, excluding the dark regions containing the MBR, VBR, and bootloader (which are benign reads during startup) is an indicator for suspicious behaviour. Figure 3.2 outlines the output for a suspicious disk read during the boot process since disk sector 31430339 is located inside a dark region.

False positives for this heuristic can, for example, be caused by bugs in the OS file system driver reading beyond the limits of the file system or by a forensic application inspecting parts of the drive outside the file system. In our solution both, the dark region modification and dark region read access technique have been implemented in QEMU's disk access system.



```
==========     printing potential malicious disk read requests      ==========
Size of hard disk in sectors 31457280 (15 GB)
Malicious read requests within the last 10 percent of the disk
starting malicious sector is 28311481 (13.5 GB)

number of sectors to read: 127
start sector to read: 31430339
target address to store content: 0x85c00000

number of sectors to read: 73
start sector to read: 31430466
target address to store content: 0x95a00000
==========        potential malicious disk read requests end        ==========
```

Figure 3.2: Sample output for the dark region read access heuristics

3.3 Interrupt Hooking

Bootkits need to be executed at least once but normally several times and at different boot process phases. As already discussed in Section 2.2, interrupt hooking is a great technique in order to achieve this goal. Although for protected mode, interrupt hooking was allowed in the past, this is no longer the case since kernel patch protection was introduced in Windows [82]. Furthermore, a benign system does not perform interrupt hooking during real mode. Hence, hooking interrupts in real mode is another indicator of compromise by bootkits. Typical interrupts hooked in practice are `0x13` (triggered by reads or writes to disk via a BIOS function) and `0x15` (triggered by calling the BIOS function to get the memory size).
After powering on a computer, the BIOS initializes the system, e.g. prepares the IVT (Interrupt Vector Table, cf. Section 2.2) and hands over control to the MBR code. Before executing the first MBR instruction, our solution stores the IVT initialized by the BIOS. After a certain amount of executed instructions (in the utilized configuration: 10), our solution dumps the current IVT and compares it to the original one in order to detect interrupt hooking. For every detected hook, it logs the interrupt number, the original and the altered ISR (Interrupt Service Routine) target address. The current implementation does not dump the malicious ISR, as function boundary recognition is highly challenging and still a partially open research topic, as shown by Bao et al. [23].
The interrupt hooking technique does not detect interrupt detouring attacks (cf. Section 2.2). An interrupt detouring attack does not modify the ISR's address in the IVT, but the ISR's code itself, e.g. it overwrites the first instruction of the ISR with an unconditional jump to the malicious one. To detect interrupt detouring attacks, the same approach as for interrupt hooking detection can be applied, i.e. dumping either the whole memory area of the ISR code or just dumping the first e.g. 20 bytes of every ISR. Dumping the first couple of bytes for every ISR is sufficient, since detours are typically installed within the first few instructions, as developing malicious ISR functions which are executed in the middle of another ISR are highly complex. Finally, the system could periodically check whether the ISR code has been modified by comparing it to the original one. Currently, detouring attack detection is not implemented in Bootcamp, but could be done without problems investing additional effort.

3.4 Executing Address 0000h:7C00h

Benign BIOS based systems load the MBR to address `0000h:7C00h` and execute it. The MBR code locates the boot partition by parsing the partition table, loads the VBR from the boot partition again at `0000h:7C00h` and executes the code. Therefore, a benign system executes this fixed address exactly twice while booting.
Bootkits exploiting the MBR or VBR normally backup the original code in another dark region, in order to load and then run parts or the entire original code again. As the original MBR and VBR are implemented using the fixed address `0000h:7C00h` (e.g. using absolute jump target addresses), they have to be loaded to this specific address. As a consequence, the number of jumps to this fixed address is increased to at least three, which is a good indication that foreign code is intermixed with the original code. This technique was also used by Haukli [42] to detect system startup anomalies.
False positives may be triggered by major operating system upgrades which change the way bootstrapping is done, e.g. by adding more functionality to the boot phase and thus using the same techniques as bootkits but for benign purposes. Additionally, the bootkit could patch the original MBR or VBR in memory or on disk to utilize a different base address than `0000h:7C00h`. Thereby, also infected systems would execute this specific address only twice.
A user pressing the key combination CTRL+ALT+DEL generates interrupt 19, which triggers a jump to `0000h:7C00h`. Therefore, a detector utilizing this heuristic should filter out this scenario and only monitor an increased number of jumps during the short time frame of the startup.
A dropper can also generate interrupt 19 on its own in order to force an activation of the freshly installed bootkit. Combined with the previously described heuristic of dark region modification, it can give a good indication that a dropper wants to activate its payload immediately. [40]

3.5 Early Boot Phase Network Traffic

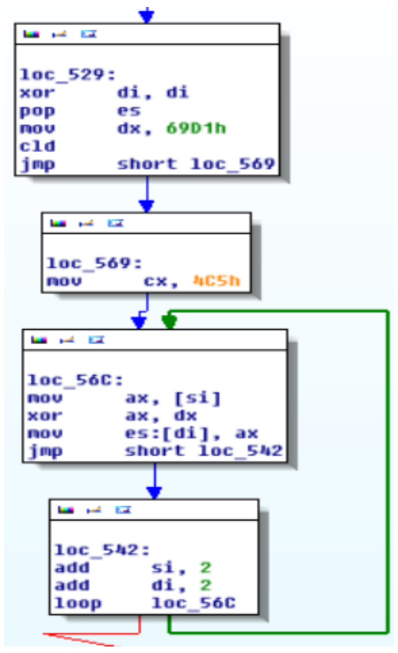
Some sophisticated bootkits use their own network driver and stack to contact Internet hosts very early during the boot phase (e.g. for updates or new configurations) even before the operating system has initialized its own networking stack. They utilize their own network stack to bypass host-based network security systems, such as firewalls or IDS. Hence, network traffic during early system startup can be an indicator for injected malicious code. A more general approach utilizing this knowledge is to perform anomaly detection on the network level in order to identify anomalies in the network traffic caused by the second network stack, e.g. specific network packet flags which are not used by the Windows network stack.

False positives may be caused by the BIOS settings to boot from the network or other BIOS level management protocols such as IPMI [46] which may generate traffic during startup. Furthermore, major OS updates introducing early or unusual network traffic can interfere with this technique.

Bootcamp does not apply this heuristic, since it takes a rather high effort to implement this technique. For example, to get a serious network traffic anomaly detection heuristics, one would have to apply machine learning (ML) to a large amount of benign and malicious network traffic to train a ML model. This effort goes beyond the scope of this work, as it is a complete topic on its own, and is considered as future work.

3.6 Decryption Loop Detection

Modern bootkits typically utilize self-decrypting code to circumvent pattern based security solutions, e.g. such as simple anti-virus solutions. As both disk and memory space is highly limited during the boot process, bootkits often utilize simple decryption operations, such as XOR based (e.g. Carberp, Rovnix [57]) or ROR (Rotate Right) based encryption techniques (e.g. TDL4 [29]). Those initial decryption routines are typically looped several hundred to thousand times. Therefore, certain instructions executed in loops with high iteration counts are another indicator of compromise. [41]

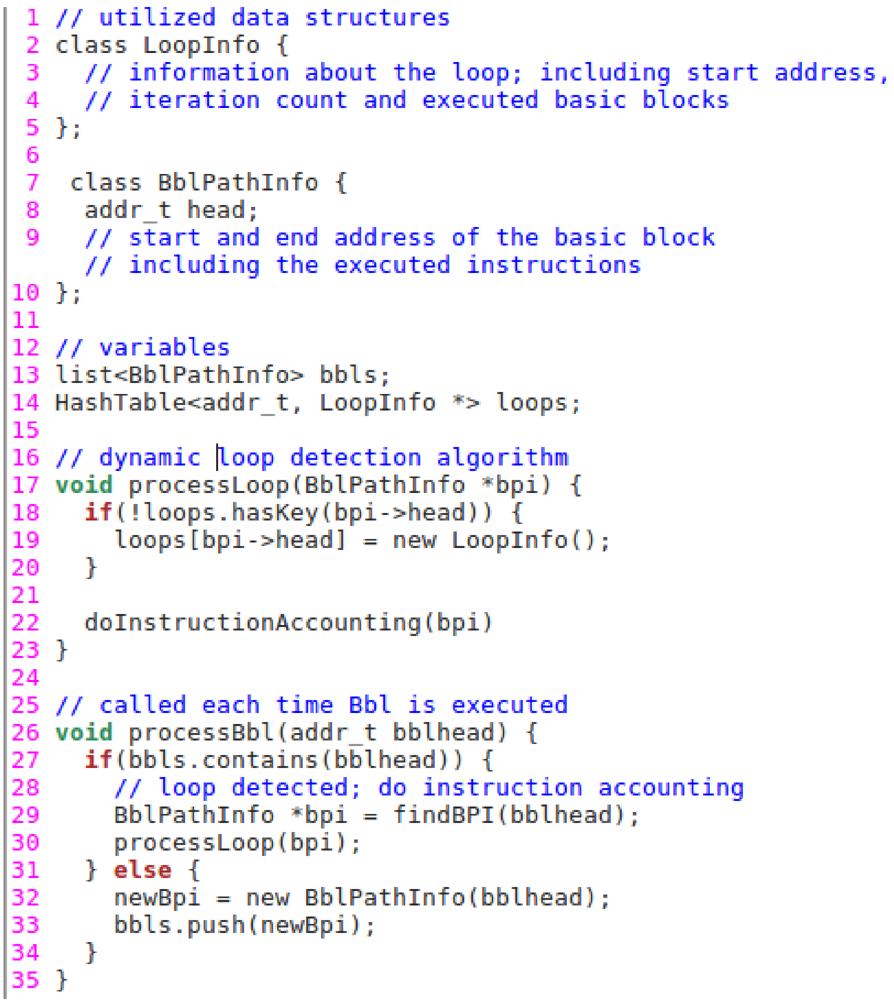


```
loc_529:
xor     di, di
pop     es
mov     dx, 69D1h
cld
jmp     short loc_569

loc_569:
mov     cx, 4C5h

loc_56C:
mov     ax, [si]
xor     ax, dx
mov     es:[di], ax
jmp     short loc_542

loc_542:
add     si, 2
add     di, 2
loop    loc_56C
```

Figure 3.3: Carberp's isomorphic self-decryption routine shown in IDA; the decryption key is 0x69d1, while the loop counter is set to 0x4c5

Figure 3.3 outlines Carberp's isomorphic self-decryption routine shown in IDA (Interactive Disassembler). Isomorphic code are instructions achieving the same semantic purpose using different instructions, e.g. `add eax, 1` and `inc eax` are isomorphic as both increment the register `eax` by one, but using different instructions. First, the decryption key `0x69d1` is loaded into the `dx` register. The decryption key is generated randomly at compile time. Afterwards, the size of the encrypted memory is loaded into the `cx` register (`0x4c5`). As the code is an isomorphic one, the size varies whenever compiled newly. Then, two bytes from memory are loaded into the `ax` register, XORed with `dx` which is the encryption key and stored in memory again (`es:[di]`). Finally, the source and destination pointers (`si` and `di` registers) are incremented by two bytes. Altogether, the loop is iterated 1221 (`0x4c5`) times, until the whole memory area is decrypted.

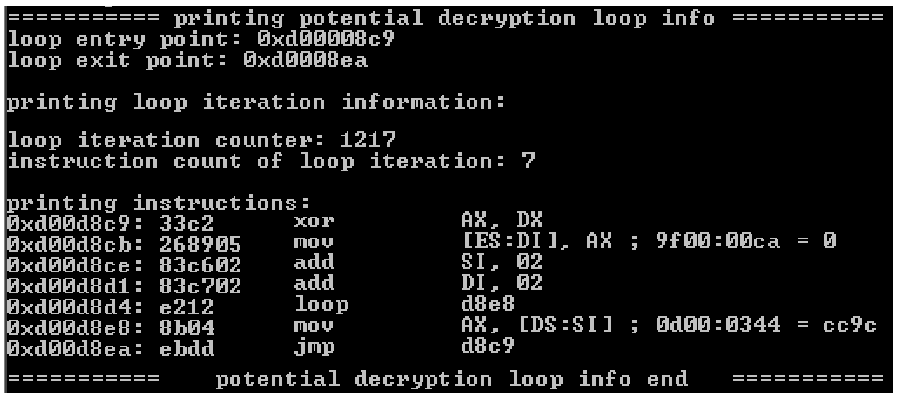
Sophisticated bootkits also utilize modern encryption techniques, such as AES or DES. But those encryption schemes are typically used much later by the bootkit, e.g. to decrypt the malicious DLLs or drivers which are later on injected into the kernel. As memory space is highly limited during the early boot phase, the initial infection vector (the MBR, VBR or the bootloader) can only be encrypted by basic encryption techniques, as previously described.
For evaluation purpose, a proof-of-concept (PoC) decryption loop detector was built. The PoC consisted of about 1,000 lines of C code. The loop detection algorithm was based on LoopProf [61] which describes a technique for dynamic loop detection. The prototype loop detector seeks for loops with more than 300 iterations and instructions which are often utilized for basic encryption such as XOR, ROR (rotate right) and ROL (rotate left).



```
// utilized data structures
class LoopInfo {
  // information about the loop; including start address,
  // iteration count and executed basic blocks
};

 class BblPathInfo {
  addr_t head;
  // start and end address of the basic block
  // including the executed instructions
};

// variables
list<BblPathInfo> bbls;
HashTable<addr_t, LoopInfo *> loops;

// dynamic loop detection algorithm
void processLoop(BblPathInfo *bpi) {
  if(!loops.hasKey(bpi->head)) {
    loops[bpi->head] = new LoopInfo();
  }

  doInstructionAccounting(bpi)
}

// called each time Bbl is executed
void processBbl(addr_t bblhead) {
  if(bbls.contains(bblhead)) {
    // loop detected; do instruction accounting
    BblPathInfo *bpi = findBPI(bblhead);
    processLoop(bpi);
  } else {
    newBpi = new BblPathInfo(bblhead);
    bbls.push(newBpi);
  }
}
```

Figure 3.4: Utilized dynamic loop detection algorithm, based on LoopProf [61]

The PoC was evaluated both on Win XP SP2 and Win7. Unfortunately, during the evaluation of the prototype it turned out that the OS itself uses a lot of such loops, which we defined as suspicious, but for a benign purpose. For example, the OS typically uses such loops to load disk content into memory or relocate memory regions during early boot phase which resulted in a lot of false positives. Whitelisting of certain loops (based on their entry and exit address) would be an option to solve this issue.



```
========== 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 3.5: Output of the decryption loop detection proof-of-concept evaluated on Carberp's self-decryption routine [41]

Figure 3.5 outlines the output for a detected decryption loop utilized by the Carberp/Cidox [52] bootkit. This is the same bootkit as shown in Figure 3.3. In this case, the decryption loop's entry is located at 0xd000:08c9 and the corresponding exit is at 0xd000:008ea. The loop is iterated 1217 times and executes 7 instructions on every iteration. The first instruction (XOR) is used to decrypt the content in memory.

Due to the significant amount of false positives with the proof-of-concept decryption loop detector, the technique is not implemented in Bootcamp. Further research in this area is necessary to reach production system readiness, which is considered as future work.

To improve the techniques' performance, one could search for self-modifying code in general by looking for memory areas which are read, modified and later on executed. To do so, taint analysis [66] could be applied. Another approach is to detect loops and entropy changes, i.e. loops and their corresponding buffers having high entropy (encrypted content) on loop entry and low entropy on loop exit (decrypted memory). This approach is also applied by Lutz et al. [56] to automatically decrypt network traffic. However, both approaches are rather complex to implement and would go beyond the scope of this work.

CHAPTER 4

Preventing Bootkit Attacks

This section presents new ways how to prevent bootkit threats. Those prevention techniques can either be implemented on the hardware level (Section 4.1) or in the OS itself, as anti-virus like solution (Section 4.2).

4.1 Bootkit Infection Prevention on the Hardware Level

This subsection discusses a bootkit prevention technique residing on the hardware level, i.e. it is especially suitable for implementation in virtualized environments. The widespread usage of virtualization makes these systems attractive targets for bootkits. Full system emulators such as QEMU [28], Bochs [31] or VMware allow the implementation of preventive measures at a very low level, inside the emulated hardware.

Based on the information of bootkits' behaviour, provided in Section 2.5, a very effective prevention measure is to restrict the modification of dark region content (be it MBR, VBR, bootloader, or inter-partition gaps) in order to defeat the persistence requirement of BKs. This prevention measure ensures that the essential infection vector is blocked. In the worst case, the malware resets the system or crashes it, but the bootstrap code and information can not be modified and therefore the system remains clean.

The prevention measure can be implemented in several ways in a full system emulator: write operations to the dark region can be diverted to another specially created shadow area, or simply replaced with a write operation of zero length. The goal is that the prevention does not crash the operating system running inside the virtual environment by generating low level disk operation errors. The second option is simpler because the management of the shadow storage space needs to be efficient if the written data is also served back to specific applications requesting it. Some droppers or bootkits may include verification steps to check whether the write operations to dark regions were successful by re-reading the disk sectors. Another more radical technique is to kill / crash the machine upon detection of a bootkit like behaviour.

Our implementation features the second option which replaces write operations to DRs with a

write operation of zero length which does not modify the disk, returns success to the initiator of the write request and as a result prevents the infection attempt.
This preventive technique is also effective against any other kind of malware utilizing dark regions, e.g. keyloggers, not exploiting bootkit technology, but using DRs to store keystrokes for data exfiltration. While it would not stop the keylogging activity itself it would stop the storage of persistent data.
This solution does not only work in virtual machines, but can be also implemented in the hard disk firmware or the disk controller of physical systems. This would require a switch (e.g. a hardware switch on the computer or a software one, for example in the BIOS) to block / allow dark region modifications in order to allow re-formatting of the disk, OS installations or major system upgrades. [40]

Another approach would be to white-list every possible MBR, VBR and bootloader and only allow the installation or startup of known benign ones. The drawback of this approach is that every single MBR, VBR and bootloader must be collected or reported in order to avoid blocking them and making the system stop working. Furthermore, such a system would induce the same problems as a DRM (digital rights management) system, as the manufacturer could determine which bootstrap code is allowed to run and which one is not [36]. For example, Microsoft locked out some Linux bootloaders with their TPM (Trusted Platform Module) and SecureBoot technique [43].

4.2 Bootkit Attack Prevention on the OS Level

Another approach is to implement the technique in an anti-virus system like solution, so inside the OS. Figure 4.1 outlines this approach. The system consists of two major components, a

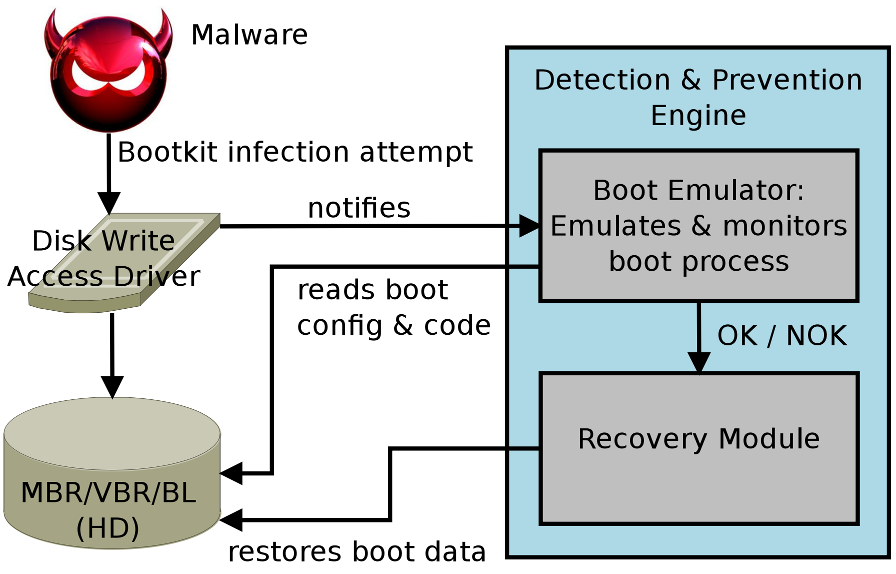


Figure 4.1: Overview on the detection and prevention system residing inside the OS

hard disk write access driver and a *detection & prevention engine*. The driver monitors every single write request to the hard disk, both malicious and benign ones. It is aware of the dark regions in the system, e.g. by scanning the partition table on startup. Whenever the driver detects a write request to a dark region, it notifies the detection & prevention engine. The engine implements the detection techniques described in Section 3. Whenever the engine is notified by the driver, the engine emulates the boot process, considering the effects on the disk by the new write request. The engine checks for bootkit like behaviour by emulating and monitoring the boot process for suspicious behaviour. Depending whether bootkit like behaviour is detected, it rejects or forwards the disk modification request to the physical disk.

Hard Disk Write Access Driver

The hard disk write access driver scans the partition table on driver startup in order to detect the system's dark regions. The driver monitors write attempts to DRs and notifies the detection & prevention engine in such cases. If the write request does not target a DR, the request is forwarded to the next handler in the driver stack.
Performance is a critical issue for this component, since every single write request (both, malicious and benign ones) is intercepted by the driver. Since the query to detect whether the write request targets a DR or not is trivial and consists only of a few 'and', 'greater than' and 'lower than' operations, the performance impact is minimal. As dark region modifications are very rare events, this solutions introduces hardly any performance overhead during normal system operation, as evaluated below.

Detection & Prevention Engine

The detection & prevention engine implements the detection techniques described in Chapter 3. Whenever the engine is notified by the driver, the engine takes the requested disk modification and the hard disk's current state as input and emulates the boot process, considering the effects on the disk by the new write request. The engine checks for bootkit like behaviour by emulating and monitoring the boot process for suspicious behaviour. During the boot process, the emulated code will execute several *interrupt 13* calls, i.e. BIOS way to access the hard disk, cf. Section 2.2. These interrupts are intercepted by the engine. Write requests are redirected to an emulated shadow disk, as the physical disk must not be altered during emulation. Read attempts are analysed and if they do not target a dark region area which is currently under modification, the system fetches the content stored on the physical disk. If the read request targets a region currently under modification, then the engine has to fetch the new content which is not on the physical disk, yet. If the engine detects a bootkit infection, it will reject the disk write attempt. Otherwise, the request is forwarded and persisted on the hard disk.
Boot process emulation induces a severe performance overhead, as it takes up to several seconds. On the other hand, this event happens only in very rare cases which should make the delay acceptable for the user. To improve the performance of several write requests triggering in a very short time, the driver could buffer the requests and instead of evaluating every single attempt on its own, pack them together and evaluate them all at once. Hence, the engine would

not fire several boot process emulations, i.e. one on every single request but collect them and perform a single one consisting of several disk modification requests at once.

Evaluation of the Disk Write Access Driver's Performance

To evaluate the approach, the driver component was implemented to evaluate and measure its performance impact on the system during normal operation. The Windows driver intercepts all disk access requests, both read and write attempts. The kernel-level driver was implemented in C and consists of about 400 lines of code. The driver checks for every request, whether it is a write attempt and its target is located within a dark region, i.e. checking if its target block is between several start and end blocks.

For evaluation, a Windows XP SP3 VM having 512 MiB RAM running on a Intel Core2 Duo E6550 @ 2.30 GHz using VMware Workstation 10 was utilized. The driver was installed in order to quantify its performance impact during normal system operation.

The performance overhead for the driver was measured by copying 5.06 GiB consisting of 31,508 files in 4,802 directories. The evaluation was performed five times each, both with and without the driver being installed. Furthermore, the VM was restarted after every evaluation to prevent caching effects. Table 4.1 shows the evaluation results. The averaged time for copying the data was 20:09 while the driver was installed and 19:57 without, resulting in an average overhead of 1.0%. The experiment invoked 128,724 handled disk write requests and 140,511 read attempts by the driver on average. An idle operating system (OS without any user interaction) generated 409 disk write attempts and 59 read requests on average in 20 minutes. Hence, the amount of processed disk accesses by the driver not triggered by our experiment is negligible. [41]

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 (copying 5.06 GiB)	140,511
Handled write requests (copying 5.06 GiB)	128,724
Handled read requests (idle system for 20 mins)	59
Handled write requests (idle system for 20 mins)	409

Table 4.1: Overview on the disk write access driver's performance [41]

The performance overhead during normal system operation, i.e. when no dark region modification is triggered, is about 1.0%, which is considered an acceptable supplement.

4.3 Discussion

Of course, using components inside the OS (such as the driver and the engine) in order to protect the system, also features some drawbacks. The biggest issue is that the malware can interfere with the system. For example, the malware could unload or crash the driver respectively the engine in order to prevent detection and install the bootkit undiscovered. Furthermore, the malware could probe the system's presence and stop infecting the machine, which is undoubtedly preferable since it results in an protected and uninfected machine, but malware could still do non-bootkit-related damage to the system. Due to performance reasons the engine has to limit the number of executed instructions in the emulator. Hence, the malware can try exhausting the instructions counter which could lead to a system infection or a DOS (denial of service) like attack.

Instead of fully emulating the boot process on every dark region modification, another idea is to white-list every possible MBR, VBR and bootloader and only allow the installation or startup of known benign ones, as already discussed earlier in this section. The drawback of this approach is that every single MBR, VBR and bootloader must be collected or reported in order to avoid blocking them and making the system stop working. But it would again suffer from the DRM problem, as the manufacturer could determine which bootstrap code is allowed to run and which one is not [36].

Anti-virus systems often utilize emulation and hence provide a contained environment for such scanning techniques out of the box. Therefore, the technique described above (having protection inside the OS) can be integrated quite easily in such existing host-based protection solutions.

CHAPTER 5

Bootcamp

This chapter provides an overview on our detection, analysis and prevention system, called *Bootcamp*, and will especially focus on dynamic bootkit analysis, Bootcamp's architecture and its software design, the software, technologies and programming languages used for implementation, the database model, and its user interface. Bootcamp fully automates bootkit malware analysis and no further user interaction is required during evaluation.

5.1 Dynamic Bootkit Malware Analysis

For dynamic bootkit analysis, we define two phases: a *bootkit infection phase* and a *bootkit execution phase*.

During *bootkit infection phase*, the malware sample installs the bootkit. Here, the bootkit might even modify the original partition table to create enough space to accommodate its hidden storage, hence creating another dark region. Some malware samples restart the system autonomously after infection in order to activate the bootkit. Such a behaviour might alert the user since the system restarts without any obvious cause. Other samples wait for a manual restart initiated by the user. In case the dropper can not carry the full payload or needs an update, it may require Internet connectivity to successfully install the BK. After infection, the dropper often deletes itself. During infection phase, Bootcamp monitors the modified dark regions, i.e. the installation of the initial infection vector and its hidden data and additional code. The monitoring is implemented in QEMU on hard disk block level, i.e. every disk read and modification request is monitored on disk block layer, e.g. a hard disk write request to block 153. The preventive measures will also become active during the infection phase in order to prevent bootkit attacks.

After some time, the system restarts the machine to enter the *bootkit execution phase*. Now, Bootcamp monitors the bootkit's behaviour during the startup process in detail. While system initialization, the bootkit is activated by the infected MBR, VBR or bootloader code and its malicious behaviour can be observed. Also during this phase, every hard disk read and modification request is monitored. After a certain time, Bootcamp kills the virtual machine and thereby the analysis for the sample ends. The approach does not rely on any information from the OS or other parts inside of the VM, e.g. kernel drivers. Hence, the dropper can not hide malicious activity, such as dark region modifications inside the environment, as all surveillance takes place in the emulated hardware itself. [40]

5.2 System Architecture & Sofware Design

System Architecture

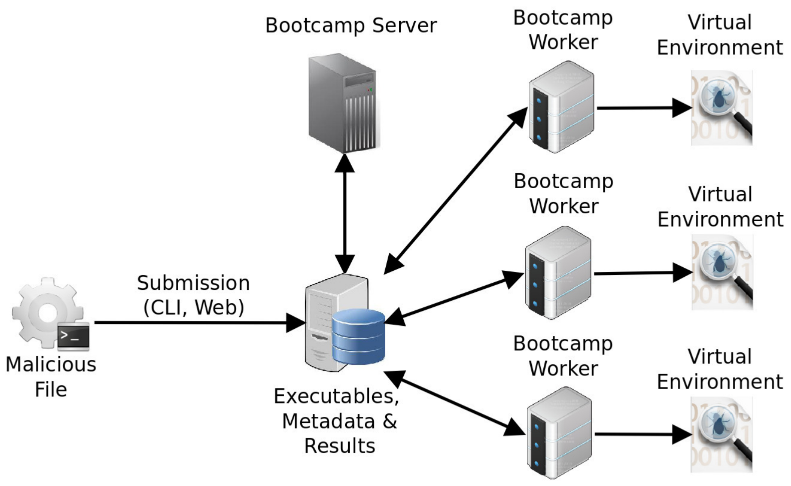


Figure 5.1: Bootcamp architecture [40]

Figure 5.1 provides an overview on the major components of Bootcamp. Binaries are stored in a database using either a command-line (CLI) or web-based submission system. The Bootcamp server consists of two components, the scheduler and a watchdog. The scheduler assigns new jobs (new submitted malware samples) to idle Bootcamp workers, while the watchdog monitors the workers whether they have crashed. The workers start an individual virtual machine (VM) per malware sample in order to dynamically analyse the file. For every analysis, the workers first enter the infection phase and observe the malware's infection techniques. The malware sample is copied via a Ruby script running inside the VM into the virtual machine utilizing a plain TCP socket connection. Furthermore, this script communicates with the worker in order to confirm that the sample has been copied and executed successfully. If preventive measures are applied, this phase also executes the prevention module. Following the system's restart, the now (potentially) infected VM performs the bootkit execution phase. During this phase, the techniques to detect and analyse bootkits are executed and the corresponding results are reported back. [40]
It is possible to host the database, server and workers on different physical machines, they just need a communication channel, e.g. ssh. This is especially relevant for workers, as they are by far the most resource intensive component. Hence, one can distribute workers to several

different physical machines. For example, it would be possible to host some parts of the system in the cloud, e.g. utilizing Amazon's EC2, Google Compute Engine, Microsoft Azure or any other private cloud.

Software Design

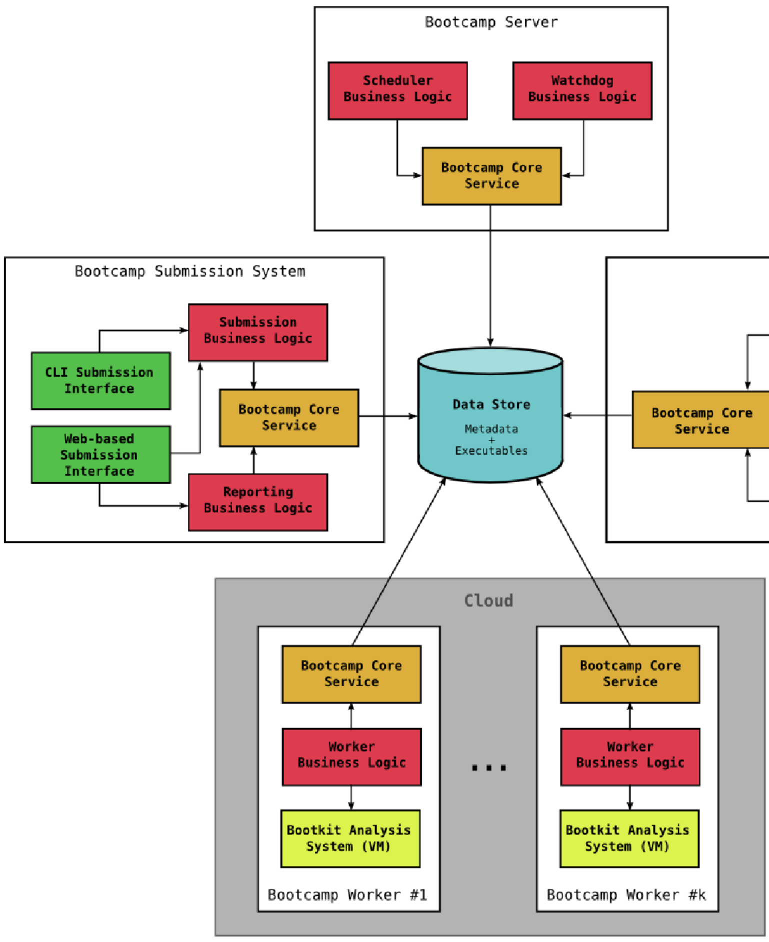


Figure 5.2: Bootcamp software components overview; core library (orange), business logic modules (red), user interfaces (green)

Figure 5.2 provides an overview on Bootcamp's software design. The core service (orange) exports basic functionality relevant for most parts of the system, e.g. get / save files from / into the database, load or persist scan results. The business logic components (red) implement the actual logic by combining the core library's exposed functionality, e.g. to submit a new file, start an analysis, store results in the database. The user interface (green) provides UI capabilities either as command-line interface (CLI) or web-based UI.

Bootcamp consists of five parts: database, submission system, server, worker and administration system. Every system follows a 3-tier architecture, structured into presentation, business logic and persistency layer using an MVC pattern. In the following, we will have a closer look on each of the components.
The database is the system's central datastore. It contains both the binaries, metadata, results and all necessary further information, e.g. scans, submissions, status information about workers. The database model is discussed in more detail in Section 5.4.
The submission system provides upload functionality, so users can submit their files for analysis. After submitting the sample, the system will create a new analysis job, wait for the scheduler to assign the job and the worker to analyse the file. Currently, there is a CLI and web-based submission system available. The web-based submission system is also capable of displaying the analysis result to the user. Further details are presented in Section 5.5.
The server consists of the scheduler and a watchdog. The scheduler assigns new files to idle workers, based on the job's priority. The watchdog monitors the workers, detects crashed ones and reschedules their incomplete scans to different workers.
The workers are responsible for analysing the submitted files. This is by far the most resource intensive job, as they start an individual virtual machine (VM) for every malware sample in order to dynamically analyse it. Upon completion, the worker stores the result in the database.
The administration system is responsible for managing Bootcamp. For example, it provides capabilities to query the submitted files, manage (start/stop) workers or observer their state, and provides a statistical overview on the system (files processed per hour, errors during processing, jobs in the scanning queue). Currently, the web-based administration system provides slightly more capabilities, therefore a direct database access and domain model knowledge is required for certain administration tasks via command line.

5.3 Utilized Technologies

Major parts of Bootcamp (server, worker, submission and administration system) are implemented in Java 1.7. Furthermore, the Java projects utilize Maven 3.3, Spring 4.0, Hibernate 4.3 and JSF (Java Server Faces) 2.2. The virtual analysis environment is a modified QEMU 2.0 including heuristics to detect, analyse and prevent bootkit attacks which are implemented in C. Additionally, MySQL 5.6 is used as data storage. Altogether, Bootcamp is built upon roughly 21,600 lines of code in about 300 files, utilizing 5 different programming languages.

Table 5.1 provides on overview on the used programming languages and the corresponding lines of code. The 16,200 lines of Java code include 4,420 lines of test cases which utilize the JUnit testing framework. The test cases are structured in 30 files and are divided into 206 individual test cases. 90% of the test cases are unit tests, while the remaining 10% are integration tests.
About 1,000 lines of the C code in QEMU was implemented by Andrei Bacs, a PhD candidate at VU University Amsterdam. He implemented the disk tracking module in QEMU which is responsible for logging every single hard disk read and modification request. This module is used by Bootcamp to track modified dark regions and is furthermore utilized by some detection heuristics. Further details on the disk tracking module are available in Bacs et al. [21], [22].

Language	Lines of code	Files	Comment
Java	16,200	227	core, server, worker, web-app for submission and administration system; including test cases
JSF	2,600	31	web UI for submission and administration
C	1,200	14	added / modified code in QEMU
Maven	650	6	config for testing, building and deployment of 5 Java projects
SQL	400	4	setup for DB
C Kernel driver	200	2	On-write-access kernel driver
Ruby	120	2	script executed in VM to copy binary into VM
Shell script	100	12	div. shell scripts for automation and administration
Config	130	7	text-based runtime configs for server, submission system and workers
Sum	21,600	302	

Table 5.1: Overview on used programming languages

The web-based UI has been implemented as a project in the ASE ("Advanced Software Engineering") lecture at TU Wien. About 4,000 lines of Java code and 2,200 lines of JSF for the web-UI have been implemented by other ASE project members.

5.4 Database Model

Figure 5.3 provides an overview on the relevant database tables. Altogether there are 16 tables but the figure focuses only on the most important ones. Furthermore, all columns have been removed due to a clear view.

Submission contains data for every single submission, independent whether the submitted file was malicious or benign. *Sample* represents the submitted files. When the same file is submitted multiple times, the *Submission* table will contain multiple entries, though the *Sample* table will contain only one, hence a de-duplication is performed. *Malware Family* contains the names of known bootkit families. The scheduler component (cf. Section 5.2) regularly scans the *Submission* table for new entries and creates *Scan* entries based on a configuration. The configuration contains the operating systems which should be used for scanning, (i.e. Windows XP and Win7) but can be easily extended. Hence, in our scenario the scheduler creates two *Scan* entries (one for XP and one for Win7) for each new submission. Every *Scan* is processed by one *Worker*. Each worker can only process suitable scans with a fitting OS, i.e. Win7 x86 workers can only process Win7 x86 scans. After performing the scan, the worker will store the output in the *Result* table. Furthermore, *Interrupt Hook* contains all hooked interrupts by the malware during analysis. As every malware can hook multiple interrupts, those results are stored in a separate table. Furthermore, there are also tables to store BLOB (binary large objects) results, such as the captured network traffic or (in the future) the dumped malicious ISRs.

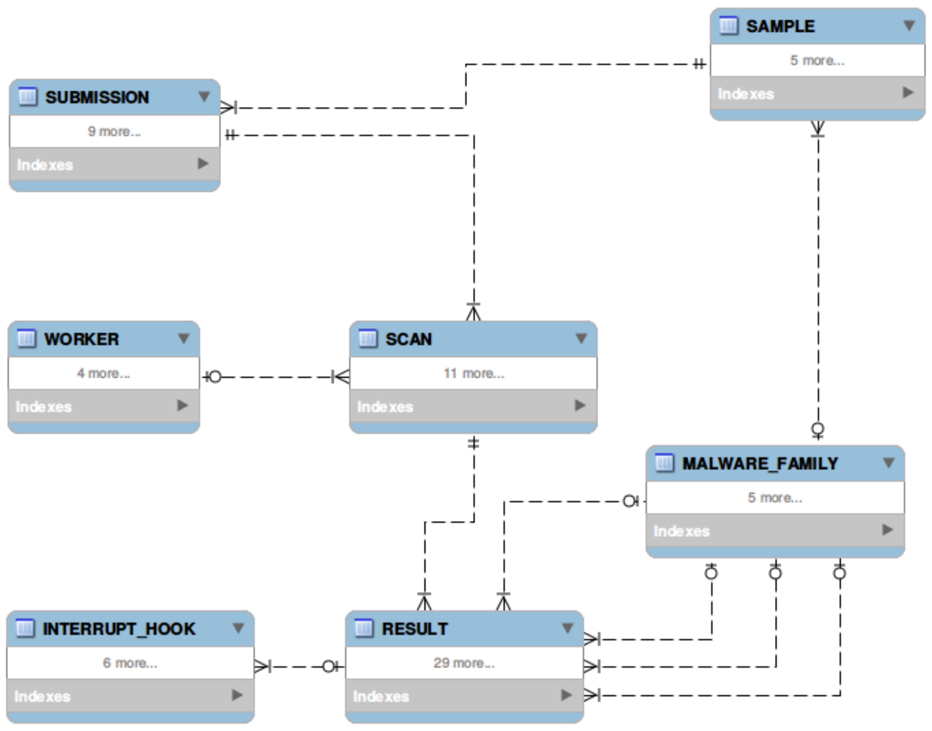


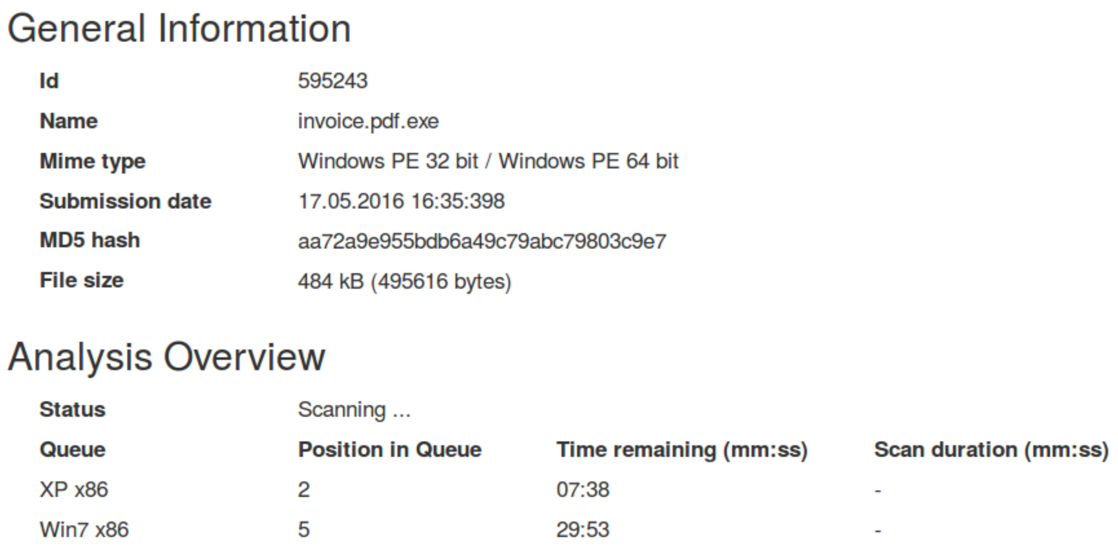
Figure 5.3: Overview on most important database tables (due to clarity only 7 out of 16 tables are displayed; columns are suppressed)

5.5 User Interface

The systems provides a command-line interface (CLI) and a web-based UI. Both provide similar capabilities, but this section will focus on the web-based one. The web UI is implemented with JSF 2.2 and PrimeFaces 5.1. PrimeFaces is a tag library for JSF and provides further UI components. As already mentioned, the web UI was implemented as a project in the ASE lecture at the university, featuring myself as head of the project team. The following will provide an overview on the public submission area to upload binaries, the corresponding scan result display and the admin section.

Public Submission and Result Area

The system provides a public upload area where users can submit suspicious binaries to be analysed. As currently the system is only able to examine Windows PE binaries, the submission system verifies the file format before initiating the analysis process. Furthermore, it also checks whether the file has already been analysed. If so, it asks the users whether they want to re-analyse the file and wait for the results or display the results of the old analysis immediately. The de-duplication feature also saves computing resources and time. After submission, the user is redirected to an overview page (cf. Figure 5.4) displaying information such as the current position in the scan queue, the time when the analysis will be finished and how long the analysis will take. The analysis results are presented in form of an analysis report (cf. Section 5.5) which


General Information

Id	595243
Name	invoice.pdf.exe
Mime type	Windows PE 32 bit / Windows PE 64 bit
Submission date	17.05.2016 16:35:398
MD5 hash	aa72a9e955bdb6a49c79abc79803c9e7
File size	484 kB (495616 bytes)

Analysis Overview

Status Scanning ...

Queue	**Position in Queue**	**Time remaining (mm:ss)**	**Scan duration (mm:ss)**
XP x86	2	07:38	-
Win7 x86	5	29:53	-

Figure 5.4: Overview page after submission, including information on the current position in the scan queue

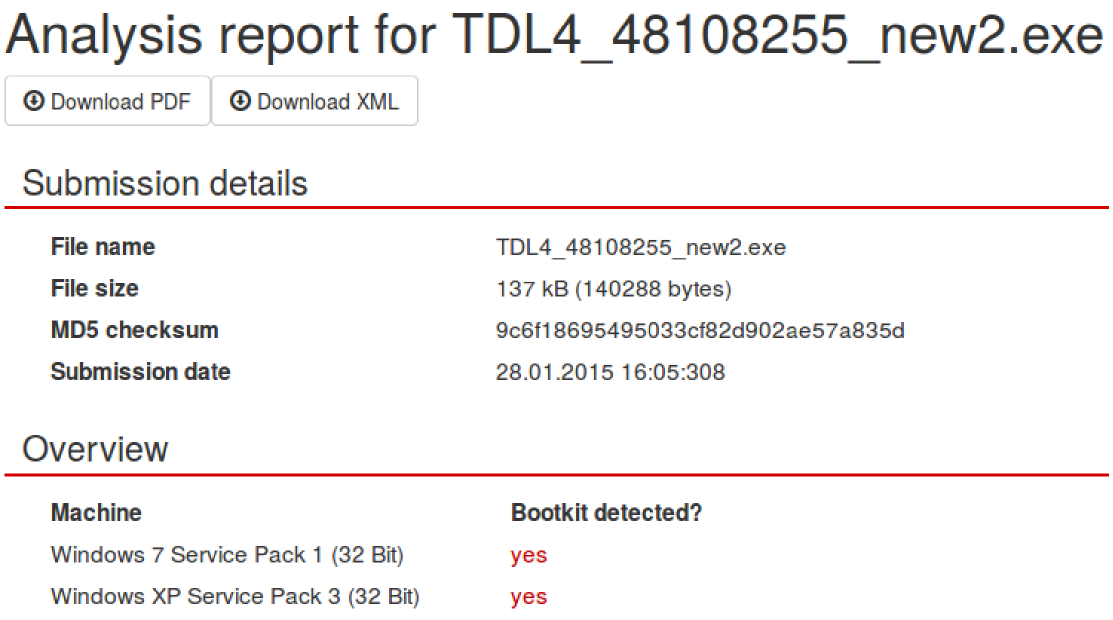
includes, e.g. general information about the executable, a classification whether the system proposes the binary to be malicious, the infection vector exploited, a network traffic dump and results of the detection heuristics. The report can be downloaded as PDF or XML document. Furthermore, it is possible to download the network traffic captured during both the infection and the boot phase. In the future, it will be also possible to download the malicious ISRs and the modified dark regions.

Admin Interface

The admin component provides means for managing and monitoring the whole system. It holds functionality to administrate workers, samples, submissions, scans or results (cf. Section 5.6). Furthermore, it offers an error log and a dashboard to get a rough overview on the system's state. The dashboard outlines information such as running and dead workers, unconfirmed errors, submission statistics (per month, week and day), an overview on the scan queue's current state or the average processing time per queue.

5.6 Application Scenarios

Application Scenarios for Bootcamp. The proposed detection heuristics can be divided into 2 different sub-types:



Analysis report for TDL4_48108255_new2.exe

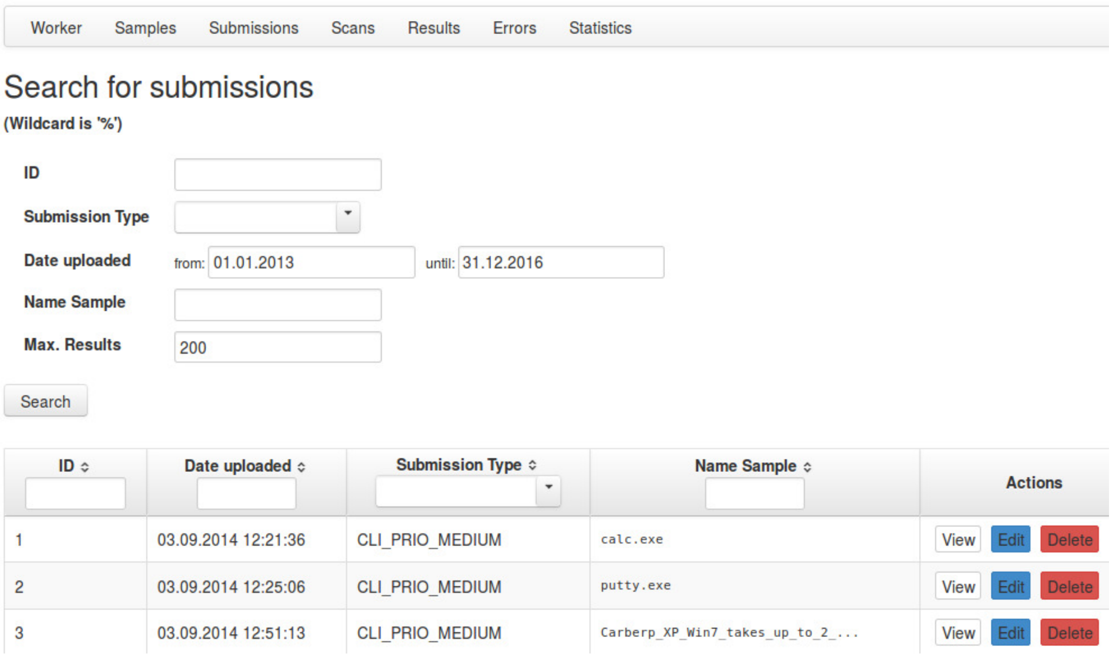
Submission details

File name	TDL4_48108255_new2.exe
File size	137 kB (140288 bytes)
MD5 checksum	9c6f18695495033cf82d902ae57a835d
Submission date	28.01.2015 16:05:308

Overview

Machine	Bootkit detected?
Windows 7 Service Pack 1 (32 Bit)	yes
Windows XP Service Pack 3 (32 Bit)	yes

Figure 5.5: A section of the analysis report for a malware sample



Search for submissions
(Wildcard is '%')

ID	
Submission Type	
Date uploaded	from: 01.01.2013 until: 31.12.2016
Name Sample	
Max. Results	200

ID	Date uploaded	Submission Type	Name Sample	Actions
1	03.09.2014 12:21:36	CLI_PRIO_MEDIUM	calc.exe	View Edit Delete
2	03.09.2014 12:25:06	CLI_PRIO_MEDIUM	putty.exe	View Edit Delete
3	03.09.2014 12:51:13	CLI_PRIO_MEDIUM	Carberp_XP_Win7_takes_up_to_2_...	View Edit Delete

Figure 5.6: Admin area for submissions

- **Install-time heuristics (type 1):** Those heuristics are able to detect BKs attacks at installation time on a running system.

- **Boot-time heuristics (type 2):** This kind of heuristics can detect infections during the boot phase of a system. Hence, they are able to detect bootkits on an already infected system.

In comparison, type 1 heuristics can only detect bootkit attacks during installation (and on a running system), while heuristics of category 2 can also detect bootkits after infection. Detecting bootkits is not restricted to VMs, one can also detect BKs on a physical system. Bootcamp can take any disk as input, e.g. another VM's disk or a physical hard disk (one wants to investigate). For example, if one wants to determine whether computer A is infected one could do the following: Plug A's disk to another computer B, mount A's disk on B and execute the detection system on B but using A's disk as input. Therefore one can detect whether A's disk is infected or not. On the other hand one can also boot a live system B on computer A. Then mount A's disk on B and do detection on hard disk A. Those use cases are especially interesting in post-infection cases, forensics or in scenarios where one wants to determine whether a system is infected. The *dark region modification* heuristic is a type 1 heuristic. However, all the other heuristics (*dark region access, interrupt hooking, executing address 0000h:7C00h, early boot network traffic* and *decryption loop detection*) are type 2 ones, hence those can be utilized to detect an infected disk even **after** infection.

CHAPTER 6

Case Study for the TDL4 Bootkit

This section presents a sample use case including an analysis report generated by *Bootcamp*.

The utilized malware sample is from the infamous TDL4 malware family camouflaging as a Java update, as indicated by the sample's name. The sample works on both XP and Win7. Network access was declined during the analysis. The system booted successfully after the infection, i.e. the malware did not crash the system. This bootkit behaves very similar on both operating systems, however this is not necessarily the normal case. Some samples only infect respectively work on one OS, while others behave differently across multiple OSs.
The following part discusses the analysis report in detail. The bootkit attacks the MBR and exploits the dark region beyond the last partition for its persistence mechanism. The *executing address 0x7c00* heuristics (cf. Section 3.4) also triggers as the address `0000h:7C00h` is executed 3 times during the boot phase. Furthermore, it hooks the interrupt `0x13`. While the original ISR is located at address `F000h:E3FEh`, however the malicious ISR injected by the bootkit is stored at `9BC0h:0050h`. Altogether the malicious ISR is executed 3,122 times during the boot phase.
The section *Reads / Writes during Infection / Boot phase* in the analysis report outlines the DRs. White indicates the absence of reads / writes during the corresponding phase. Red sections indicate disk read attempts respectively modification events during the corresponding phase. The darker the red color, the higher is the share of DR reads / modifications during this phase, e.g. in this analysis the DR at the disk's end concentrates a lot of modifications during the infection phase respectively read events during the boot phase, as this part of the disk is utilized as hidden persistence space and is hence read and written frequently. The first light red area in the section *writes during infection phase* illustrates the infection of the MBR during the bootkit attack.

The following illustration outlines the PDF version of the report. In the web UI, it is furthermore possible to download the report as XML as well as the network traffic dump during both the infection and the boot phase.

Analysis report for java_update.exe

Submission details

File name	java_update.exe
File size	610.8 kB (625468 bytes)
MD5 checksum	a53b5100a911830130826195ebb45193
Submission date	14.09.2016 22:07:445

Overview

Machine	**Bootkit detected?**
Windows XP Service Pack 3 (32 Bit)	yes
Windows 7 Service Pack 1 (32 Bit)	yes

Detailed analysis results

Windows XP

Scan information

Analysis start date	14.09.2016 22:07:512
Analysis end date	14.09.2016 22:09:512
OS name	Windows XP
OS version	Service Pack 3
OS hardware architecture	(32 Bit)
Internet access during analysis	no
Bootkit infection detected	yes
Infection phase failed	no
Number of attempts for infection phase	1
Timeout on VM shutdown occurred	yes

Sytem boots after infection	yes
Boot phase failed	no
Number of attempts for boot phase	1

Results of detected heuristics

Boot phase modifications

MBR (Master Boot Record) modified	yes
VBR (Volume Boot Record) modified	no
Bootloader modified	no

Suspicious disc accesses

Hard disk end modified	yes
Sectors between partitions modified	no

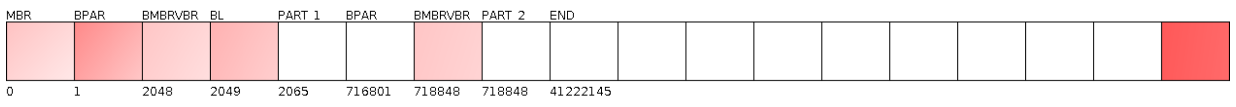
Time-based sequence of HDD modification during infection phase

no modifications

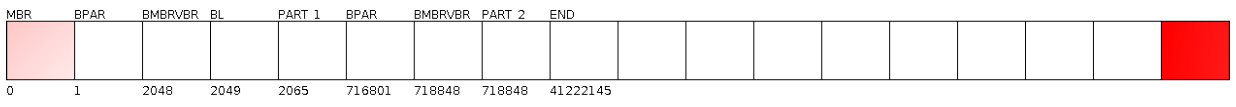
Time-based sequence of HDD modification during boot phase

no modifications

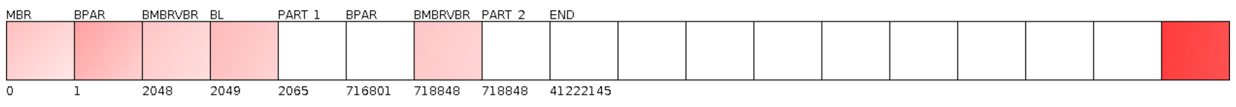
Reads during infection phase



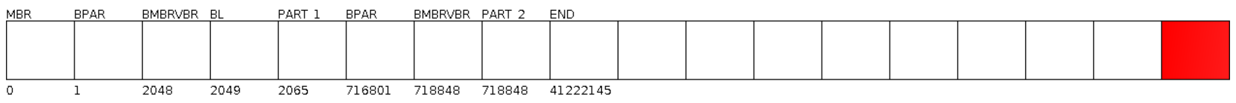
Writes during infection phase



Reads during boot phase



Writes during boot phase



Execute counts for 7c00

Infection phase:	2
Boot phase:	3

Detected BIOS interrupt hooks during boot phase

Interrupt no.	Original ISR address	Malicous ISR address	Execution count
0x13	0xf000e3fe	0x9bc00050	3122

Windows 7

Scan information

Analysis start date	14.09.2016 22:07:512
Analysis end date	14.09.2016 22:10:512
OS name	Windows 7
OS version	Service Pack 1
OS hardware architecture	(32 Bit)
Internet access during analysis	no
Bootkit infection detected	yes
Infection phase failed	no
Number of attempts for infection phase	1
Timeout on VM shutdown occurred	yes
Sytem boots after infection	yes
Boot phase failed	no
Number of attempts for boot phase	1

Results of detected heuristics

Boot phase modifications

MBR (Master Boot Record) modified	yes
VBR (Volume Boot Record) modified	no
Bootloader modified	no

Suspicious disc accesses

Hard disk end modified	yes
Sectors between partitions modified	no

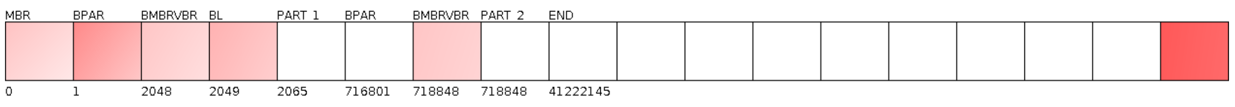
Time-based sequence of HDD modification during infection phase

no modifications

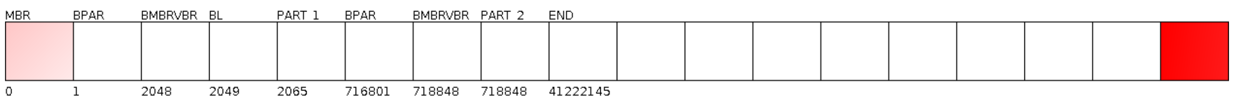
Time-based sequence of HDD modification during boot phase

no modifications

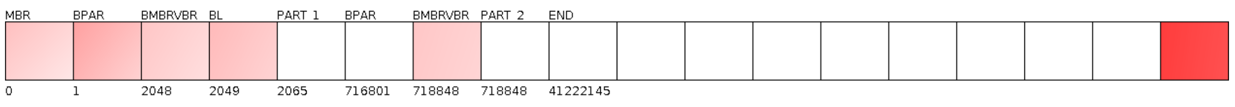
Reads during infection phase



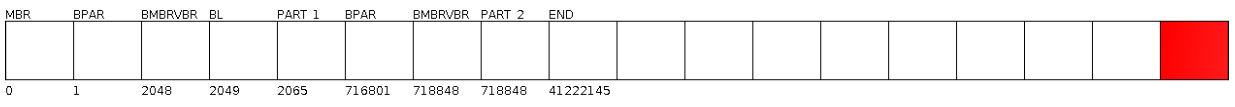
Writes during infection phase



Reads during boot phase



Writes during boot phase



Execute counts for 7c00

Infection phase:	2
Boot phase:	3

Detected BIOS interrupt hooks during boot phase

Interrupt no.	Original ISR address	Malicous ISR address	Execution count
0x13	0xf000e3fe	0x9bc00050	3122

CHAPTER 7

Evaluation

To evaluate Bootcamp, a large-scale malware analysis has been performed. This chapter discusses the utilized malware dataset, the setup for the evaluation, and the containment measures applied during the experiment.

This section is an extended version of my paper presented at DIMVA 2015 [40]. However, this work contains more details and further insights gained by analysing 2.564 additional samples, performed between February and May 2015.

7.1 Utilized Bootkit Dataset

26,378 malware samples from 29 different families were utilized to evaluate Bootcamp and gain deeper insights into bootkits' behaviour and their historic evolution. The malware families for the evaluation were selected due to their increased probability installing a bootkit according to a Kaspersky report [77] and are listed in Table 7.2. The malware samples were provided by malware collectors, samples freely available on the Internet and an anti-virus provider. The individual samples were selected based on the malware's label matching one of the 29 previously selected families. In general, all samples of a family in our dataset were analysed. If the sample count of a malware family exceeded the analysis capacity, the samples were chosen randomly within the set. Table 7.1 outlines the annual distribution of the malware dataset spanning a time period of eight years. A more exhaustive overview on the analysed malware families, their corresponding sample count and the share per family is provided in Table 7.2. [40]

Furthermore, 100 non-malicious executables were analysed in order to examine Bootcamp's behaviour for benign binaries. The non-malicious executables consisted of programs such as mathematical tools, system updates, hash sum calculators and the Sysinternals tool suite [78]. The benign samples were removed from the dataset for the final malware experiment.

First appeared in	Sample count	Share per year
2006	30	0.1%
2007	26	0.1%
2008	622	2.4%
2009	1,855	7.0%
2010	3,018	11.4%
2011	6,531	24.8%
2012	7,904	30.0%
2013	2,830	10.7%
2014	3,562	13.5%
	26,378	100.0%

Table 7.1: Distribution of the malware dataset per year [40]

7.2 Experimental Setup for the Evaluation

To achieve clean results for the evaluation, the guidelines for malware experiments described by Rossow et al. in [76] were followed. Table 7.3 outlines the checklist for malware experiments and to what extent the suggested guidelines were applied. The terms *FPs, FNs, TPs* stand for false-positives, false-negatives and true-positives. Criteria which are not applicable for this evaluation, are marked with NA.

The system was evaluated using Windows XP SP3 and Windows 7 SP1, both x86, running inside the analysis VMs. During the evaluation, the Windows firewall and Windows UAC (User Account Control) were disabled, since the deactivated security techniques allowed us to monitor more detailed malicious behaviour as discussed in Rossow's setup for malware experiments in [76].
The experiment itself was conducted between September 2014 and May 2015. Therefore, two physical systems, each running 3 analysis workers in parallel to speed up the experiments were used. Every worker consists of a virtual machine performing the evaluation composed of the *bootkit infection phase* and *bootkit execution phase*, as described in Section 5.1, for each sample. Internet access for every machine (and therefore the executed sample inside) is configurable. Experiments, both allowing and prohibiting the samples to access the Internet, were performed. In general, Internet connectivity for the samples was declined, whenever it was allowed, details on the availability are stated next to corresponding experiment. Every XP worker was provided with 1024 MB and each Win7 worker 2048 MB of main memory. Every worker needs a dedicated CPU, as sharing a CPU core among different workers slows the entire system tremendously down.
Each sample had about 90 seconds to perform the infection. After rebooting, the systems had 65 seconds on XP respectively 110 seconds on Win7 to boot successfully. However, if the OS did not boot within this period the system was marked as broken.
The Windows XP worker used a disk layout with MBR partitioning and a single partition. In this case, the dark regions are located at the beginning and the end of the drive. The Win7 worker

Malware family	Sample count	Share per family
Backboot	1	0.0%
Careto	5	0.0%
Cidox	2,913	11.0%
CPD	40	0.1%
Finfish	29	0.1%
Fisp	255	1.0%
Geth	6	0.0%
GoodKit	1	0.0%
Infinaeon	1	0.0%
Korablin	30	0.1%
Lapka	106	0.4%
Mybios	51	0.2%
Nimnul	2,602	9.9%
Niwa	42	0.2%
Phanta	1012	3.8%
Pihar	452	1.7%
Plite	4,002	15.2%
Qvod	2,906	11.0%
Sinowal	3,014	11.4%
Smitnyl	302	1.1%
SST	75	0.3%
Stoned	26	0.1%
TDSS	3,025	11.5%
Trup	1,883	7.1%
Wistler	902	3.4%
Xpaj	622	2.4%
Yurn	46	0.2%
ZAccess	2,024	7.7%
Zhaba	5	0.0%
	26,378	100.0%

Table 7.2: Overview of the utilized malware families, their corresponding sample count and share per family; vertical lines are inserted for increased readability only

used a similar layout but with the additional Win7 hidden system partition. As a consequence, there is an extra dark region in-between the two partitions. [40]

Criterion	Satisfied	Comment
Correct Datasets		
Removed goodware	Y	performed extra experiment for goodware
Avoided overlays	Y	discussed in Chapter 10
Balanced families	Y	samples selected based on malware family label
Separated dataset	NA	no separated datasets utilized
Mitigated artifacts/biases	Y	see Section 9
Higher privileges	Y	techniques implemented on emulator level (QEMU)
Transparency		
Interpreted FPs	Y	non observed during eval; cf. Chapter 3 and 9
Interpreted FNs	Y	may be caused by evasion techniques; cf. Chapter 3 and 9
Interpreted TPs	Y	discussed in Chapter 8
Listed malware families	Y	outlined in Table 7.2
Identified environment	Y	see Section 7.2
Mentioned OS	Y	XP SP3 and Win7 (both 32 bit); cf. Section 7.2
Described naming	Y	according to [77]
Described sampling	Y	all samples of family or randomly chosen; cf. Section 7.1;
Listed malware	Y	Sept. 2014 - May 2015; cf. Section 7.1
Described NAT	Y	using NAT and rate limited; cf. Section 7.3
Mentioned trace duration	Y	90 seconds for infection, 2 minutes for reboot
Realism		
Removed moot samples	N	older samples were explicitly focus of evaluation
Real-world FP exp.	NA	no real-world experiments on physical user machines
Real-world TP exp.	NA	no real-world experiments on physical user machines
Used many families	Y	utilized 29 malware families; cf. Table 7.2
Allowed Internet	Y	experiments with and without Internet access; cf. 7.3
Added user interaction	Part.	short experiment time since focus on infection
Used multiple OSs	Y	Win XP SP3 and Win7 (both 32 bit); cf. Section 7.2
Safety		
Deployed containment	Y	discussed in Section 7.3

Table 7.3: Rossow's guideline checklist for malware experiments [76]

7.3 Containment Measures taken during the Malware Experiment

The analysis was performed with and without Internet connectivity for the analysis environment and the corresponding samples. The main experiment was conducted without Internet access, since gaining a historic view on bootkit evolution was one of the goals and Internet access would have allowed the samples to fetch new payloads instead of the embedded and historic one. As outlined in Table 7.4, there is not so much difference between denying and allowing network connectivity utilizing our sample set. During the experiments with active Internet connection, the samples were limited in network bandwidth and behind a NAT. Table 7.4 outlines the evaluation outcome for XP when denying or allowing the samples' Internet connectivity.

	XP without Internet	XP with Internet	Difference (in %)
BK like behaviour	2,464	2,945	16.3%
Bootkit detected	2,103	2,131	1.3%
Working infections	1,172	1,187	1.3%
Succ. infection rate	47.6%	40.3%	7.3%

Table 7.4: Comparing analysis results without and with Internet connectivity [40]

The term "***bootkit like behaviour***" is defined as samples writing to any DR, "***bootkit detected***" are samples modifying the MBR, VBR or bootloader during infection phase (so *bootkit detected* implies *bootkit like behaviour* but does not imply the inference), "***working infections***" are bootkits satisfying *bootkit detected* and the OS successfully reboots after the sample's execution, and "***successful infection rate***" is the share between the number of samples with *bootkit like behaviour* and *working infections*.
On Windows XP, granting the samples to access the Internet introduces a 16.3% increase of samples with *bootkit like behaviour* (writing somewhere outside of partitions). But both, the working infections and the detected bootkits raise only by roughly 1%. Table 7.4 indicates, there is hardly any difference in the amount of *working infections* allowing or prohibiting Internet connectivity. As previously discussed, the rest of the evaluation focuses on the results without Internet access as network connectivity would distort the results on the historic evolution of bootkit technology. [40]

CHAPTER 8

Results

This chapters presents the results for the conducted large-scale bootkit malware analysis utilizing Bootcamp. Based on the large-scale analysis, it illustrates a historic perspective on the evolution of bootkit technology and shows the performance of the proposed detection and prevention techniques.

This section is an extended version of my paper presented at DIMVA 2015 [40]. However, this work contains more details and further insights gained by analysing 2.564 additional samples, performed between February and May 2015.

8.1 Large-Scale Bootkit Analysis

Out of 26,378 examined samples, 2,464 binaries (on XP) respectively 2,206 samples (on Win7) indicated *BK like behaviour*. Table 8.1 provides an overview on the results of the samples with *BK like behaviour* grouped by the operating system. *XP && Win7* are the amount of binaries working on XP and Win7. *XP || Win7* defines the number of samples working on at least one OS. The other terms such as *BK like behaviour*, *bootkits detected*, *working infections*, and *succ. infection rate* are defined as in Section 7.3.

Overall, 2,501 binaries out of 26,378 malware samples uncovered *bootkit like behaviour* on at least one operating system. Altogether, more bootkits or samples with *bootkit like behaviour* are detected on Windows XP than Windows 7 (2,103 / 2,464 on XP compared to 1,852 / 2,206 on Win7). On the other, hand we detected more working bootkit infections on Win7 than on XP (1,694 compared to 1,172). Roughly 44% of the bootkits are operating successfully on both OSs. 258 bootkits can only install themselves on Windows XP, while 15 samples work on Windows 7 exclusively. Therefore, nearly all samples working on Win7 operate on Windows XP, too. This observation does not hold for the other direction.

Table 8.2 provides a survey on the results for each malware family utilized in the evaluation. It includes the number of samples analysed per family (*sample count*), the *first and last appearance* of a sample within the family, the number of *working infections* (i.e. machines which

	XP	Win7	XP && Win7	XP || Win7
BK like behaviour	2,464	2,206	2,192	2,501
Bootkit detected	2,103	1,852	1,831	2,104
Working infections	1,172	1,694	954	1,908
System boot fails	1,292	512	1,238	593
Succ. infection rate (in %)	47.6	76.8	43.5	76.3

Table 8.1: Overview on samples with bootkit like behaviour on different OSs

rebooted successfully after bootkit infection) and *succ. infection rate* (i.e. the share of successful infections compared to all analysed samples). The survey is separated for XP and Win7.

The *succ. infection rate* depends on the OS and family. While some malware families exclusively target Win7, such as the infamous TDSS family or Pihar, Lapka and MyBios target Win XP only. However, the majority target both operating systems, e.g. Sinowal and Cidox. Some malware families have quite high *succ. infection rates*, e.g. Pihar and TDSS for Win7 respectively Sinowal and Stoned for Windows XP. Certain families feature a very high number of samples (e.g. Plite, Qvod and Nimnul), nevertheless resulting in a very low *succ. infection rate*. This fact might be caused by several reasons. Our system might not provide the proper environment for the bootkit's installation (e.g. service pack, installed software). Those malware families could employ cloaking techniques as outlined in Chapter 9. On the other hand, the bootkits may focus on certain environments only, for example targeting users of a specific language exclusively.

Typically, bootkits have to utilize further disk space as the initial infection vector is too small for their entire data and code and hence, they have to persist it somewhere else. Table 8.3 provides an overview on the dark region utilization used by the malware samples. *End of disk* refers to disk utilization beyond the last partition, while *between partitions* indicates exploitation of any space between partitions, e.g. the space between MBR and VBR. *Both* refers to bootkits utilizing both locations to store data, while *just inside partitions* describes samples using just disk space inside partitions to store their data. Finally, *writes to DR* describes samples writing to any dark region. The hard disk sectors responsible for the boot process (MBR, VBR, BL) were excluded in this table as every bootkit has to write there anyway. Table 8.3 outlines the data and code storage locations for bootkits.

The majority (80.8% on average) utilizes the *end of the disk* as persistent data storage location. Only a few samples split the data storage and use both locations. The rest (18.4% on average) uses the space between partitions. We observed a trend moving away from utilizing the dark region at the end of the disk towards exploiting the space between partitions. The utilized dark region layout for Win7 is presented in Figure 2.5. The dark region layout for XP is very similar, with the absence of a hidden OS partition at the disk's end and therefore dark region 5 on XP is equivalent to dark region 6 on Win7. During our analysis, all samples wrote to dark region 2 or 5 (beyond the last partition) on XP or to dark region 2, 5 or 6 on Win7. We exploited

				XP		Win7	
Malware family	First appearance	Last appearance	Sample count	Working infections	Succ. infection rate	Working infections	Succ. infection rate
Yurn	2006-05	2013-06	45	1	2.2%	1	2.2%
SST	2006-05	2012-10	64	0	0.0%	0	0.0%
Sinowal	2006-05	2014-12	3,023	689	22.8%	601	19.9%
Plite	2006-10	2014-12	4,024	1	0.0%	1	0.0%
Infinaeon	2007-10	2007-10	1	0	0.0%	0	0.0%
Trup	2007-12	2014-12	2,078	76	3.7%	52	2.5%
TDSS	2008-07	2014-12	3,025	0	0.0%	505	16.7%
Zhaba	2008-10	2010-05	5	0	0.0%	0	0.0%
Qvod	2009-03	2014-12	2,901	0	0.0%	0	0.0%
Stoned	2009-07	2014-02	23	4	17.4%	2	8.7%
Smitnyl	2009-08	2014-03	303	58	19.1%	68	22.4%
Xpaj	2009-10	2014-01	565	17	3.0%	18	3.2%
Niwa	2009-10	2014-12	39	8	20.5%	1	2.6%
Phanta	2010-03	2014-05	980	81	8.3%	29	3.0%
Wistler	2010-05	2014-12	814	1	0.1%	0	0.0%
Nimnul	2010-07	2014-12	2,578	0	0.0%	2	0.1%
Finfish	2010-10	2014-06	30	1	3.3%	0	0.0%
Fisp	2011-03	2014-04	251	0	0.0%	0	0.0%
ZAccess	2011-05	2014-12	2038	0	0.0%	0	0.0%
Lapka	2011-06	2014-12	102	17	16.7%	0	0.0%
Cidox	2011-07	2014-12	2,902	201	6.9%	201	6.9%
Mybios	2011-07	2014-03	48	13	27.1%	0	0.0%
Pihar	2011-08	2013-03	459	0	0.0%	211	46.0%
GoodKit	2011-11	2011-11	1	0	0.0%	0	0.0%
CPD	2012-05	2014-12	39	3	7.7%	1	2.6%
Geth	2012-06	2012-10	6	0	0.0%	0	0.0%
Korablin	2012-08	2014-12	28	0	0.0%	0	0.0%
Backboot	2013-03	2013-03	1	1	100.0%	1	100.0%
Careto	2014-02	2014-02	5	0	0.0%	0	0.0%
			26,378	1,172	4.4%	1,694	6.4%

Table 8.2: Survey on results separated by malware family

this highly interesting observation for the detection and prevention techniques.

Table 8.4 provides an overview on the exploited infection vectors to obtain initial control during the startup phase. The Master Boot Record (MBR) is, with 82.7% exploitation on average, the dominating initial infection vector, succeeded by the bootloader (BL) utilized by 17.2%

	XP		Win7		
Data storage location	Count	Share	Count	Share	Avg on both
End of disk	979	83.5%	1,338	79.0%	80.8%
Between partitions	170	15.0%	352	20.8%	18.4%
Both	16	1.5%	4	0.2%	0.7%
Writes to DR	**1,172**	**100.0%**	**1,694**	**100.0%**	**100.0%**
	1,172	100.0%	1,694	100.0%	100.0%

Table 8.3: Dark regions utilized as data storage (for successful infections only)

	XP		Win7		
Infection vector	Count	Share	Count	Share	Avg on both
MBR	969	82.7%	1,492	88.1%	85.9%
VBR	2	0.2%	1	0.1%	0.1%
BL	201	17.2%	201	11.9%	14.0%
	1,172	100.0%	1,694	100.0%	100.0%

Table 8.4: Overview on exploited infection vectors (for successful infections only)

samples averagely. In the contrary, the Volume Boot Record (VBR) is exploited by hardly any bootkit as infection vector. There is a severe trend on the used infection vectors, moving away from the MBR and shifting towards BL exploitation. Further details on this shift are provided in Section 8.2.

	XP		Win7		
Hooked interrupt	Count	Share	Count	Share	Avg on both
0x13	975	80.7%	1,506	88.1%	85.0%
0x15	211	17.5%	202	11.8%	14.2%
0x83 & 0x85	8	0.7%	1	0.1%	0.3%
None	14	1.2%	1	0.1%	0.5%
	1,208	100.0%	1,710	100.0%	100.0%

Table 8.5: Outline of utilized targets for interrupt hooking (for successful infections only)

The exploited interrupts for hooking are endorsed in Table 8.5. As described in Section 2.2 interrupt hooking is used by bootkits to regain control of the system after executing benign code e.g. the original MBR. Some samples hook multiple interrupts, hence the sum increased from 1,172 on XP and 1,694 on Win7 to 1,208 (XP) and 1,710 (Win7), respectively. The dominating hooking target is interrupt `0x13` used by 85.0% on average followed by `0x15` exploited by 14.2% on average (for both operating systems). As discussed in Section 2.2, interrupt `0x13` is normally used to load hard disk content into memory. Hence, it is highly convenient to determine whether the BL loads the kernel into memory. As bootkits typically patch or inject code into the kernel, interrupt `0x13` is the most obvious option to observe whether the kernel is loaded. On the other hand, utilizing interrupt `0x13` results in a vast number of invocations during the boot process, as this interrupt is normally raised several thousand times. Interrupt `0x15` is called typically only once during the *bootkit execution phase* and executed before passing control to the kernel. This fact makes it an excellent interrupt hooking target for bootkits.

Interrupt `0x83` and `0x85` were utilized by only 8 samples (*Niwa*) on XP and 1 on Win7, in the time frame between 2009 and 2012. Those two interrupts are just re-directions to the original `int 0x13` respectively `int 0x15` interrupt service routines. 1 *Plite* sample on both operating systems and 13 *Mybios* samples on XP do not perform interrupt hooking at all. Those malware families employ a slightly different infection technique. Instead of in-memory kernel patching, those samples place a malicious *explorer.exe* on the system's disk [13]. But clearly this exploitation technique leaves infection traces on the file system. We observed this behaviour starting in 2011 until 2014. This technique is discussed in more detail in Section 8.2.

8.2 Evolution of Bootkits

This section discusses the evolution of bootkit technology. The evolution on successful bootkit infections rates is outlined in Table 8.6.

	XP			Win7		
Year	BK like behaviour	Working infections	Succ. infections	BK like behaviour	Working infections	Succ. infections
2008	7	7	100.0%	0	0	-
2009	473	461	97.5%	465	391	84.1%
2010	430	194	45.1%	388	222	57.2%
2011	932	131	14.1%	815	593	72.8%
2012	326	131	40.2%	283	246	86.9%
2013	50	38	76.0%	38	28	73.7%
2014	246	210	85.4%	217	214	98.6%
	2,464	1,172		2,206	1,647	

Table 8.6: Evolution on successful infection rates

While the successful infection rate started quite high in 2008 and 2009 (between 100.0% and 84.1%) it decreased tremendously within the next years and hit the absolute low-point with a rate of 14.1% (XP) and 57.2% (Win7). The successful infection rate started to raise again in 2012 in order to peak 2 years later in a local maximum, resulting in rates of 85.4% on XP and 98.6%

on Win7. As Windows 7 was released at the end 2009, there was not working bootkit infection monitored before that year. The *successful infection rate* for XP was quite low between 2010 to 2012, which might point out a strong focus on Win7 and Win8 (which has not been evaluated in this work) prior to maintain operability on the older XP. Though we can not explain the high infection rates in the early days the remaining trend indicates an obvious improvement and increase in professionalism in the underground malware industry for bootkits.

Figure 8.1 provides an overview on the evolution of exploited initial infection vectors.

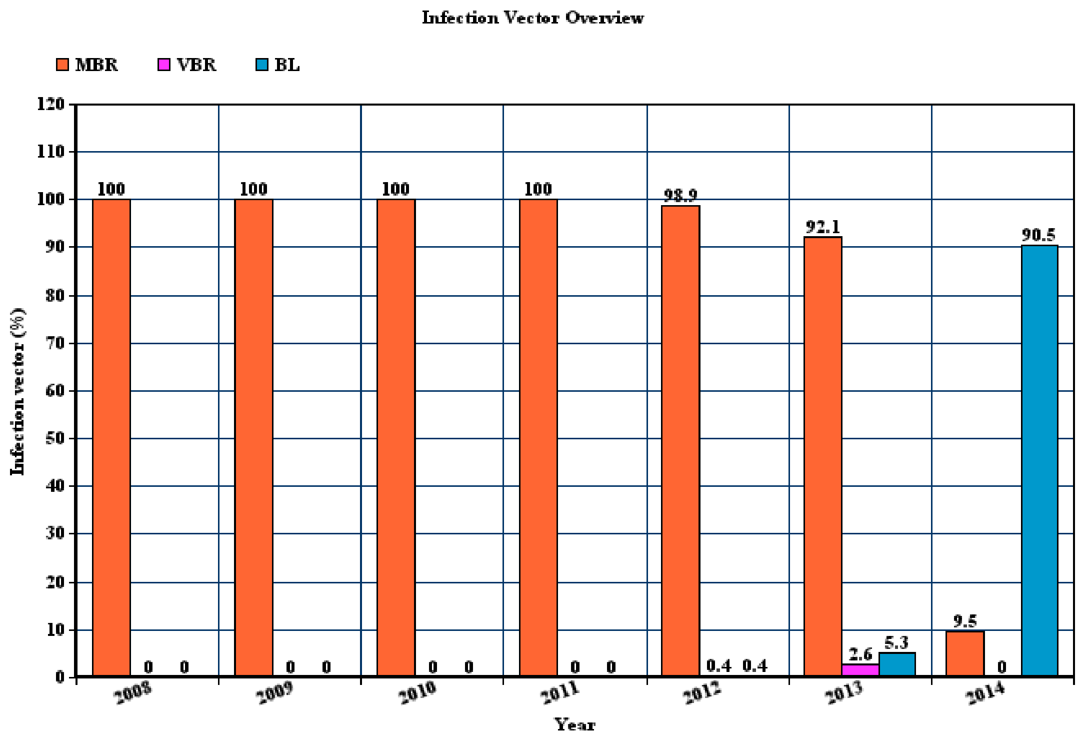


Figure 8.1: Outline on the evolution of infection vectors

The MBR was the only infection vector utilized between 2008 and 2011. Some samples exploited the VBR as infection vector (0.4% / 2.6%) in 2012. Starting with the following year, the BL infections became more and more popular (0.4% / 5.3%). More than 90% of the BKs utilized the BL in 2014, while only less than 10% exploited the MBR. In 2013, the source code of the *Carberp* botnet was leaked which is likely the explanation on the explosion of BL infections [52]. Additionally, the leak contained the source code of the *Cidox* bootkit which is utilized by *Carberp* [77]. Regrettably, the *Cidox* bootkit has become the standard template for bootkits, similar to *Zeus* after its infamous leak in 2011 [34]. After the *Zeus* leak in 2011, a massive amount of mutations spread across the Internet resulting in several highly sophisticated, and still state-of-the-art, banking trojans such as SpyEye [79], Ice IX [32], Citadel [20] or Zeus

P2P variants like ZeroAccess and Kelihos [74]. Similar things are happening with *Cidox*. The evolution peaked in *Zberp* - a mutation between *Zeus* and *Carberp* especially applying the *Cidox* bootkit component to *Zeus* which was discovered in June 2014 [11].

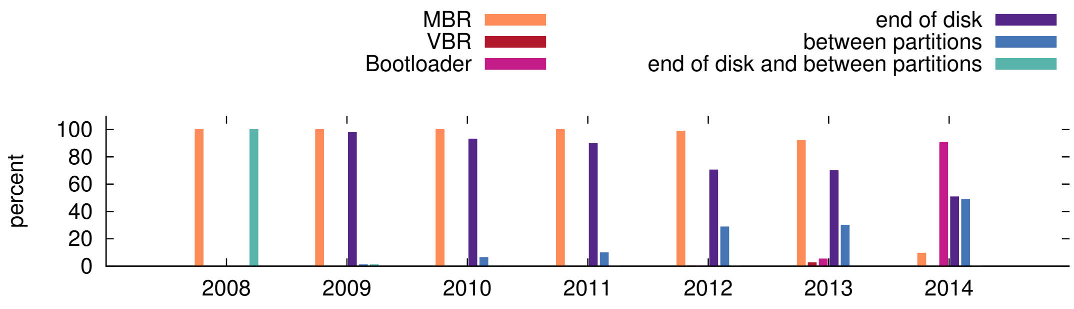


Figure 8.2: Outline on the evolution of dark region exploitation [40]

The utilization of dark regions (DRs) throughout the years is outlined in Figure 8.2. In 2008, bootkits mostly used the MBR and end of disk regions to store their code / data. Initially, bootkits focused on utilizing the space beyond the last partition. However, starting with 2010, more and more bootkits shifted towards utilizing the space between partitions. Which resulted in similar DR exploitation behaviour between space at the end of the disk and between partitions in 2014.

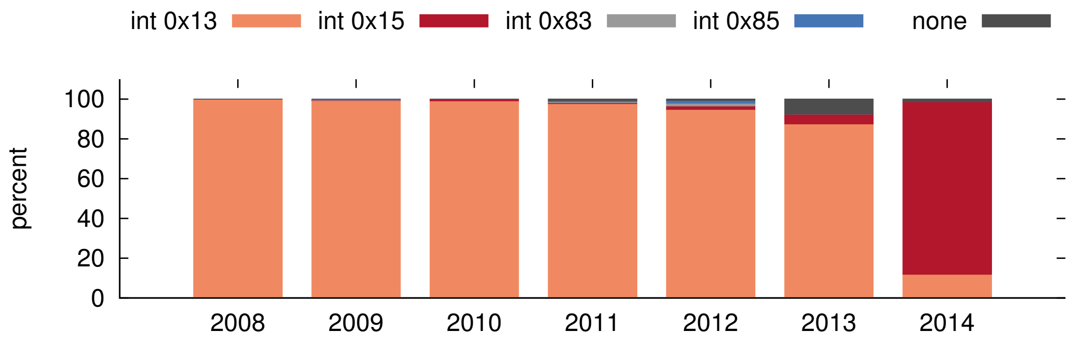


Figure 8.3: Historical transformation of interrupt hooking [40]

Figure 8.3 provides an overview on the historical transformation of interrupt hooking. The first observed working samples exploit interrupt `0x13` in 2008, while in 2009 hooking the interrupts `0x13`, `0x15`, `0x83` and `0x85` has been started by *Niwa* . The interrupts `0x83` and `0x85` are a redirection to the interrupt service routines for `int 0x13` and `int 0x15`, and hence calling them is just an obfuscation technique. During the experiment, the *Niwa* family were the only samples exploiting interrupt `0x83` and `0x85`. After 2012, no *Niwa* sample was detected anymore. In general, our analysis shows a clear shift towards exploiting interrupt `int 0x15` instead of int `int 0x13`. The previously discussed *Carberp* and *Cidox* source code leak [52] might be a very good reason for this observation, as this bootkit utilizes interrupt `0x15`. Hence, interrupt `0x15` will keep the dominant utilized hook.

In 2011, the *Mybios* family introduced an interesting attack strategy. They did not use interrupt hooking to regain control after the kernel resides in memory to patch it. Furthermore, *Mybios* did not perform any in-memory kernel patching either, but instead they replaced the original *explorer.exe* with a malicious one on disk during the startup phase. In 2014, this strategy was also utilized by *Plite*. Although this is a very interesting technique, it did not establish itself in the underground bootkit economy. This might be caused by the fact that this technique leaves potential suspicious traces within the file system.

The detailed results for the evolution (corresponding to Figures 8.1, 8.2 and 8.3) are outlined in the Tables 8.7, 8.8, 8.9.

Year	MBR	VBR	BL
2008	100.0%	0.0%	0.0%
2009	100.0%	0.0%	0.0%
2010	100.0%	0.0%	0.0%
2011	100.0%	0.0%	0.0%
2012	98.9%	0.4%	0.4%
2013	92.1%	2.6%	5.3%
2014	9.5%	0.0%	90.5%

Table 8.7: Overview on detailed historic evolution of the initial infection vectors

Year	End of disk	Between partitions	Both
2008	0.0%	0.0%	100.0%
2009	97.8%	1.1%	1.1%
2010	93.1%	6.4%	0.4%
2011	89.9%	9.9%	0.2%
2012	70.5%	28.8%	0.3%
2013	70.0%	30.0%	0.0%
2014	50.8%	49.0%	0.3%

Table 8.8: Historic perspective on the data storage utilization

8.3 Performance of Proposed Bootkit Detection Techniques

This subsection discusses the results for the bootkit detection techniques. The performance of those is presented in Table 8.10. As a short reminder, the *dark region read* heuristic triggers

Year	0x13	0x15	0x83	0x85	None
2008	100.0%	0.0%	0.0%	0.0%	0.0%
2009	99.3%	0.2%	0.2%	0.2%	0.0%
2010	99.1%	0.9%	0.0%	0.0%	0.0%
2011	97.7%	0.5%	0.5%	0.5%	0.9%
2012	94.8%	1.7%	1.4%	1.4%	0.7%
2013	87.5%	5.0%	0.0%	0.0%	7.5%
2014	11.9%	86.8%	0.0%	0.0%	1.4%

Table 8.9: Evolution of interrupt hook exploitation

whenever a disk sector outside a partition (except the MBR as this one has to be read during a benign boot process, too) is read. This detection heuristics caught all 1,175 respectively 1,694 bootkit samples on both operating systems. This behaviour also correlates with the observation highlighted in Table 8.3, indicating that every bootkit utilizes some space in dark regions. The *interrupt hook* technique performed also very well as it triggered for 99.8% of the samples on Win7 (1,691) and 98.7% on XP (1,157). This heuristic didn't catch samples which do not perform interrupt hooking but infected files inside the file system instead (cf. *Niwa* malware family description in Section 8.2). The *reuse existing code* technique caught 978 samples on XP (83.4%) and 1,501 on Win7 (88.6%), resulting in an overall average success rate of 86.0%. This technique missed the samples infecting the BL stage as those do not execute the malicious MBR / VBR followed by the benign one and therefore executing the address `0000h:7C00h` just twice (like benign boot processes also do), instead of three times. In addition, 100 benign samples (hashing programs, mathematical calculators and Windows utility programs) were analysed. Furthermore, the system was monitored while one hour of typical office usage (web browsing, text processing,...) for both OSs in order to evaluate the detection heuristics during normal operation. During those experiments, no false-positives were monitored, also owed to the fact that dark region modifications are a highly unlikely event for benign samples. Only some programs may write outside partitions, for example tools to correct errors within the MBR / VBR / BL, e.g. *fixmbr.exe* or re-partitioning a file system with *gparted*. [40]

Detection heuristic	XP	XP (%)	Win7	Win7 (%)	Avg
Dark region reads	1,172	**100.0%**	1,694	**100.0%**	**100.0%**
Interrupt hooks	1,157	**98.7%**	1,691	**99.8%**	**99.3%**
Executing address 0000h:7C00h	978	83.4%	1,501	88.6%	86.0%
Dark region writes (during bootkit infection phase)	1,175	**100.0%**	1,694	**100.0%**	**100.0%**

Table 8.10: Overview on performance of proposed detection techniques

8.4 Performance for the Bootkit Prevention Mechanism

Finally, Bootcamp's preventive technique, as described in Section 4, were applied and evaluated. For this evaluation, all 2,501 samples having *BK like behaviour* (writes outside partitions) during analysis on either XP or Win7 were utilized. The BK infection prevention technique was activated and the whole sample set was re-scanned. While *bootkit infection phase* the prevention system prohibited samples from modifying any space outside partitions and hence interrupting the bootkit's initial infection and storing their data. After reboot, the OS entered the *bootkit execution phase*. During this phase the system monitored the OS's boot behaviour by applying the detection heuristics in order to determine whether a successful infection occurred despite the prevention measures being active.

In 3.0% of the cases on XP and 7.2% on Win7 the system did not boot after executing the malware anymore. This might be caused by destroying essential parts located inside the OS's file system by the malware. For the successful booting systems, the detection heuristics were applied during the operating system's boot phase. Thereby no signs of infection were detected anymore, i.e. for all successful booting systems (on both OSs) none of the detection heuristics triggered. Hence, according to the proposed detection heuristics, the preventive mechanism stopped all 2,501 bootkit infection attempts on XP and Win7. [40]

CHAPTER 9
Discussion

Naturally, the presented approach comes with some general limitations and evasion techniques for particular components which are presented in this chapter.

9.1 General Limitations

As already discussed, the presented approach does not fully support UEFI (Unified Extensible Firmware Interface). It is applicable for systems utilizing UEFI for the bootkit infection phase, using the heuristic for dark region tracking. In general, UEFI severely differs from BIOS/MBR based startup process. Hence, the bootkit execution phase heuristics may not be effective and this scenario was not evaluated. As discussed in Chapter 7, the presented approach was only evaluated with MBR/VBR and BL-based bootkits, which are, however, the most widely spread bootkit types in the wild. Bootkits could try to evade the presented system by attacking the BIOS and trying to flash a malicious one into the system and thereby not attacking any boot process relevant sectors.

The current implementation does not support GPT (GUID Partition Table) yet, as discussed in Section 2.3. However, the concept of GPT is very similar to MBR partition tables, hence it can be implemented but takes additional effort and is therefore considered as future work.

The malware could also refrain from attacking disk sectors relevant for the startup process and thereby only stay inside of the file system. For example, malware could replace relevant system components by malicious ones directly on the file system, as discussed in Section 8.2. However, exploiting this approach leaves suspicious tracks inside the file system.

9.2 Evasion Techniques for Dynamic Analysis

Dynamic analysis suffers from several major limitations inherent to all dynamic analysis systems. For example, malware could utilize approaches to detect system emulators, e.g. QEMU which is used in Bootcamp, and could decline malicious behaviour [70].

Malware could exploit vulnerabilities in QEMU to gain crucial information or even take over the host system [10]. Attackers could also try to fingerprint the analysis environment, for example MAC / IP addresses, deployed hard-/software combinations, user names, or Windows serial keys. So far, Bootcamp does not implement any cloaking techniques.

Furthermore, malware can execute stalling code and wait before calling malicious functionality. Kolbitsch et al. [51] introduced techniques to detect stalling code and mitigation techniques, however it can not handle all types of this awkward behaviour. For example, some encrypted malware samples do not contain decryption keys but brute force themselves before execution. So far, it is an open research problem to reliably detect those kinds of techniques having limited resource and time constraints. Our implementation does not include any techniques to circumvent or detect those kinds of functionality and hence such malware can trick the presented approach.

9.3 Misdetection and Evading the Detection Heuristics

The following discusses the limitations of the presented detection heuristics.

Dark region modification heuristic

False positives occur in case of legitimate DR updates, for instance during a complete system re-installation, a major system upgrade or partitioning. Modern, sophisticated bootkits need more space for their code. Therefore, they utilize further and bigger DRs. For example, they use the inter-partition gaps and the gap at the end of the disk to store additional content, e.g. malware config, DLLs to inject. These dark regions can also be utilized by other kind of malware running within the target system and using it as hidden storage. This might trigger a false positive for this heuristic (although the false positive would still point to a real infection), as this behaviour also triggers the dark region modification heuristic.

Dark region read heuristic

False positives can for example be caused by bugs in the OS file system driver reading beyond the limits of the file system or by a forensic application inspecting parts of the drive outside the file system. In our solution both, the dark region modification and dark region read access technique have been implemented in QEMU's disk access system.

Bootkits could also store their data in unsuspicious hard disk areas inside a partition. However, this would inducing increased risk, as the BK could get accidentally overwritten by the OS, as is is not aware of the disk sectors utilized by the bootkit.

Interrupt hook heuristic

The interrupt hooking technique does not detect interrupt detouring attacks (cf. Section 2.2). An interrupt detouring attack does not modify the ISR's address in the IVT, but the ISR's code itself, e.g. it overwrites the first instruction of the ISR with an unconditional jump to the malicious one.

To detect interrupt detouring attacks, the same approach as for interrupt hooking detection can be applied, i.e. dumping either the whole memory area of the ISR code or just dumping the first e.g. 20 bytes of every ISR. Dumping the first couple of bytes for every ISR is sufficient, since detours are typically installed within the first few instructions, as developing malicious ISR functions which are executed in the middle of another ISR are highly complex. Finally, the system could periodically check whether the ISR code has been modified by comparing it to the original one. Currently, detouring attack detection is not implemented in Bootcamp, but could be done without problems investing additional effort.

Furthermore, bootkits may not exploit interrupt hooking at all, as discussed in Section 8.2. Those infections lack the capability to regain control after calling benign bootstrap code and therefore rely on a different attacker model. Hence, they rely on modifying files within the file system instead of injecting code in-memory / patching. However, replacing files inside the file system leaves suspicious traces.

Executing address 0000h:7C00h heuristic

False positives may be triggered by major operating system upgrades which change the way bootstrapping is done, e.g. by adding more functionality to the boot phase and thus using the same techniques as bootkits but for benign purposes. Additionally, the bootkit could patch the original MBR or VBR in memory or on disk to utilize a different base address than `0000h:7C00h`. Thereby, also infected systems would execute this specific address only twice.

Bootkits infecting the BL stage can circumvent this heuristic, as already discussed in Section 8.2 . Those samples do not execute a malicious MBR / VBR, followed by the benign one and therefore execute the address `0000h:7C00h` just twice. Moreover an attacker might patch the original MBR / VBR to be loaded and executed to a different memory address or refrain from reusing existing code.

9.4 Evasion Techniques for Attack Prevention on the OS Level

Raffetseder et al. [70] present several techniques in order to detect system emulators as utilized by the bootkit prevention technique on the OS level. Either the emulator is detected by the malware before it infects the target host, or a bootkit detects the emulator during the boot phase while being emulated. The following enumeration discusses techniques on how to achieve the evasion:

- **Engine / driver probing:** Before installing the bootkit, the sample could probe whether the proposed system components are present e.g. by detecting relevant files using a full disk search or by detecting the presence of the on-write-access driver.
- **Remove on-write-access driver:** Bootkits have to infect the start-up relevant disk sectors in order to interfere with the boot process. Assuming sufficient permissions, the malware could try to remove the on-write-access driver in order to circumvent the triggering condition for the detection engine. Furthermore, the attacker has to reboot the system which might potentially alert the user.

Furthermore, the bootloader could try to detect whether it is running in an emulated environment. However, it is important to note that the system's capabilities during the startup phase are restricted, e.g. in the x86 real mode. Hence, this would significantly raise the bar for implementing the following techniques.

- **System probing:** The bootkit could detect the emulator's presence by probing the environment, for example using the system's CPU, motherboard, network card.
- **Stalling code:** The emulator has to restrict the amount of spent time and resources to detect an infection. Hence, bootkits could utilize stalling code or exceed the number of instructions executed in the emulator.
- **Evade detection heuristics:** The bootkit could try to circumvent the detection heuristics to stay beyond the radar, as discussed in Section 9.3.

9.5 Potential Future Bootkit Evolution

In the future, bootkits might utilize the following trends:

- **Shift towards BL as initial infection vector:** Following the results presented in Section 8.2, the bootloader will likely become the major initial infection vector in the future. This is due to the fact that BL infections offer more space and furthermore is also more stealthy compared to the MBR and VBR. This trend is furthermore amplified by the Carberp / Cidox bootkit source code leak in 2013 [52].
- **Diversification of DR exploitation:** Figure 8.2 shows a trend towards diversifying the utilization of DRs within the last years. This behaviour might expand into more stable sections within the file system (e.g. Windows alternate data streams, file metadata or slack space). Thus bootkits could create a hybrid (in and outside of file system) storage. This is exemplified by *Mybios* which avoids using interrupt hooking and instead replaces a core OS file located in the file system. However, utilizing those techniques introduces more complexity and increases the bootkits risk of being accidentally overwritten by the OS, as it is not aware of the disk areas utilized by the bootkit. In addition, the BK must be able to understand the file system, be aware of the location of every exploited file and its slack space. Furthermore, it would have to observer update, copy, delete and move operations on those files to keep the data consistent.
- **BIOS/UEFI based Bootkits:** In 2011, the first BIOS based bootkit was detected in the wild [39], but hardly spotted since then because they are highly complex to implement and have to consider BIOS vendor and version specific details for their attack to succeed. Though a few UEFI PoC bootkits exist, to the best of our knowledge none has been spotted in the wild ever. Nevertheless, this does not mean BIOS/UEFI bootkits will not become popular one day.

- **VM based Bootkits:** Future bootkits may be able to utilize advanced CPU features, such as recent virtualization instructions. Exploiting those features would enable them aiming towards VM based BKs, as e.g. SubVirt proposed [48]. Modern BKs already inject kernel-level drivers to prevent the hidden DRs from being detected by AVs or other detection systems hosted inside the infected system. VM based BKs would be the next consequent step, by moving the infected system into a separate VM and thereby leveraging the concealment to a new level, e.g. the bootkit would run in x86 level 0 (i.e. kernel-level), while the actual victim OS kernel is moved to ring 1 and thereby having by design less privileges than the malware. Such techniques would not rely on injecting code into the victim's system any more, hence leaving the attacked system un-tampered and making detection from inside the infected system impossible by design.

CHAPTER 10

Related Work

The related work section is divided into dynamic malware analysis, bootkits, large-scale malware analysis and preventing malware infections.

10.1 Dynamic Malware Analysis

There are several contributions dealing with automatic malware analysis like [89] and [67], their detection [50] or their distribution [62], [58], but only a few deal with bootkit techniques, such as [54]. Sandboxes are instrumented virtual execution environments which execute malware samples in a contained environment. Typical malware analysis sandboxes such as e.g. Anubis [27] or CWSandbox [89] do not deploy any bootkit detection technology. Idika et al. [45] show a survey on malware detection techniques, while [35] 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 [64]. VMI is not only used in security research but also in other research and industry areas [64].

Most dynamic malware analysis contributions focus on capturing user-mode Windows API as Anubis [27] or CWSandbox [89] do. Some approaches like K-Tracer [53], dAnubis [65] or [88] target kernel level infections, whereas Kirat et al. [49] and Lindorfer et al. [55] target cloaking malware trying to evade analysis systems.

Other papers address on analysing network traffic produced by malware as e.g. Sandnet [75] does. None of them target bootkits or analyse them in detail.

10.2 Bootkits

In [54], the authors provide an overview on offensive techniques used by bootkits and perform a classification on BKs according to the infected boot process stage.

Rodionov et al. [37] outline an overview on bootkit evolution and classify them based upon their infection vector (MBR / VBR / BL). This overview focuses on the first occurrence of certain characteristics, e.g. first MBR / BL infection, and is not build upon essential bootkit techniques as our evolutionary survey is. [38] and [54] provide insights into bootkit attack techniques such as BIOS-, MBR-, NTLDR- or other technology based bootkit infection vectors. Grill et al. [41] proposed bootkit detection heuristics, but their work considers interrupt `0x13` hooks only and monitors just loading content from the hard disk's end during boot phase. Both heuristics are bypassed by more and more recent bootkits as shown in Section 8.1. On the contrary, both newly proposed hook detection and dark region access approaches are more general and feature better detection characteristics. Furthermore, they performed just a minor proof of concept evaluation compared to the large scale evaluation. [42] utilized a jump counter to address `0000h:7C00h` during the system startup phase as a heuristic in order to detect suspicious behaviour but did not evaluate his approach.

Icelord, the first BIOS-based proof-of-concept bootkit, was published in 2007 [1]. Such bootkits try to flash a malicious BIOS into the system and gain control even before the MBR is executed. In 2009 and 2013, further work has been presented by Wojtczuk et al. [69] and Schlaikjer et al. [73] in this area. In 2011, such a BIOS based bootkit has been first detected in the wild [39], but since then such kinds are hardly spotted in the wild as they are very complex to implement. For example, they have to respect BIOS vendor and version specific details for their attack.

GrayFish, a highly sophisticated malware recently used by the infamous Equation group, was analysed by Kaspersky Labs [47]. It utilizes VBR bootkit infection in order to hijack the first kernel driver's loading. Afterward it loads further components from the registry.

However, none of them presented a historic, large-scale and detailed bootkit analysis or large-scale evaluation on their approaches, as this work does.

10.3 Large-Scale Malware Analysis

Large-scale malware analysis have been conducted by various papers for varying objectives so far. The majority target scanning and analysing large malware sample sets, such as [89], [71], [33], [24], or [68] do. Others focus on the issues of scaling large malware analysis respectively improving the efficiency of dynamic malware analysis, e.g. [25], [44], [26]. However, none of them target bootkit technology related problems.

10.4 Preventing Malware Infections

Several papers have been published in the area of infection prevention respectively recovery.

Zhu et al. proposed an approach to analyse hard disk images after malware attacks [90]. [41] outlined a system to prevent bootkit infections utilizing an anti-virus system which is based on components inside the system to protect. Therefore, malware may disable the bootkit protection prior performing the attack. This drawbacks does not apply to the infection prevention approach proposed in this work.

Bacs et al. [22] utilized low-level hard disk inspection in order to recover from intrusions which also includes bootkit attacks.

Riley et al. exploit shadow memory and light-weight virtual machine monitor (VMM) in order to stop unauthorized kernel level code execution. This approach could stop kernel level code injected by bootkits, but not their attacks in general [72].

Weiqing et al. proposed an isolated and secure process execution environment for Linux. Hard disk I/O is redirected through a kernel module proxy which is creating a shadow drive to store the disk I/O separated by process. Later, the user can commit the shadowed data to the real disk. Though, this approach can prevent bootkit infections it targets experts and would be hard to utilize for less IT affine users [80].

CHAPTER 11

Future Work & Conclusion

This section outlines the future development directions and finally concludes the work.

11.1 Future Work

The following future improvements are planned:

- **Implement early boot phase network traffic detection:**
  One future development strategy is implementing the early boot phase network traffic detection, as described in Section 3.5. Implementing this heuristic takes a rather high effort. For example, to get a serious network traffic anomaly detection heuristics, one would have to apply machine learning (ML) to a large amount of benign and malicious network traffic to train a ML model.
- **Implement loop decryption detection heuristic:**
  As described in Section 3.6, implementing and fine-tuning the loop decryption detection is another future task. To improve the techniques' performance, one could search for self-modifying code in general by looking for memory areas which are read, modified and later on executed. To do so, taint analysis [66] could be applied. Another approach is to detect loops and entropy changes, i.e. loops and their corresponding buffers having high entropy (encrypted content) on loop entry and low entropy on loop exit (decrypted memory). This approach is also applied by Lutz et al. [56] to automatically decrypt network traffic. However, both approaches are rather complex to implement and would go beyond the scope of this work.
- **Implement interrupt detouring detection technique:**
  The interrupt hooking technique does not detect interrupt detouring attacks (cf. Section 2.2). An interrupt detouring attack does not modify the ISR's address in the IVT, but the ISR's code itself, e.g. it overwrites the first instruction of the ISR with an unconditional

jump to the malicious one. To detect interrupt detouring attacks, the same approach as for interrupt hooking detection can be applied, i.e. dumping either the whole memory area of the ISR code or just dumping the first e.g. 20 bytes of every ISR. Dumping the first couple of bytes for every ISR is sufficient, since detours are typically installed within the first few instructions, as developing malicious ISR functions which are executed in the middle of another ISR are highly complex. Finally, the system could periodically check whether the ISR code has been modified by comparing it to the original one. Currently, detouring attack detection is not implemented in Bootcamp, but could be done without problems investing additional effort.

- **GPT support:**
  So far GPT (GUID Partition Table) are not supported, as discussed in Section 2.3. However, the concept of GPT is very similar to MBR partition tables, hence the additional implementation effort is fungible.

- **Dumping malicious injected ISR code:**
  The current implementation does not dump the malicious ISR, as function boundary recognition is highly challenging and still a partially open research topic, as shown by Bao et al. [23]. However, it would be still possible to dump the first several hundred bytes of every malicious ISR, which would still result in providing highly relevant information, e.g. for clustering based on the injected ISR.

- **Dumping the modified dark regions:**
  At the moment, the modified dark regions are not extracted, although this feature would be very helpful. The modified DRs would not only contain the code for the malicious infection vector (MBR, VBR, BL) but also the (perhaps even un-encrypted) hidden data storage. All relevant information are available anyway, as all the writes to dark regions are logged. Therefore, this feature could be implemented with arguable effort and time. This information would be very helpful for both a human analyst but also for automated clustering of the bootkits.

11.2 Conclusion

This work presents a large-scale bootkit analysis and proposes detection and prevention techniques based on this analysis.

The large scale malware analysis is based on a dataset composed of more than 26k different malware samples on XP and Win7, spanning a timer period over 8 years. The presented results outline the evolution of bootkit technology since 2006 and show a major shift from exploiting the MBR as initial infection vector to utilizing the bootloader instead. Furthermore, modern bootkits are hiding their presence by exploiting the space between partitions, instead of only utilizing the area beyond the last partition.

The presented detection approach relies on behaviour anomaly identification during the system's boot process. Therefore, hard disk write attempts during the infection phase and disk accesses to dark regions, interrupt hooks and the execution count for the memory address `0000h:`

`7C00h` during the boot process are monitored in order to determine whether a bootkit infection occurred. The two proposed prevention approaches stop the samples by (i) preventing them from writing outside of partitions and therefore hindering the bootkit infecting the system and storing its data there or (ii) by observing dark region disk modifications and emulating their impact on the system's boot process.

The works shows that the proposed heuristics detected all bootkit infections. In addition, the infection prevention techniques was evaluated by executing malware samples with applied protection system. Afterwards we evaluated whether one of the proposed detection techniques identified a bootkit attack. As a result, it was shown that the prevention solution was successful stopping all bootkit infections.

Bibliography

[1] BIOS Rootkit: Welcome home, my Lord! http://blog.csdn.net/icelord/article/details/1604884, 2007, last accessed: 2016/08/30.

[2] Win2k Master Boot Record (MBR) revealed! http://thestarman.narod.ru/asm/mbr/Win2kmbr.htm, 2010, last accessed: 2016/08/30.

[3] Backdoor.pihar, 2011, last accessed: 2016/08/30. http://www.symantec.com/security_response/writeup.jsp?docid=2011-120817-1417-99&tabid=2.

[4] Advanced Evasion Techniques by Win32/Gapz. http://www.welivesecurity.com/wp-content/uploads/2013/05/CARO_2013.pdf, 2013, last accessed: 2016/08/30.

[5] krab.rar - leaked Carberp malware source code. https://mega.co.nz/#!0YsXWBRD!CMqd9nrm1d0XABKlifI9vmxprpQ6RnfsdhBHeKrDXao, 2013, last accessed: 2016/08/30.

[6] Finfisher, 2014, last accessed: 2016/08/30. http://www.finfisher.com/FinFisher/products_and_services.html.

[7] Finfisher malware dropper analysis, 2014, last accessed: 2016/08/30. https://www.codeandsec.com/FinFisher-Malware-Dropper-Analysis.

[8] GUID Partition Table. https://wiki.archlinux.org/index.php/GUID_Partition_Table, 2014, last accessed: 2016/08/30.

[9] Hacking team - remote control system, 2014, last accessed: 2016/08/30. http://www.hackingteam.it.

[10] Qemu vga emulator bug. http://www.securitytracker.com/id/1030817, 2014, last accessed: 2016/08/30.

[11] Zberp – the baby super-trojan. http://www.cyactive.com/zberp-baby-super-trojan/, 2014, last accessed: 2016/08/30.

[12] Bootkit Threats: In Depth Reverse Engineering & Defense. http://www.welivesecurity.com/wp-content/media_files/REcon2012.pdf, last accessed: 2016/08/30.

[13] Plite bootkit. http://labs.bitdefender.com/2012/05/plite-rootkit-spies-on-gamers/, last accessed: 2016/08/30.

[14] System Initialization (x86). http://wiki.osdev.org/System_Initialization_%28x86%29, last accessed: 2016/08/30.

[15] TDSS. TDL-4. https://securelist.com/analysis/publications/36563/xpaj-reversing-a-windows-x64-bootkit, last accessed: 2016/08/30.

[16] x86 protected mode. https://en.wikibooks.org/wiki/X86_Assembly/Protected_Mode, last accessed: 2016/08/30.

[17] x86 real mode. https://www.princeton.edu/~achaney/tmve/wiki100k/docs/Real_mode.html, last accessed: 2016/08/30.

[18] x86 real mode basics. http://www.rcollins.org/articles/pmbasics/tspec_a1_doc.html, last accessed: 2016/08/30.

[19] XPAJ: Reversing a Windows x64 Bootkit. https://securelist.com/analysis/publications/36563/xpaj-reversing-a-windows-x64-bootkit, last accessed: 2016/08/30.

[20] Sood Aditya and Bansal Rohit. Prosecting the citadel botnet – revealing the dominance of the zeus descendent. *Virus Bulletin*, 2014.

[21] Andrei Bacs, Cristiano Giuffrida, Bernhard Grill, and Herbert Bos. Slick: an intrusion detection system for virtualized storage devices. In *Proceedings of the 31st Annual ACM Symposium on Applied Computing*, 2016.

[22] Andrei Bacs, Remco Vermeulen, Asia Slowinska, and Herbert Bos. System-level support for intrusion recovery. In *Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, 2012.

[23] Tiffany Bao, Jonathan Burket, Maverick Woo, Rafael Turner, and David Brumley. Byteweight: Learning to recognize functions in binary code. In *23rd USENIX Security Symposium*, 2014.

[24] Ulrich Bayer, Imam Habibi, Davide Balzarotti, Engin Kirda, and Christopher Kruegel. A view on current malware behaviors. In *USENIX workshop on large-scale exploits and emergent threats (LEET)*, 2009.

[25] Ulrich Bayer, Engin Kirda, and Christopher Kruegel. Improving the efficiency of dynamic malware analysis. In *Proceedings of the 2010 ACM Symposium on Applied Computing*, 2010.

[26] Ulrich Bayer, Engin Kirda, and Christopher Kruegel. Improving the efficiency of dynamic malware analysis. In *Proceedings of the 2010 ACM Symposium on Applied Computing*, 2010.

[27] Ulrich Bayer, Christopher Kruegel, and Engin Kirda. Ttanalyze: A tool for analyzing malware. In *15th European Institute for Computer Antivirus Research (EICAR 2006) Annual Conference*, 2006.

[28] Fabrice Bellard. Qemu, a fast and portable dynamic translator. In *ATEC '05: Proceedings of the annual conference on USENIX Annual Technical Conference*, 2005.

[29] Hynek 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: 2016/08/30.

[30] Bill Blunden. *The Rootkit arsenal: Escape and evasion in the dark corners of the system.* Jones & Bartlett Publishers, 2012.

[31] Bochs IA-32 Emulator Project. Bochs. `http://bochs.sourceforge.net/`.

[32] Lucian Constantin. Citadel banking malware is evolving and spreading rapidly, researchers warn. `http://www.pcworld.com/article/249631/citadel_banking_malware_is_evolving_and_spreading_rapidly_researchers_warn.html`, 2012, last accessed: 2016/08/30.

[33] Marco Cova, Christopher Kruegel, and Giovanni Vigna. Detection and analysis of drive-by-download attacks and malicious javascript code. In *Proceedings of the 19th International Conference on World Wide Web*, 2010.

[34] Roland Dela Paz. ZeuS Source Code Leaked, Now What? `http://blog.trendmicro.com/trendlabs-security-intelligence/the-zeus-source-code-leaked-now-what/`, 2013, last accessed: 2016/08/30.

[35] Manuel Egele, Theodoor Scholte, Engin Kirda, and Christopher Kruegel. A survey on automated dynamic malware-analysis techniques and tools. *ACM Computing Surveys (CSUR)*, 2012.

[36] Paul England, John D DeTreville, and Butler W Lampson. Loading and identifying a digital rights management operating system, December 4 2001. US Patent 6,327,652.

[37] David Harley Eugene Rodionov, Aleksandr Matrosov. Bootkits: Past, Present and Future. In *Virus Bulletin Conference*, 2014.

[38] Hongbo Gao, Qingbao Li, Yu Zhu, Wei Wang, and Li Zhou. Research on the working mechanism of bootkit. In *8th International Conference on Information Science and Digital Content Technology (ICIDT), 2012*, 2012.

[39] Marco Giuliani. Mebromi: the first bios rootkit in the wild. http://www.webroot.com/blog/2011/09/13/mebromi-the-first-bios-rootkit-in-the-wild/, 2011, last accessed: 2016/08/30.

[40] Bernhard Grill, Andrei Bacs, Christian Platzer, and Herbert Bos. “nice boots!”- a large-scale analysis of bootkits and new ways to stop them. In *Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*. 2015.

[41] Bernhard Grill, Christian Platzer, and Jürgen Eckel. A practical approach for generic bootkit detection and prevention. In *Proceedings of the Seventh European Workshop on System Security*, 2014.

[42] Lars Haukli. Exposing bootkits with bios emulation. *Black Hat US*, 2014, last accessed: 2016/08/30. https://www.blackhat.com/docs/us-14/materials/us-14-Haukli-Exposing-Bootkits-With-BIOS-Emulation-WP.pdf.

[43] Joel Hruska. Linux’s worst-case scenario: Windows 10 makes secure boot mandatory, locks out other operating systems, 2015, last accessed: 2016/08/30. http://www.extremetech.com/extreme/201722.

[44] Xin Hu, Tzi-cker Chiueh, and Kang G. Shin. Large-scale malware indexing using function-call graphs. In *Proceedings of the 16th ACM Conference on Computer and Communications Security*, 2009.

[45] Nwokedi Idika and Aditya P Mathur. A survey of malware detection techniques. *Purdue University*, 2007.

[46] Intel. Intelligent Platform Management Interface. http://www.intel.com/content/www/us/en/servers/ipmi/ipmi-specifications.html.

[47] Kaspersky Lab. Equation group: Questions and answers, 2015. https://securelist.com/files/2015/02/Equation_group_questions_and_answers.pdf.

[48] Samuel T. King, Peter M. Chen, Yi-Min Wang, Chad Verbowski, Helen J. Wang, and Jacob R. Lorch. Subvirt: Implementing malware with virtual machines. In *IEEE S&P*, 2006.

[49] Dhilung Kirat, Giovanni Vigna, and Christopher Kruegel. Barecloud: bare-metal analysis-based evasive malware detection. In *23rd USENIX Security Symposium*, 2014.

[50] Clemens Kolbitsch, Paolo Milani Comparetti, Christopher Kruegel, Engin Kirda, Xiaoyong Zhou, and XiaoFeng Wang. Effective and efficient malware detection at the end host. In *USENIX Security Symposium*, 2009.

[51] Clemens Kolbitsch, Engin Kirda, and Christopher Kruegel. The power of procrastination: detection and mitigation of execution-stalling malicious code. In *Proceedings of the 18th ACM conference on computer and communications security (CCS)*, 2011.

[52] Brian Krebs. Carberp source code leak. `https://krebsonsecurity.com/tag/carberp-source-code-leak/`, 2013, last accessed: 2016/08/30.

[53] Andrea Lanzi, Monirul I Sharif, and Wenke Lee. K-tracer: A system for extracting kernel malware behavior. In *Network and Distributed System Security Symposium (NDSS)*, 2009.

[54] Xiang Li, Yan Wen, Minhuan Huang, and Qiang Liu. An overview of bootkit attacking approaches. In *7th International Conference on Mobile Ad-hoc and Sensor Networks (MSN)*, 2011.

[55] Martina Lindorfer, Clemens Kolbitsch, and Paolo Milani Comparetti. Detecting environment-sensitive malware. In *Recent Advances in Intrusion Detection (RAID)*, 2011.

[56] Noé Lutz. Towards revealing attacker's intent by automatically decrypting network traffic. *Mémoire de maıtrise, ETH Zürich, Switzerland*, 2008.

[57] Aleksandr Matrosov. ESET - Rovnix bootkit framework updated. `http://www.welivesecurity.com/2012/07/13/rovnix-bootkit-framework-updated`, 2012, last accessed: 2016/08/30.

[58] Niels Provos Panayiotis Mavrommatis and Moheeb Abu Monrose. All your iframes point to us. *USENIX Security Symposium*, 2008.

[59] Microsoft. Early launch antimalware, 2014, last accessed: 2016/08/30. `https://msdn.microsoft.com/en-us/library/windows/desktop/hh848061%28v=vs.85%29.aspx`.

[60] Microsoft. Kernel Patch Protection, 2014, last accessed: 2016/08/30. `http://msdn.microsoft.com/en-us/library/windows/hardware/Dn613955%28v=vs.85%29.aspx`.

[61] Tipp Moseley, Dirk Grunwald, Daniel A Connors, Ram Ramanujam, Vasanth Tovinkere, and Ramesh Peri. Loopprof: Dynamic techniques for loop detection and profiling. In *Proceedings of the 2006 Workshop on Binary Instrumentation and Applications (WBIA)*, 2006.

[62] Alexander Moshchuk, Tanya Bragin, Steven D Gribble, and Henry M Levy. A crawler-based study of spyware in the web. In *Network and Distributed System Security Symposium (NDSS)*, 2006.

[63] MSDN. Driver Signing. `http://msdn.microsoft.com/en-us/library/windows/hardware/ff544865(v=vs.85).aspx`.

[64] Kara Nance, Brian Hay, and Matt Bishop. Virtual machine introspection. *IEEE Computer Society*, 2008.

[65] Matthias Neugschwandtner, Christian Platzer, Paolo Milani Comparetti, and Ulrich Bayer. Danubis–dynamic device driver analysis based on virtual machine introspection. In *Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*. 2010.

[66] James Newsome and Dawn Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. *12th Network and Distributed Systems Security Symposium (NDSS)*, 2005.

[67] Michalis Polychronakis and Niels Provos. Ghost turns zombie: Exploring the life cycle of web-based malware. *USENIX workshop on large-scale exploits and emergent threats (LEET)*, 8, 2008.

[68] G. Portokalidis, A. Slowinska, and H. Bos. Argos: an emulator for fingerprinting zero-day attacks. In *ACM SIGOPS EuroSys '06*, 2006.

[69] Wojtczuk Rafal and Tereshkin Alexander. Attacking intel bios, 2009, last accessed: 2016/08/30. `https://www.blackhat.com/presentations/bh-usa-09/WOJTCZUK/BHUSA09-Wojtczuk-AtkIntelBios-SLIDES.pdf`.

[70] Thomas Raffetseder, Christopher Kruegel, and Engin Kirda. Detecting system emulators. In *International Conference on Information Security*. 2007.

[71] Konrad Rieck, Philipp Trinius, Carsten Willems, and Thorsten Holz. Automatic analysis of malware behavior using machine learning. *Journal of Computer Security*, 2011.

[72] Ryan Riley, Xuxian Jiang, and Dongyan Xu. Guest-transparent prevention of kernel rootkits with vmm-based memory shadowing. In *Recent Advances in Intrusion Detection (RAID)*, 2008.

[73] Schlaikjer Ross. Overview of bios rootkits, 2013, last accessed: 2016/08/30. `https://tuftsdev.github.io/DefenseOfTheDarkArts/students_works/final_project/rschlaikjer.pdf`.

[74] Christian Rossow, Dennis Andriesse, Tillmann Werner, Brett Stone-Gross, Daniel Plohmann, Christian J Dietrich, and Herbert Bos. Sok: P2pwned-modeling and evaluating the resilience of peer-to-peer botnets. In *Proceedings of the 34th IEEE Symposium on Security and Privacy (S&P)*, 2013.

[75] Christian Rossow, Christian J Dietrich, Herbert Bos, Lorenzo Cavallaro, Maarten Van Steen, Felix C Freiling, and Norbert Pohlmann. Sandnet: Network traffic analysis of malicious software. In *Proceedings of the First Workshop on Building Analysis Datasets and Gathering Experience Returns for Security*, 2011.

[76] Christian Rossow, Christian J. Dietrich, Christian Kreibich, Chris Grier, Vern Paxson, Norbert Pohlmann, Herbert Bos, and Maarten van Steen. Prudent Practices for Designing Malware Experiments: Status Quo and Outlook . In *Proceedings of the 33rd IEEE Symposium on Security and Privacy (S&P)* , 2012.

[77] Vyacheslav Rusakov and Sergey Golovanov. Attacks before system startup. `http://securelist.com/blog/research/63725/attacks-before-system-startup/`, 2014, last accessed: 2016/08/30.

[78] Mark Russinovich. Sysinternals suite, 2014, last accessed: 2016/08/30. `https://technet.microsoft.com/en-us/sysinternals/bb842062.aspx`.

[79] Bodmer Sean. First zeus, now spyeye. `https://www.damballa.com/first-zeus-now-spyeye-look-the-source-code-now/`, last accessed: 2016/08/30.

[80] Weiqing Sun, Zhenkai Liang, R. Sekar, and V. N. Venkatakrishnan. One-way isolation: An effective approach for realizing safe execution environments. In *Network and Distributed System Security Symposium (NDSS)*, 2005.

[81] Symantec. More information on Alureon, 2014, last accessed: 2016/08/30. `http://www.symantec.com/security_response/writeup.jsp?docid=2008-091809-0911-99&tabid=2`.

[82] Technet. Kernel patch protection for x64-based operating systems. `http://technet.microsoft.com/en-us/library/cc759759%28v=ws.10%29.aspx`.

[83] Alex Tereshkin and Joanna Rutkowska. Evil Maid Attack. `http://theinvisiblethings.blogspot.com/2009/10/evil-maid-goes-after-truecrypt.html`.

[84] tigzy. Carberp bootkit : How self-protection is effective, 2009, last accessed: 2016/08/30. `http://www.adlice.com/carberp-bootkit-how-self-protection-is-effective`.

[85] AVR Tutorial. Basic understanding of microcontroller interrupts. `http://www.embedds.com/basic-understanding-of-microcontroller-interrupts`, last accessed: 2016/08/30.

[86] Rusakov Vyacheslav. Mybios. is bios infection a reality? `https://securelist.com/analysis/publications/36421/mybios-is-bios-infection-a-reality/`, 2011, last accessed: 2016/08/30.

[87] Steve R. White, Jeffrey O. Kephart, and David M. Chess. Computer viruses: A global perspective. In *Virus Bulletin International Conference*, 1995.

[88] Jeffrey Wilhelm and Tzi-cker Chiueh. A forced sampled execution approach to kernel rootkit identification. In *Recent Advances in Intrusion Detection (RAID)*, 2007.

[89] Carsten Willems, Thorsten Holz, and Felix Freiling. Toward automated dynamic malware analysis using cwsandbox. *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2007.

[90] Yu Zhu, Sheng Liu Liu, Honghu Lu, and Wenbin Tang. Research on the detection technique of bootkit. In *International Conference on Graphic and Image Processing 2012*, 2013.