

**Blacksheep: a Tool for Kernel Rootkit Detection,
based on Physical Memory Crowdsourced Analysis**

BY

ANTONIO BIANCHI

B.S., Politecnico di Milano, 2008

M.S., Politecnico di Milano, 2012

THESIS

Submitted as partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Chicago, 2012

Chicago, Illinois

Defense Committee:

V.N. Venkatakrishnan, Chair and Advisor

Jakob Eriksson

Marco D. Santambrogio, Politecnico di Milano

ACKNOWLEDGMENTS

First of all I want to thank my parents and my brother because they have always helped me “in their own way”. I also have to thank Andrea Bianchessi and Carlo Ongini because we shared our school experiences for 12 years.

During all my academic career I have met tens of teachers, professors, TAs, RAs, and Ph.D. students. Most of them did their job with passion and skill and they have been a constant inspiration during my studies. I cannot list all of them, but I want to mention at least Alessandro Barengi and Simone Ceriani.

Big thanks go to Alessandra Bonetto that took care of us for four months at UIC, and to Manual Fossemò that made us laugh.

A huge thank you to Alberto Magni. You have helped me in countless situations and you never avoid answering my numerous questions.

I really have to thank Yanick Frantantonio for lots of different reasons. You always give me good suggestions and without you I would have never been at UCSB.

I want to thank Vincenzo Maffione, Daniele Di Proietto, Timon Van Overveldt, and Pierre Payet. Even though I had “stuffs to do” I really had good times being an intern at UCSB with you.

I also want to thank Yan Shoshitaishvili. You are the one who started this project and you patiently heard my complaints about “strange things” I found in Windows kernel.

ACKNOWLEDGMENTS (Continued)

A huge thank you to all the people at UCSB SecLab: it has been a privilege to work (and to party) with you.

I would like to thank prof. Carlo Ghezzi for his support.

I am truly thankful to professors Christopher Kruegel and Giovanni Vigna for the trust they have placed in me.

Finally, big thanks go to the people that helped me in reviewing this work: my mother and my brother, Alberto Magni, and Yan Shoshitaishvili.

AB

TABLE OF CONTENTS

<u>CHAPTER</u>		<u>PAGE</u>
1	INTRODUCTION	1
1.1	Motivations and goals	1
1.2	Definitions	8
1.3	Overcoming limitations of the current tools	11
2	RELATED WORK	13
2.1	Automatic rootkit detection	13
2.2	Physical memory analysis	18
2.3	Invariant-based detection	19
3	PHYSICAL MEMORY ANALYSIS ON WINDOWS OPER- ATING SYSTEMS	21
3.1	Memory Imaging Techniques	21
3.2	Windows memory management	24
3.3	Windows kernel modules	33
3.4	Windows common data structures	34
3.5	User-Kernel mode switching	38
4	WINDOWS ROOTKIT OVERVIEW	42
4.1	Rootkit classification	43
4.2	System modifications caused by kernel rootkits	44
4.3	Common rootkit uses	48
5	MEMORY DUMP ACQUISITION AND LOW-LEVEL AC- CESS	49
5.1	Dumping method used	49
5.2	Volatility integration	53
5.3	Blacksheep overview	58
5.4	Low level access layer	61
6	HIGH-LEVEL ANALYSES	70
6.1	Code analyses	71
6.2	Kernel entry point analyses	78
6.3	Data analyses	81
7	TECHNICAL DETAILS	91

TABLE OF CONTENTS (Continued)

<u>CHAPTER</u>		<u>PAGE</u>
7.1	Kernel pool allocation detection	91
7.2	Double linked list detection	93
7.3	Kernel module identification	94
7.4	Detecting code and data sections within a PE file	95
7.5	Data pointer detection	97
7.6	Function code comparison	97
7.7	Performance optimizations	99
8	EVALUATION	101
8.1	Used data sets	101
8.2	Tested rootkits	102
8.3	Methodology	106
8.4	Data set 1 - Trained Analyses	108
8.5	Data set 1 - Untrained Analyses	111
8.6	Data set 2 - Trained Analyses	113
8.7	Data set 2 - Untrained Analyses	116
8.8	Other results	118
8.9	Execution time and memory consumption	119
8.10	Discussion	122
9	CONCLUSIONS AND FUTURE WORK	125
	CITED LITERATURE	128
	VITA	132

LIST OF TABLES

<u>TABLE</u>		<u>PAGE</u>
I	OVERVIEW OF PHYSICAL MEMORY DUMPING METHODS	21
II	OVERVIEW OF TECHNIQUES USED BY WINDOWS ROOTKITS	42
III	THE SETS OF MEMORY DUMPS USED TO EVALUATE BLACKSHEEP	101
IV	ROOTKITS USED TO TEST BLACKSHEEP	103
V	ROOTKITS USED TO TEST BLACKSHEEP: WINDOWS 7 COMPATIBILITY AND HASHES	105
VI	OVERVIEW OF THE RESULTS IN <i>WINXP</i> SET, USING TRAINED ANALYSES	108
VII	OVERVIEW OF THE RESULTS IN <i>WINXP</i> SET, USING UN-TRAINED ANALYSES	111
VIII	OVERVIEW OF THE RESULTS IN <i>WINXP</i> SET, USING UN-TRAINED DATA ANALYSIS	112
IX	OVERVIEW OF THE RESULTS IN <i>WIN7</i> SET, USING TRAINED ANALYSES	113
X	OVERVIEW OF THE RESULTS IN <i>WIN7</i> SET, USING UN-TRAINED ANALYSES	116
XI	OVERVIEW OF THE RESULTS IN <i>WIN7</i> SET, USING UN-TRAINED DATA ANALYSIS	117
XII	NUMBER OF COMPARISONS NEEDED BY EACH TYPE OF ANALYSIS	119

LIST OF FIGURES

<u>FIGURE</u>		<u>PAGE</u>
1	Overview of the flow of data during utilization of <i>Blacksheep</i>	5
2	Execution privileges in x86-compatible CPUs	10
3	The number of signatures used by Symantec products	14
4	Page mapping in Windows 32-bit versions, with PAE enabled	27
5	PTE bits relative to address translation	28
6	Virtual address space layout in 32-bit Windows versions	32
7	Double Linked List examples	36
8	SYSENTER-based System Call Mechanism	40
9	Process hiding using DKOM	47
10	UML sequence diagram of the initialization of a MemDump instance	65
11	An example of the internal representation of a memory dump	66
12	An example of code untrained analysis with multiple infections . . .	118
13	Execution time during different types of comparisons	121

LISTINGS

<u>CHAPTER</u>		<u>PAGE</u>
1	<i>LIST_ENTRY</i> data structure definition	35
2	<i>POOL_HEADER</i> data structure definition	37
3	Procedure used to get the handle of the swap file	52
4	Procedure used to get a DWORD at a specified virtual address . .	69
5	An example of a detected double linked list	86
6	Procedure used to detect <i>POOL_HEADER</i> data structures	92
7	Procedure used to decide if an address is inside a code section . . .	96
8	Procedure used to compare the code of two functions	98
9	An example of speed optimization	100
10	A detected malicious data difference caused by a rootkit	110
11	Three examples of data modifications within the <i>CI.dll</i> module . .	115

LIST OF ABBREVIATIONS

AMD	Advanced Micro Devices
API	Application Program Interface
BIOS	Basic Input/Output System
CPU	Central Processing Unit
CRC	Cyclic Redundancy Check
DKOM	Direct Kernel Object Manipulation
DLL	Dynamic-link Library
DMA	Direct Memory Access
DWORD	A 32-bit unsigned integer
GDI	Graphics Device Interface
GDT	Global Descriptor Table
IAT	Import Address Table
IDT	Interrupt Descriptor Table
IOMMU	Input/Output Memory Management Unit
IRP	I/O Request Packet
KPCR	Kernel Processor Control Region

LIST OF ABBREVIATIONS (Continued)

KPP	Kernel Patch Protection
KVM	Kernel-based Virtual Machine
MBR	Master Boot Record
MD5	Message-Digest Algorithm (version 5)
MMU	Memory Manager Unit
MSR	Machine State Register
NDIS	Network Driver Interface Specification
OS	Operating System
PAE	Physical Address Extension
PCI	Peripheral Component Interconnect
PCIe	Peripheral Component Interconnect Express
PDE	Page Directory Entry
PE	Portable Executable
PFN	Page Frame Number
PTE	Page Table Entry
QEMU	Quick Emulator
RAM	Random Access Memory
RVA	Relative Virtual Address

LIST OF ABBREVIATIONS (Continued)

SHA	Secure Hash Algorithm
SP	Service Pack
SSDT	System Service Dispatch Table
SST	System Service Table
TLB	Transition Look-aside Buffer
TXT	Trusted Execution Technology
UAC	User Access Control
UML	Unified Modeling Language
UPGMA	Unweighted Pair Group Method with Arithmetic Mean
USB	Universal Serial Bus
VT	Vanderpool Technology
WDK	Windows Driver Kit

SUMMARY

Detecting rootkit infestations is a complicated security problem faced by modern organizations. Many possible solutions to this have been proposed in the last decade, but various drawbacks prevent these approaches from being ideal solutions.

In this thesis, we present *Blacksheep* a detection tool for utilizing a crowd of similar machines to detect rootkit infestations. In particular we focus on kernel rootkits infecting the Windows operating system.

We propose a novel technique to detect kernel rootkits based on the analysis of physical memory dumps acquired from a set of machines. These memory dumps are compared with each others and the results of these comparisons are used to classify them in infected and non-infected.

Three different comparisons are performed: code comparison, kernel entry point comparison and data comparison. Their results are used by two different analyses: a trained classification and an untrained classification. The trained classifier relies on a set of memory dumps manually flagged as having been acquired from machines in a non-infected state. The goal of this analysis is to classify a set of memory dumps as having come from infected or non-infected machines. The untrained classifier generates a hierarchy of clusters of memory dumps based on their similarity. The aim of this analysis is to separate the analyzed memory dumps into subsets based on the state of the machines which they have been taken from.

SUMMARY (Continued)

As part of our investigation into Windows kernel rootkits, much research was needed to be done in two main areas: the internals of the Windows kernel itself and the methods to acquire and analyze dumps of the physical memory and copies of the swap area. Part of our contribution is the summary of these researches.

We have tested *Blacksheep* on two sets of memory dumps acquired from differently configured machines infected with eight different rootkits. Some of the analyses performed by *Blacksheep* achieve a 100% detection rate, with no false positives in both sets. Others are able to give interesting information about the behaviors of the analyzed rootkits.

This dissertation is organized as follows.

- **Chapter 1** outlines the context of this work, its motivations and goals. It also provides an overview of how *Blacksheep* works.
- **Chapter 2** gives an overview of the state of the art in all the different research areas we worked on.
- **Chapter 3** details the crucial aspects that need to be considered during analysis of physical memory.
- **Chapter 4** provides a classification of Windows rootkits and of the techniques they use.
- **Chapter 5** explains how *Blacksheep* extracts data within a memory dump and how it is internally represented.

SUMMARY (Continued)

- **Chapter 6** describes the analyses performed by *Blacksheep*.
- **Chapter 7** provides details on some technical aspects about how *Blacksheep* works.
- **Chapter 8** presents the tests we performed and their results.
- **Chapter 9** summarizes the conclusions and depicts future development of the project.

CHAPTER 1

INTRODUCTION

1.1 Motivations and goals

1.1.1 Introduction

Modern organizations rely on large networks of computers to accomplish their daily workflows. In most cases, to simplify maintenance, upgrades, and replacement of these computers, these organizations utilize a standard build. For example, a large company might make a standard hard-disk image for employee workstations, another image for servers, and so forth. At the same time, these almost-similar computers are treated as unique entities when enforcing security policies and scanning for malware. Oftentimes, by leveraging the similarities between these computers, malware can be detected more effectively and without the limitations of modern malware detection techniques. Specifically, the detection of kernel rootkits can greatly benefit from such an approach.

Rootkits are software designed to stealthily modify the behavior of an operating system to achieve several goals such as hiding malicious user space objects (e.g. processes, files, network connections), logging keystrokes, disabling security software and installing backdoors for continued access. Although several detection and prevention techniques have been developed and deployed, all have considerable drawbacks, and so rootkits remain a widespread security threat: according to recent estimations the percentage of rootkits

among all antivirus detections is in the range of 7-10% (1)(2). The situation is further complicated by the fact that the techniques used by rootkits for evasion are continuously evolving (1).

The goal of our work is to detect kernel rootkits, a broad class of rootkits that operate by modifying kernel code or kernel data structures. We focus on the Windows operating system, since it is the most widespread and the most targeted platform, however most of the concepts and techniques used are applicable to any operating system. We propose a novel technique for detecting kernel rootkits, based on the analysis of physical memory dumps (and, if necessary, the swap area) of a running operating system.

We demonstrate *Blacksheep*, as an implementation of our approach and validate it by analyzing memory dumps acquired from two sets of computers. Two different types of analysis are performed by *Blacksheep*: a trained classification and an untrained classification.

The trained classifier relies on a set of memory dumps that have been manually flagged as acquired from a machine in a non-infected state. The goal of this analysis is to partition a set of memory dumps to be checked in infected and non-infected.

The untrained classifier generates a hierarchy of clusters of memory dumps based on their similarity. The aim of this analysis is to separate the tested memory dumps in subsets based on the state of the machines which they have been taken from. For instance, all memory dumps acquired from non-infected machines or all those taken from a machine with a specific infection, should be classified in separate clusters.

Both analyses depend on three different comparisons that are performed between pairs of dumps:

- code comparison
- kernel entry point comparison
- data comparison

This approach has several advantages over the state of the art. Rootkits using novel infection techniques can be detected. Additionally, *Blacksheep* can be deployed easily to a crowd of pre-existing machines. Since the entire population of machines is used in the comparison, the system can be installed on machines that are already compromised and still deliver useful detection. Finally, because *Blacksheep* detects the differences among the crowd, anti-virus software which modifies the kernel (often producing false positives for other rootkit detection techniques) can be properly accommodated, as such software would be deployed on all machines.

We validate our approach analyzing memory dumps taken from machines with two different configurations. In each configuration *Blacksheep* is able to detect kernel modifications introduced by all the different kernel rootkits we tried and to discriminate between memory dumps acquired from non-infected and infected machines. Additionally, it is also able to cluster memory dumps with an adequate precision according to whether they have been acquired from an infected machine and, in this case, to separate them based on the infection type.

The main contributions of our work are:

- To show that the analysis of physical memory dump and swap file is an effective way to detect Windows kernel rootkits.
- To infer invariant properties (kernel entry point values, code section content and data invariants) that a non-infected system should have, analyzing different dumps.
- To automate the rootkit detection process, checking if these properties hold in dumps acquired from machines to be tested.
- To build an untrained classifier of the infection state of a machine.

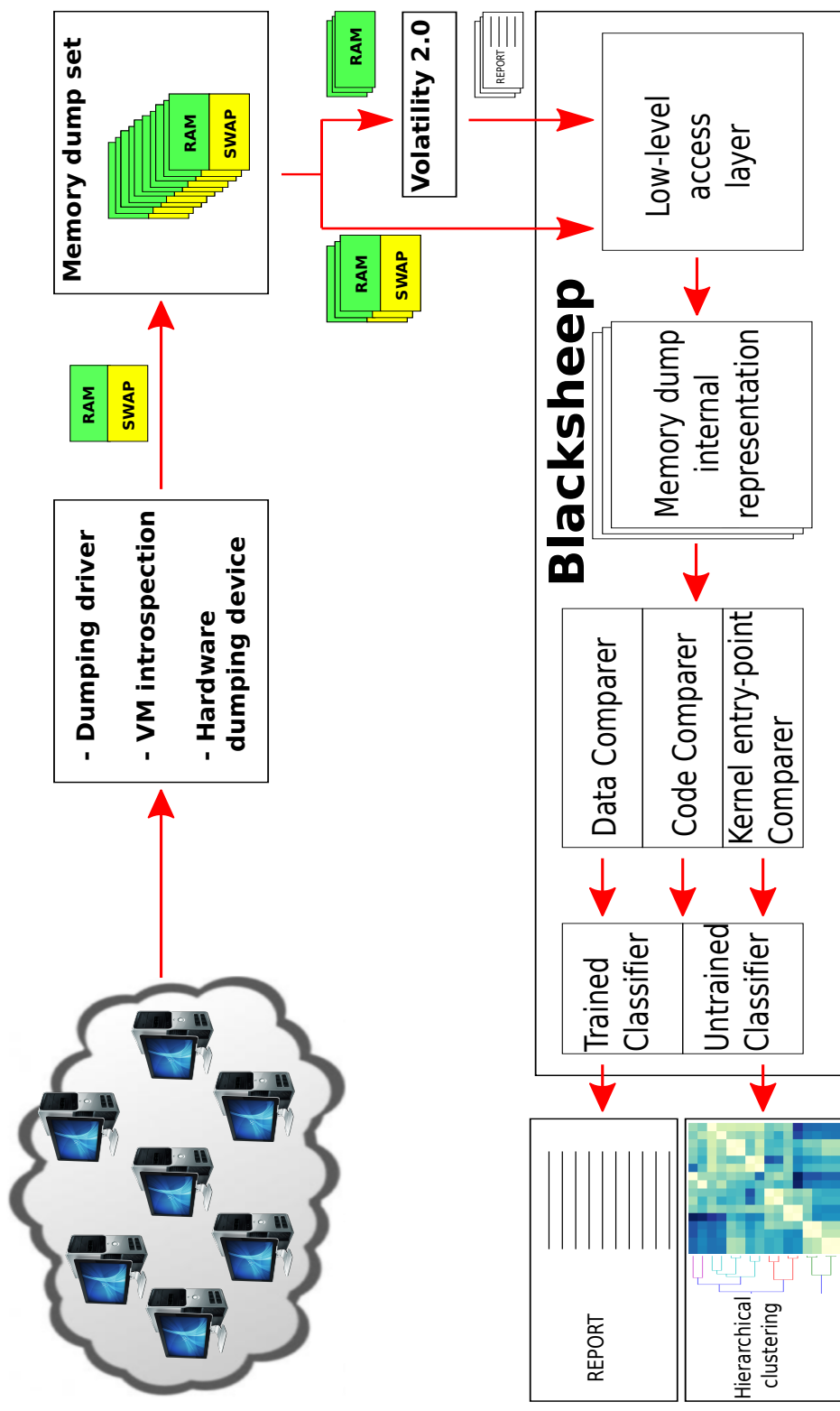


Figure 1: Overview of the flow of data during utilization of *Blacksheep*.

1.1.2 Common problems

Any rootkit detection methods has to deal with three common problems (3):

1. Knowing which security-sensitive locations of a system should be monitored or protected and what should be their expected state in a non-infected machine.
2. Getting reliable access to the system components they have to protect (e.g. processes, file system, network flows).
3. Protecting their own code and data from being tampered by malicious code.

Regarding the first problem, many security software tools analyze the integrity of several different Windows components (see Section 2.1.5). Developing such software requires a deep knowledge of the operating system internals and involves dealing with undocumented components and data structures. However, such systems can be easily circumvented by rootkits using previously unknown techniques to execute themselves and to subvert operating system behavior. For instance, kernel modules can use function pointers stored in their data sections that may be eventually modified by a rootkit: it is substantially impossible for a rootkit detection tool to check all these pointers. Additionally, it could even be difficult to verify the integrity of well-known system critical locations such as the system call table: these locations are modified by some legitimate software (see Section 4.3.2) and their content could change with any operating system updates.

The problem of getting a reliable access to system components is usually dealt with by attempting to access resources at the lowest possible level, oftentimes bypassing or over-

coming standard operating system APIs. Moreover, accessing resources through different methods can reveal system information hidden by rootkits (see Section 2.1.6). However, modern rootkits are able to cloak themselves in a way in which they are undetectable by several different access methods. For this reason, many modern rootkits can be detected and removed only using ad-hoc solutions.

Many security software face the problem of protecting their own code by hooking Windows APIs that could be used to disable them (see Section 4.3.2). Additionally, some of them allow users to scan their system before booting it, minimizing the possibilities a rootkit has to run while the scan is performed. Nonetheless, since security software and rootkits usually run with the same privileges it is not possible for the former to completely defend itself.

To solve limitations current products have, solutions based on hardware support have been proposed (see Section 2.1.5). However such solutions have not yet been implemented in commonly used security tools. Furthermore, such technologies could also be utilized by rootkits, allowing them to be executed at a higher privilege level than that at which the security software is (4).

For all of these reasons, the problem of the rootkit detection is still considered an arms race between rootkits and security software (5, Chapter 6).

The approach we propose is able to overcome problems that current approaches have, exploiting information collected by analyzing a set of similarly configured machines (see Section 1.3).

1.2 Definitions

Before going on with our discussion, it is necessary to state precise definitions of how some terms are used in the context of this thesis.

1.2.1 Malware

Malware is any malicious software, harmful to a computer user, designed to gather sensitive information, disrupt operations, gain unauthorized access to computer systems, and other abusive behavior. Worms, trojan horses, computer viruses, spyware, adware, rootkits, and other malicious programs are considered malware.

1.2.2 Rootkit

Rootkits are software designed to stealthily modify the behavior of an operating system to achieve several goals such as hiding malicious user space objects (e.g. processes, files, network connections), logging keystrokes, disabling security software, and installing backdoors.

1.2.3 Kernel Rootkit

A broad class of rootkits that operate by modifying kernel code or kernel data structures. Their code is executed at the same privilege level than the operating system kernel code. In this thesis, we restrain our analyses to kernel rootkits infecting the Windows operating system, unless otherwise specified.

1.2.4 Memory dump (shortened to *dump*)

A copy of the physical memory of a running operating system saved on a non-volatile memory. In case a system uses paging mechanisms (e.g. swap space) to store memory on a secondary storage (i.e. *pagefile.sys* file on Windows operating systems), a copy of this memory is considered as a part of the dump.

1.2.5 Kernel memory

The virtual memory region reserved by the operating system to its kernel. See section 3.2.3 for details.

1.2.6 Kernel module

We consider as a kernel module any unit of code that is executed at kernel level (Ring 0 in x86 terminology, see Figure 2). In Windows operating systems, they are usually loaded from files with *.sys* extension and a *LDR_DATA_TABLE_ENTRY* data structure is stored in kernel memory for each loaded module (see Section 3.3). We use this term to refer to both device drivers and operating system core components (e.g. *ntoskrnl.exe*, *hal.dll*, *win32k.sys*).

1.2.7 Kernel entry point

Any pointer to kernel code locations, used when execution is switched from user mode to kernel mode or an interrupt is processed. See section 3.5 for details.

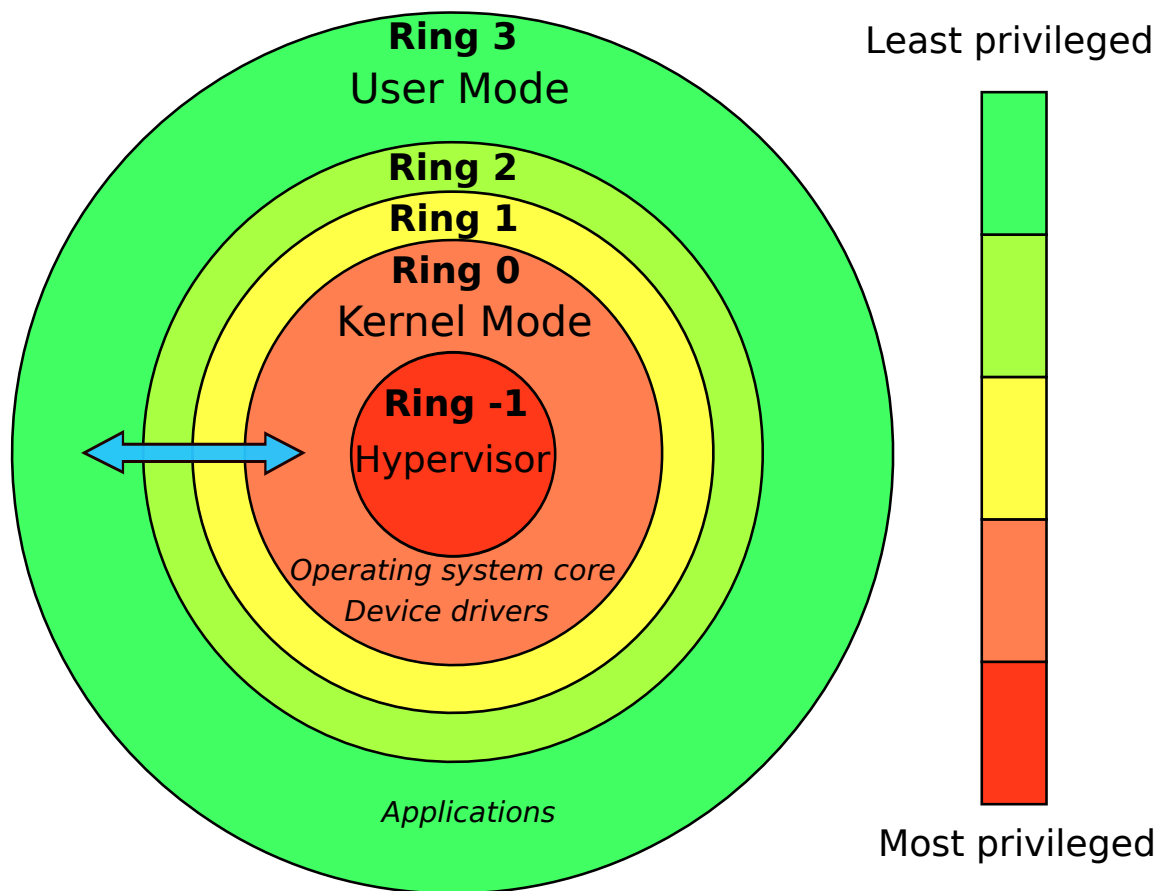


Figure 2: Execution privileges in x86-compatible CPUs.

Notice that in Windows operating systems, Ring 1 and Ring 2 privilege levels are not used, thus execution moves directly between Ring 3 and Ring 0. Hardware virtualization (in processors where it is available) can be used to run a hypervisor.

1.3 Overcoming limitations of the current tools

Blacksheep works on a set of memory dumps acquired from similarly configured machines. We state that, under the following conditions, it is able to overcome limitations that current detection methods have:

- Machines which the memory dumps are taken from must have the same software (same installed software versions and same operating system version) and hardware (installed peripherals).
- Most of the dumps must be taken from non-infected machines.

The problem of knowing which are the sensitive locations of a system to be monitored is addressed by analyzing all running kernel modules. Moreover, the analysis take into consideration both code and data. To know the *expected state* of such locations, the system exploits the assumption that most of the dumps are acquired from non-infected machines, thus the *expected state* is simply the most common among all the dumps. Since the analyzed machines are similarly configured, we expect that most of the kernel code, kernel entry points, and data invariants that we are able to detect should not change between dumps acquired from non-infected machines.

To get reliable access to system resources, we directly analyze physical memory. *Blacksheep* can work with physical memory dumps taken with different methods, some of these do not rely on any operating system APIs but directly read the content of the memory at the hardware level. Additionally, even if a rootkit could interfere with one of these meth-

ods, others can still be reliable (see Section 3.1). Furthermore, our work shows that it is possible to integrate physical memory images with copies of the swap area of an operating system to get a more complete analysis.

Since the analysis only needs memory dumps, it can be performed on different machines than those analyzed. For this reason the analysis code cannot be tampered by malicious code.

CHAPTER 2

RELATED WORK

Our work spans many different research areas related to rootkit detection and physical memory analysis. In this chapter we will discuss the state of the art in all the different areas we worked on.

2.1 Automatic rootkit detection

A considerable amount of research has been done both into detection of rootkit infections and in the development of countermeasures. Many of the studied techniques have been implemented in commercial products (6). Usually these tools work automatically, requiring limited user interaction. They are designed for users with little or no knowledge about operating system design and rootkit behaviors.

2.1.1 Malware detection techniques

Although rootkit detection poses specific problems, malware detection methods can sometimes be effective for this purpose as well.

2.1.2 Signature based detection

This classic method compares files and processes against a list of byte-level signatures describing invariant content of known malicious software (7)(8). This technique is still widely used, but it suffers major limitations. To begin with, as seen in Figure 3, the number of signatures which are required to detect currently known malware infections is ex-

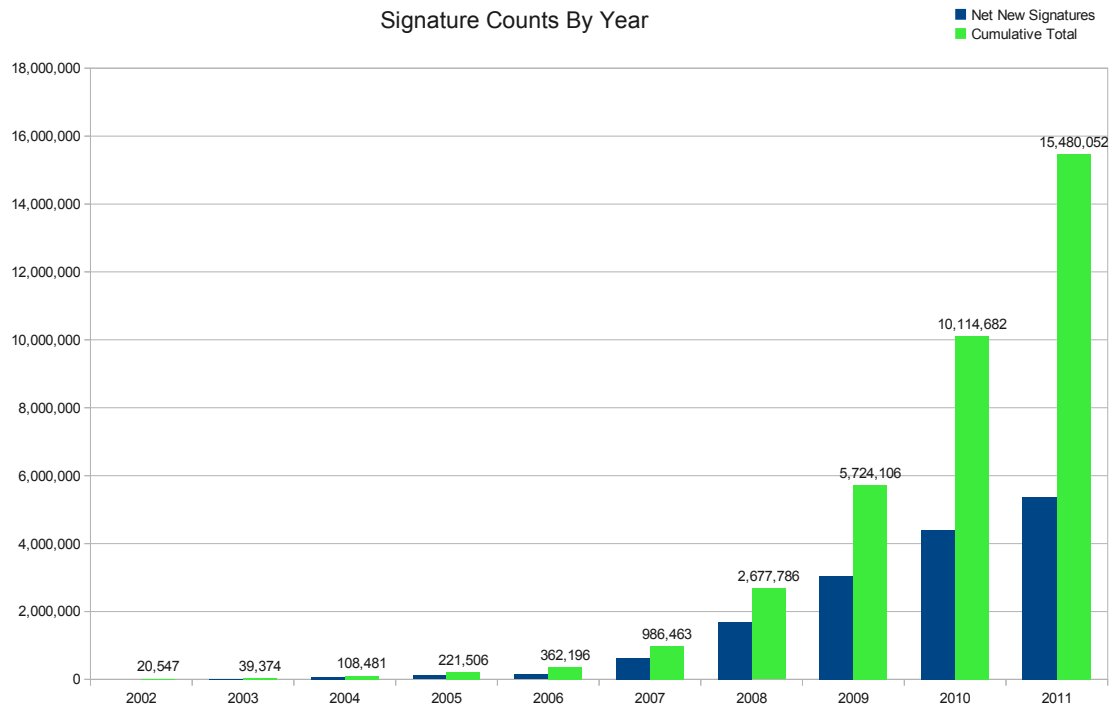


Figure 3: The number of signatures used by Symantec products.

source: http://www.triumfant.com/Signature_Counter.asp

ponentially increasing. Even taking into account only kernel rootkits, it is still difficult to generate signatures which describe polymorphic software. Additionally, writing such signatures takes time, and a completely new piece of malware often enjoys precious unhindered time while new signatures for it are manually generated.

Furthermore, such signature-based approaches generally utilize hooks in order to scan software as it is written to disk or loaded for execution. This is often accomplished by hooking system calls and other kernel entry points, but these methods can be evaded by adequately sophisticated software. For instance, some malware programs avoid saving themselves using filesystem APIs and instead write themselves to disk by accessing it directly (9). Other malware samples exploit undocumented and unmonitored mechanisms to be executed, thus evading detection by signature-based antivirus software.

2.1.3 Behavioral heuristic analysis

To overcome the limitations of signature-based detection, antivirus software often pairs it with a heuristic behavioral analysis. With this approach, a process is analyzed while it is running and its behavior is monitored.

For instance, a process which invokes some particular security-critical system calls with certain parameters (e.g. modifying file access permissions or adding boot entries) might be classified as suspicious and it might be halted. This approach has been studied in several works (10)(11).

Such analysis is very hard to execute properly. Any framework performing this analysis must have a very good understanding of the direct and indirect effects of monitored events. Such understanding is often imperfect, allowing malware to evade suspicions.

2.1.4 Sandboxed execution

Some malware detection software can execute a program in a virtual environment, isolated from the rest of the operating system, and trace actions performed by it. Using

this method they are able to deal with polymorphic malware, since it does not depend on the binary content of the file to be analyzed. Moreover it allows a behavioral heuristic analysis of a program executing it and, at the same time, preventing it to compromise the system in which it is run.

The two biggest drawbacks of this detection method are performance and evasion. Because detection software must wait until the sandbox execution has finished in order to start the program on the actual system, such a detection method is noticeably slow. Additionally, many different techniques allow malware to detect if they are executed inside an emulated environment, and, in such a case, modify their execution flow.

2.1.5 System integrity checking

In the process of subverting normal OS behavior, rootkits must modify critical system code and/or data structures (see Section 4.2 for details). For this reason, a possible rootkit detection method is to check if all critical components of an operating system are in their expected state.

Windows kernel, in 64-bit versions, implements a feature called *Kernel Patch Protection* (*KPP*) (12). It consists in an obfuscated, periodically executed, kernel function that checks the integrity of critical kernel code and data structures. ((13, Chapter 3.14) contains further information as to which kernel components are checked by KPP). Additionally, x64-based versions of Windows allow only signed code to be executed at kernel level (13, Chapter 3.15).

A similar approach has been implemented in System Virginty Verifier (14). This tool is based on the idea that, excluding some specific locations (e.g. relocated pointers, data sections), the image in memory of a kernel module should be equal to the content of the file from which it is loaded.

Other approaches, specifically designed as a defense against function pointer hijacking in kernel memory, have also been developed (15)(16).

Yet more solutions to system integrity checking problem based on hardware virtualization processor features (e.g. Intel VT-x, Intel VT-d, Intel TXT, AMD-V, AMD-Vi) have been proposed (17)(18). The idea behind these solutions is to take advantage of hardware virtualization to perform integrity verification at a higher privilege level than that at which the kernel code and rootkits are executed.

One fundamental challenge with all such systems is the fact that they must identify a baseline with which to compare the current state of the system that they are protecting. In the case of the System Virginty Verifier, the baseline is defined to be the actual files on disk from which the kernel is loaded. However, malware that is motivated enough could modify these files as well, thus corrupting the baseline. In other cases, the state of the system when the software was loaded is used.

2.1.6 Cross-view detection

Cross-view detection is a another popular rootkit detection technique (19, Chapter 7.9), which is implemented by several detection tools. This approach relies on the fact that the same information about the state of a system can be obtained in different ways.

For instance, the common way to detect the presence of a file is to use specific user-level APIs. The values returned by such APIs can be easily altered by a rootkit to hide the presence of a file. However, scanning low-level file system structures can reveal a file hidden in such a way. So, comparing the results collected enumerating files using user level APIs with those collected directly scanning file system structures, can detect discrepancies caused by a rootkit.

Another common anomaly that can be detected with this approach is that introduced when a process is hidden unlinking it from the process list used by the Windows kernel (see Section 4.2.5 for details).

2.2 Physical memory analysis

Physical memory analysis is an active area of research which aim is to capture reliable and complete information from a live dump of the physical memory of a running system. It has been studied mainly for forensic and malware analysis (20)(21) and different specialized tools exist (e.g. *HBGary Responder Pro* (22), *Volatility* (23)).

Volatility is an open source framework for physical memory analysis. It has an extensible plugin structure that allows the implementation of several different types of analyses, in particular a plugin designed specifically to malware detection and analysis has been developed (24). Specific studies have been carried on the detection of memory allocations in the Windows kernel memory pools (25)(26). The use of information extracted from the Windows swap file has been examined too (albeit, mainly for forensic purposes (27)).

2.3 Invariant-based detection

The problem of kernel integrity verification is closely linked to that of discovering and verifying invariant properties within the kernel memory. In particular, several works have been proposed in order to detect invariant properties in kernel data structures. Such invariant properties can then be automatically checked to ensure that the integrity of kernel data structures has not been violated by malicious software to subvert normal kernel behavior.

Petroni et al. (28) propose a framework to manually specify kernel data invariants and to check them automatically. This framework allows to easily declare properties that must hold inside an intact machine. However, manually specifying such properties requires a deep knowledge of operating system internals and it is particularly difficult when no source code is available. Furthermore, in a modern operating system the attack surface a rootkit has modifying data structures, is very large. For this reason, it is not feasible to manually state all data invariants that must hold in an uncompromised system. Additionally, even if source code is present and invariants automatically specified, the ability to load kernel-resident modules still makes this task impossible, as the contents of the kernel cannot always be known ahead of time with complete certainty.

The state of the art in automatic invariants detection is implemented in Daikon (29). It is a tool developed to automatically discover pre-conditions and post-conditions that hold when program functions are called. Baliga et al. have adapted Daikon to work on kernel

data structures. Gibraltar (30), the tool they have developed, is able to detect previously-known rootkit modifying data structures of Linux kernel. It works in two steps: initially, it extracts data structures, parsing Linux kernel source code, then kernel memory snapshots are used to infer invariant properties on fields of the data structures previously extracted.

A similar approach has been implemented for Windows kernel in KOP (31). Differently from the solution we propose, this tool needs Windows kernel source code to extract a graph of kernel data structure relationships.

The use of invariants based on graph-signatures, have been explored also by Sig-Graph (32). This tool is able to work simply scanning memory snapshots, without needing the availability of the source code.

CHAPTER 3

PHYSICAL MEMORY ANALYSIS ON WINDOWS OPERATING SYSTEMS

3.1 Memory Imaging Techniques

TABLE I: OVERVIEW OF PHYSICAL MEMORY DUMPING METHODS.

Method	Presence of dump artifacts	Can dump swap file?	Can be tampered?	Deployment
Software methods	High	Yes	Very Easily	Easy
Crash dump Hibernation file	High	No	Easily	Difficult, not always possible
Physical devices	Low	No	Hardly	Difficult
Virtual machine introspection	Very low or None	Yes	Very Hardly	Difficult, not always possible

Physical memory analysis is an effective way to detect rootkits, because of the so called Rootkit Paradox (33). In fact, even if rootkits try to hide themselves, in order to run, the operating system has to be able to find and execute them. In addition, many different methods to dump memory exist, some of them are very difficult to be tampered since they relay on components external to the analyzed system (external physical devices, virtual machine host system).

3.1.1 Software methods

Different software have been developed to dump physical memory of a running Windows system. They usually relay on accessing `\\Device\\PhysicalMemory`, a device created by Windows OS, mapping the whole physical memory of the running system. For security reasons, the access to such device has been restricted in modern Windows versions (Windows 2003 SP1 or newer) only to code running in kernel-mode.

Memory dumps created in this way have several artifacts for two different reasons. First of all, the dumping application itself must be loaded in memory modifying it. More importantly dumping memory by software is not an atomic operation, so, while the dumping procedure is performed, memory itself is modified continuously. The most critical modifications are those occurring in paging tables after that the memory regions where these tables reside have been already read by the dumping process (see Section 3.2.1). For this reason, several inconsistencies are expected in memory dumps obtained in this way.

Dumping software can be easily tampered by rootkits. Using debug registers or modifying paging tables it is possible to hide specific memory locations to specific processes.

Moreover, a rootkit can easily monitor all active processes and disable those that are known to be used to dump physical memory.

3.1.2 Crash dump and Hibernation file

When a Windows system crashes or it is hibernated, the operating system saves a copy of the memory. Such copy can then be read by memory analysis software. Even if these methods are effective in creating memory dumps, they are not feasible for a widespread usage, since they require the system to be rebooted.

3.1.3 Physical devices

Hardware solutions have been proposed for dumping physical memory, exploiting the fact that external peripherals could have direct access to the system memory (DMA). In particular, hardware devices working on Firewire, PCI, PCIe, and ExpressCard interfaces are available.

This method does not need any running software on the target system, but some inconsistencies in the dumped memory are still possible if the system is not suspended while dumping. Additionally, some specific memory locations cannot be accessed by this method.

Techniques to avoid the dumping of some memory regions by hardware devices have been studied (34). However, they are very specific on the hardware configuration of the target machine and they are not used by common rootkits.

3.1.4 Virtual machine introspection

When a system is running inside a virtual machine, the virtualization software, located on the host operating system, can easily save the memory of the guest system. In QEMU, for instance, it is possible to use the *pmemsave* command to do so.

Dump artifacts are minimized because while the memory dump is acquired the guest operating system is suspended. Minor inconsistencies are however still possible due to in-progress memory writes, especially in multi-processor systems.

Excluding virtual machine escaping techniques, the dumping process can not be tampered by rootkits running on guest operating system because it runs inside the host. However, a rootkit could use virtual machine detection techniques to modify its behavior when running inside a virtual machine, not to be detected.

3.2 Windows memory management

In this section we will briefly summarize how memory management works under Windows. For an in-detail explanation refer to (13, Chapter 9) and (35, Chapters 3 and 4). We will focus in particular on Windows kernel virtual address layout and the virtual to physical translation mechanism.

Unless otherwise specified, we will refer to Windows 32-bit versions running on an x86-compatible processor, with Physical Address Extension (PAE) enabled and the `/3GB`¹ boot switch disabled, because it is the most common configuration among Windows machines.

The two primary tasks Windows memory management fulfills are:

- Provide to each running process a separate virtual address space.
- Paging some areas of the memory to the disk when necessary.

Understanding how memory is managed is the first crucial step in memory analysis of any running operating system. Tools working on memory dumps must know how memory is managed by the CPU and the operating system, to access it coherently on how it is done in a running system.

In general, two problems need to be dealt with. Firstly, since all modern operating systems use virtual memory, it is necessary to understand how to convert virtual addresses to physical ones and vice-versa. In a running system this translation is performed in hardware by the Memory Management Unit (MMU) of the CPU and accelerated by the Transition Look-aside Buffer (TLB). The operating system can enhance this process, for instance implementing a swap area or particular protection policies on memory regions.

¹`/3GB` split is a special boot option used to extend the amount of virtual memory available to each user-mode application. When it is enabled only virtual addresses between `0xC0000000` and `0xFFFFFFFF` are assigned to the kernel, leaving 3GB of virtual memory available to user-mode applications. On the contrary, when it is disabled, virtual addresses between `0x80000000` and `0xFFFFFFFF` are assigned to the kernel.

Secondly, it is necessary to know which are the virtual locations where data, relevant to the required analysis, are stored.

It should also be noted that some addresses are mapped by the operating system to peripherals (e.g. video cards), configuring accordingly the input/output memory management unit (IOMMU). In this way such peripherals can directly access to system memory without using the CPU. These regions are present in memory dumps, even if they are usually ignored during analyses.

3.2.1 Virtual to physical address translation

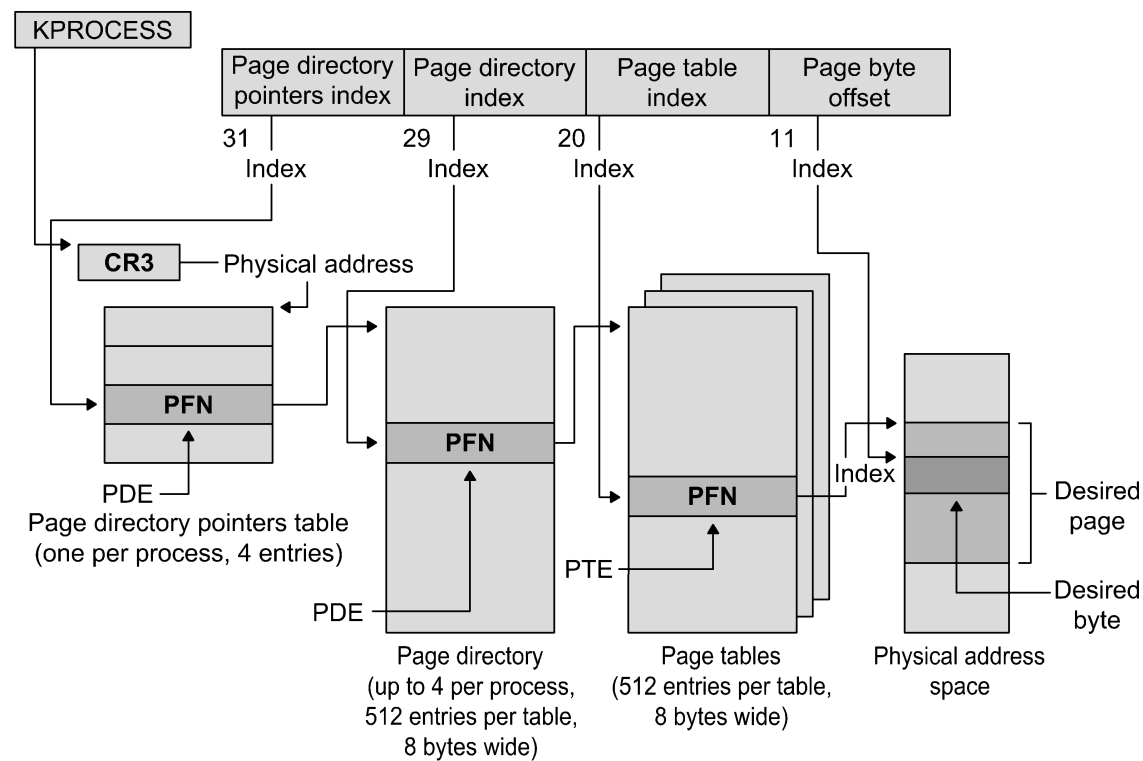


Figure 4: Page mapping in Windows 32-bit versions, with PAE enabled.

source: (13, page 769)

A x86-compatible processor provides two different mechanisms to translate virtual addresses to physical ones: segmentation and paging. Windows operating systems use mainly paging, configuring segmentation just to a flat segment spanning all available virtual addresses (from 0x00000000 to 0xFFFFFFFF in 32-bit systems).

Starting from the physical address stored in the *CR3* register, the CPU parses a table structure to translate each virtual address to the corresponding physical one (unless the required value is found in the TLB). The operating system saves the *CR3* value of each running process inside a *KPROCESS* data structure and changes its value during task switches. It is also responsible to create and modify paging tables correctly.

In addition, the memory manager keeps track of each page of memory in an array called the Page Frame Number database (PFN).

3.2.2 Page types

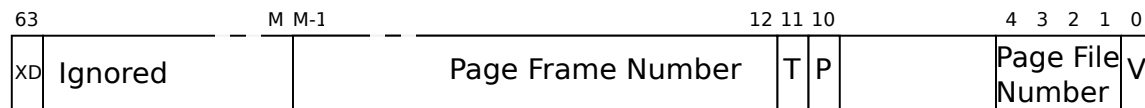


Figure 5: PTE bits relative to address translation (32-bit, PAE enabled).

XD: Execution Disabled – *V*: Valid – *P*: Prototype – *T*: Transition –
M: MAXPHYADDR ($52 \leq \text{MAXPHYADDR} \leq 32$, depending on processor capabilities)

Virtual memory is usually partitioned in 4KB pages, each of them is indexed by a Page Table Entry (PTE). Larger pages are possible (4MB or 2MB if PAE is enabled), for instance, we have noticed that *ntoskrnl.exe* is loaded in large pages for performance reasons. Any large page is indexed by a Page Directory Entry (PDE). In this paragraph we will always refer to 4KB pages.

When the *Valid* bit is set the translation is performed automatically by the CPU, transparently to the operating system. On the contrary, if the *Valid* bit is not set, the page is considered as invalid, so a *PAGE_FAULT* exception is generated by the processor. This exception is then handled by the OS. It checks if the requested page is really unavailable or some operations need to be performed to retrieve it. In this case, some flags and values stored in the PTE record specify the category of the invalid PTE and how the exception needs to be treated. In the following paragraphs, we will discuss only categories relevant to the analyses we have performed.

Page File

Both *Prototype* and *Transition* bits are zero, Page Frame Number is different from zero.

This entry points to a frame in a paging file. Paging files are used as a swap area by Windows when the system memory is overcommitted. The use of more than one paging file is allowed, however, in common configurations, only one is used, usually stored in *C:\pagefile.sys* location.

The offset of the desired page within the pagefile is in bits 12—(*MAXPHYADDR*-1), bits 1—4 give the page file identification number.

Transition

Prototype bit is zero and *Transition* bit is one.

These pages have been modified, but not yet written back to a paging file. Transition pages are still in system memory and they can be retrieved by using value stored in bits 12—(*MAXPHYADDR*-1), like any valid entry.

Mapped File

Prototype bit is one and the PTE entry is inside a prototype PTE¹.

A Mapped File is a file that is loaded into system memory on demand. For instance, when Windows executes code from any file, it is loaded as a mapped file. Thus, its content is not immediately read from disk and copied in memory, but the reading procedure is performed only when code or data inside a given page needs to be accessed.

For this reason it is common that some pages belonging to an executable file are not present in system memory. This is a problem we had to deal with during the development of *Blacksheep* (see Section 6.1.1).

Even if the execute-disable (*XD*) bit (only available when PAE is enabled) is not used in the address translation mechanism, during the development of *Blacksheep* we have analyzed how it is used by Windows kernel. When the XD bit is set, the content of the page cannot

¹Prototype PTEs are special PTEs that can be shared among different processes. They are pointed by regular invalid PTEs.

be executed. So, this value could be used by an OS to enforce the security policy that pages containing data must not be executable. While analyzing a memory dump this value could then be used to easily detect pages containing code. However, we have noticed that this bit is rarely used by Windows kernel. For this reason, it is ignored by *Blacksheep*.

3.2.3 Kernel virtual addresses

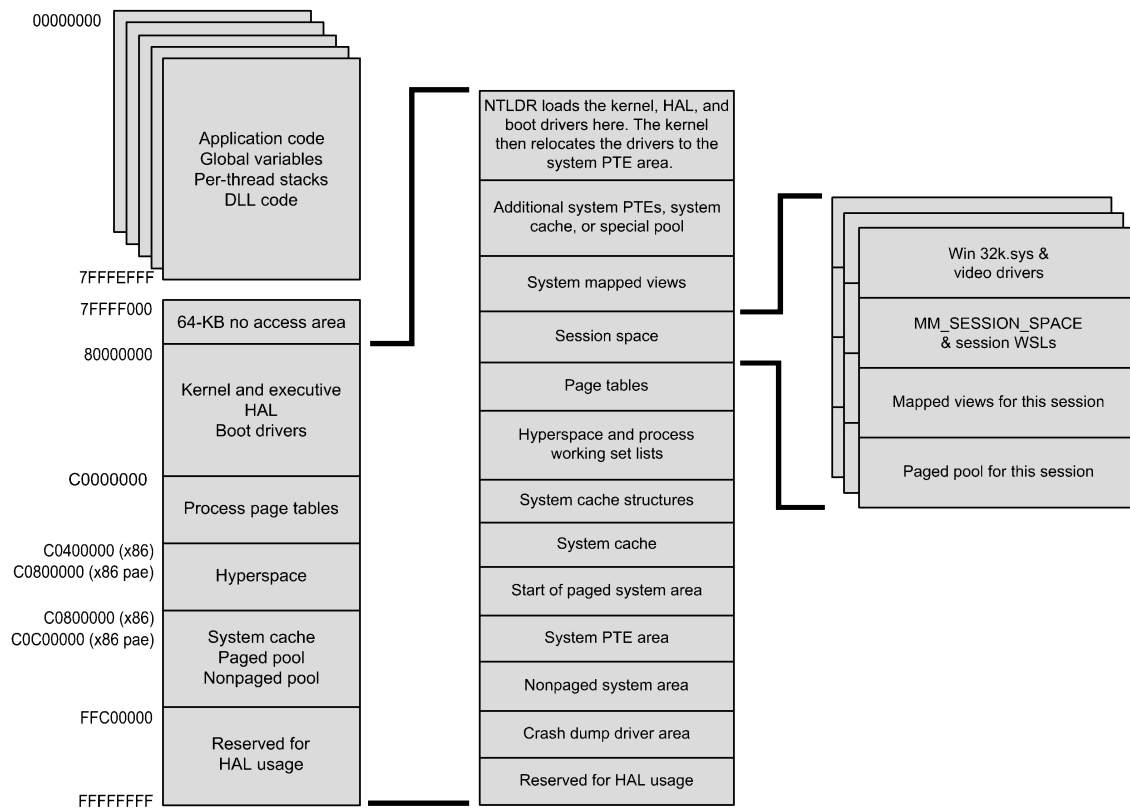


Figure 6: Virtual address space layout in Windows 32-bit versions.

source: (13, page 738 and 741)

As shown in Figure 6 each process has a dedicated virtual address space from 0x00000000 to 0x7FFFFFFF. On the contrary, kernel virtual addresses (from 0x80000000 to 0xFFFFFFFF) are shared between every process. For this reason we could assume that, for what concerns analyses performed by *Blacksheep*, it exists an unique mapping between virtual addresses and physical ones.

Session space is a significant exception to this assumption. Windows supports multiple sessions running at the same time, even if in common usage scenarios, only one session is active. Therefore kernel memory related to sessions can be mapped differently among processes.

It should be noticed that page tables are stored at virtual addresses inside the kernel memory. Virtual addresses mapped to peripherals are in kernel memory too, for instance, usually hundreds of Megabytes are assigned to the graphic card.

3.3 Windows kernel modules

Kernel modules are PE files (36), loaded in kernel memory and executed at Ring 0 privilege level. Some of them, such as Windows core modules (e.g. *ntoskrnl.exe*, *hal.dll*, *win32k.sys*, *kdcom.dll*, *BOOTVID.dll*) or most of the device drivers, are loaded during the boot process, others are loaded when necessary. For instance some processes need to load a kernel driver to perform system-level operations or some device drivers are loaded only when a peripheral (e.g. an USB device) has been connected.

As every PE file, a kernel module is composed by a header and some sections, each of them has a name and some associated flags. Imported and exported functions are listed in specific data structures pointed by the PE header.

Kernel modules are loaded at a random location, their code is automatically relocated by the loader, according to their base address, in locations listed in a specific section (usually named *.reloc*).

3.4 Windows common data structures

Hundreds of data structures are used by Windows kernel. Some of them are documented in the official Windows documentation¹, others in the Windows Driver Kit² in the form of debugging symbols.

Blacksheep deals explicitly with two of them: double linked lists (*LIST_ENTRY*) and kernel pool allocations (*POOL_HEADER*).

¹<http://msdn.microsoft.com/en-us/library>

²<http://msdn.microsoft.com/library/windows/hardware/gg487428>

3.4.1 Double linked list

```
1 typedef struct _LIST_ENTRY
2 {
3     PLIST_ENTRY Flink;
4     PLIST_ENTRY Blink;
5 } LIST_ENTRY, *PLIST_ENTRY;
```

Listing 1: *LIST_ENTRY* data structure definition.

Many different kernel lists, such as the list of all running processes, are stored in memory as double linked lists. When a list is empty, only the list head is present (as shown in Figure 7). Otherwise elements are added, creating a circularly linked structure.

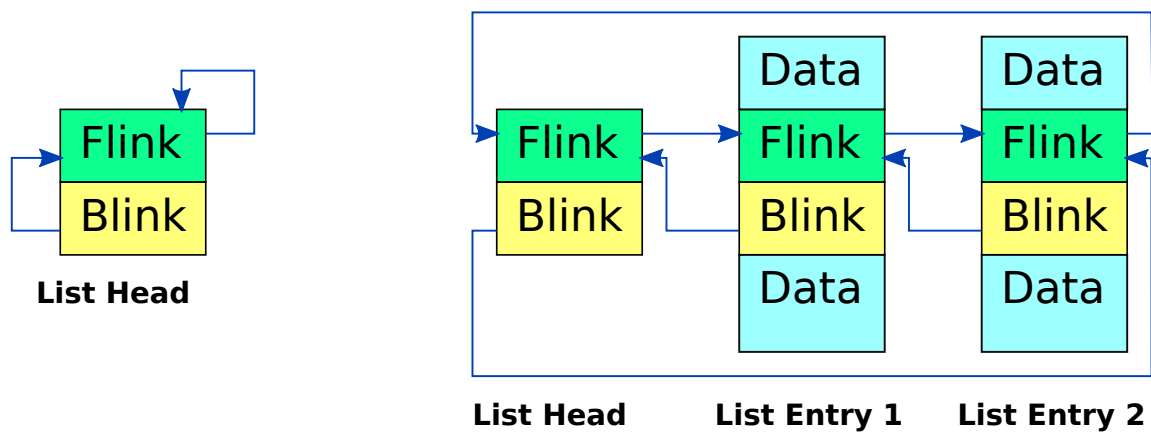


Figure 7: An empty Double Linked List (on the left) and one with two elements (on the right)

It is worth to note that the *LIST_ENTRY* structure is used in two different roles: as a list head or as a list entry. When it is used as a list entry, it can be placed at any position inside a bigger data structure.

3.4.2 Kernel pool allocation

```
1 typedef struct _POOLHEADER
2 {
3     ULONG PreviousSize: 9;
4     ULONG PoolIndex: 7;
5     ULONG BlockSize: 9;
6     ULONG PoolType: 7;
7     ULONG PoolTag;
8 } POOLHEADER, *PPOOLHEADER;
```

Listing 2: *POOL_HEADER* data structure definition (when a pool tag is specified).

When kernel modules require to allocate memory they use specific APIs to allocate it inside system pools. These pools are used as common heaps shared by code running in kernel-mode. Typically the API *nt!ExAllocatePool* is used.

Such allocations are wrapped inside a *POOL_HEADER* data structure, thus they can be recognized scanning a memory dump (25). *BlockSize* field can be used to determine the size of the allocated memory.

Usually a pool tag is specified. It is a four-character string that identifies the code allocating the memory. A list of tags used by Windows kernel modules is available in the WDK.

3.5 User-Kernel mode switching

Several mechanisms are used by Windows to switch execution from user-mode to kernel-mode and to handle hardware interrupts. In response to such events different kernel function pointers are used. We call these function pointers *kernel entry points*, since they are addresses where kernel-mode execution starts.

The analysis of such pointers is crucial in rootkit detection tools, since rootkits frequently modify their values allowing their functions to be executed when a specific event occurs. In this way, a rootkit ensures that its code is executed, moreover hooking such functions allows it to subvert kernel behavior, filtering kernel function invocations.

3.5.1 Interrupt Description Table (IDT)

IDT is the hardware mechanism provided by x86-compatible processors to allow OS response to interrupts. It is a vector of at most 256 function pointers, every function is associated to an interrupt and it is automatically invoked when the corresponding interrupt is raised. Windows uses only a limited set of interrupts, assigning the remaining ones to generic functions (named *nt!KiUnexpectedInterruptXX*) In a non-infected system, all interrupts are associated to kernel functions inside *ntoskrnl.exe* or *hal.dll* modules.

Interrupt *0x2E* was used by old operating systems to switch to kernel-mode when a system call was performed. Even though, on modern CPUs, Windows uses *SYSENTER*

instruction, this IDT entry is still set to *nt!KiSystemService* API. Such kernel function is used as a generic entry point for all invoked system calls.

3.5.2 SYSENTER

SYSENTER assembler instruction is used to quickly switch from user-mode to kernel-mode execution. When this instruction is called, the execution moves to an address set in particular machine-specific registers (MSRs). Windows sets these registers in a way that when *SYSENTER* instruction is executed the function *nt!KiFastCallEntry* is called.

When any system call is invoked, a *SYSENTER* instruction is executed, as a consequence the kernel function *nt!KiFastCallEntry* is called. Such function, in turn, calls the requested system call, according to the value stored in the *EAX* register and the currently active thread (see Figure 8).

3.5.3 System Service Dispatch Table

A System Service Dispatch Table (SSDT) is an array of virtual addresses, where each address is the entry point of a kernel function. When any kernel function is invoked, *nt!KiSystemService* read this table to know its entry point. The address where the SSDT is located is specified on per-thread base in the *KTHREAD* data structure. Moreover, a thread can use more than one SSDT.

Usually, all threads share the same two SSDTs (*KiSystemService* and *W32pServiceTable*), however a rootkit can create a new SSDT and modify a *KTHREAD* structure to make the associated thread using the new SSDT. In this way, it may avoid to be detected by tools checking only the two canonical SSDTs.

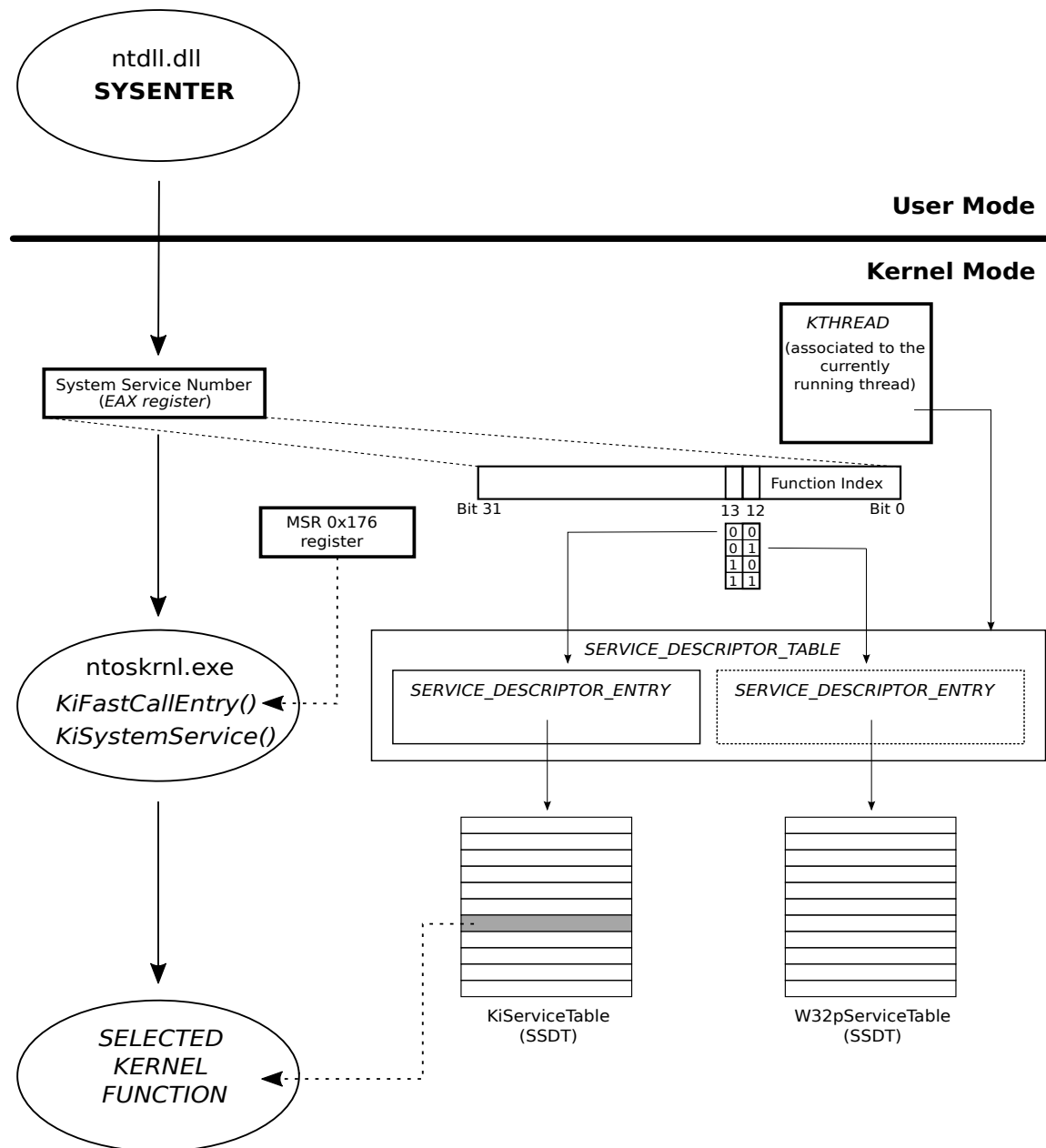


Figure 8: SYSENTER-based System Call Mechanism.

Usually two different SSDTs are used: *KeServiceDescriptorTable* contains pointer to Windows native APIs (implemented by *ntoskrnl.exe* module), *W32pServiceTable* contains pointers to user and GDI functions (implemented by *win32k.sys* module).

Some threads only use *KiSystemService* SSDT, leaving empty the *SERVICE_DESCRIPTOR_ENTRY* usually pointing to *W32pServiceTable*.

3.5.4 Call Gates

Call Gates are an additional way to transfer control between CPU privilege levels. Call Gate descriptors are specified in the Global Descriptor Table (GDT), a data structure used by x86-compatible processors, defining characteristics of various memory areas.

Even if Call Gates are not used by modern operating systems, they can still be configured by rootkits as a backdoor, allowing user-mode applications to call custom kernel-mode functions, without performing any other system modifications.

3.5.5 I/O request packet handlers

I/O request packets (IRPs) are kernel data structures used by kernel modules to communicate with each other and with user-mode code. When a kernel module is loaded, an array of function pointers (one for each IRP type the module can handle) is initialized. Each of these functions is invoked when the corresponding I/O request is received by the module.

CHAPTER 4

WINDOWS ROOTKIT OVERVIEW

TABLE II: OVERVIEW OF TECHNIQUES USED BY WINDOWS ROOTKITS.

Techniques written in blue, red and green can be detected respectively by code, kernel entry point and data analysis performed by Blacksheep.

	Code	Data
User Space	Patch binary file Patch memory image Inject a DLL Inject a thread	Hook process IAT
Kernel Space	Patch binary file Patch memory image Install a filter driver Modify MBR	Modify Kernel Objects (DKOM) Modify module data structures Hook Windows structures (SSDT, IRP) Hook processor structures (IDT, GDT, SYSENTER)

Table II shows the most common techniques used by Windows rootkits. For an in-detail description of each of them, refer to (19)(37)(5, Part II). In this chapter we will give a brief description of the topic, focusing on techniques that cause Windows kernel modifications detectable by *Blacksheep*.

4.1 Rootkit classification

Rootkits are generally classified according to the execution environment in which their code runs.

4.1.1 User-mode rootkits

They run at Ring 3 privilege level as any standard Windows application.

4.1.2 Kernel-mode rootkits

They run at Ring 0 privilege level as a Windows kernel module. Some kernel-mode rootkits are also called bootkit, since they modify operating system bootloader, typically the disk Master Boot Record (MBR).

4.1.3 Hypervisor rootkits

They exploit hardware virtualization functionalities to run at a higher privilege level than the operating system.

4.1.4 Firmware rootkits

They modify device or platform firmware to create a persistent copy of their code in hardware such as graphic card, network card or BIOS.

Hypervisor and firmware rootkits have been studied and developed in recent years, but they are not widespread because of their complexity. On the contrary, user-mode rootkits have a significant distribution, but they are easily detectable by existing tools running at Ring 0 privilege level. For these reasons we focus our work on kernel-mode rootkits.

4.2 System modifications caused by kernel rootkits

4.2.1 Installing a kernel module

The code of many kernel rootkits is stored inside a kernel module and loaded during Windows boot-process. The standard way of installing a kernel module consists in saving it into the filesystem and adding it to the list of modules to be loaded at runtime by updating Windows register keys. Such kernel modules can usually be detected by using specific APIs. However, a rootkit can subvert the system to hide its module.

Kernel modules installed by rootkits often act as *filter drivers*. In Windows a filter driver is a kernel module that is inserted into an existing driver stack to add functionalities to a controlled device. For instance, the access to a file on a disk is managed by a stack of different drivers (from the filesystem structure manager to the physical interface with the disk). Inserting a filter driver in this stack allows a rootkit to filter all requested filesystem accesses.

64-bit Windows versions require every kernel module to be signed, refusing to load unsigned ones. Even though only verified modules, provided by trusted developers, should

be signed, some kernel rootkits have been correctly signed by a trusted authority (e.g. Stuxnet (38)).

4.2.2 Kernel “entry point” modification

Function pointers discussed in Section 3.5, are typically modified by rootkits, overwriting some of them to point to their own code. This provides a stealthy way for rootkit to be executed. Additionally it allows a rootkit to filter system function invocations. For instance, a rootkit can hide a file by changing all filesystem-related APIs to point to its own code and then returning fake results when any user-mode application tries to access such file.

4.2.3 Kernel function hooking

Another way to alter system functions is to modify their code in a way that, when they are invoked, execution is redirected to rootkit functions. This technique is usually referred as hooking and provides similar functionality to kernel “entry point” modification.

Functions called using code pointers (typically imported and exported functions) can be hooked by simply modifying such pointers. Other functions are usually hooked by overwriting their first bytes with any of the following assembler instructions:

- an indirect **call** or an indirect **jmp**.
- a direct **call** or a direct **jmp**.
- a **push** instruction followed by a **ret** instruction.

However, more complex (and less easily detectable) modifications are possible.

4.2.4 Kernel module code patching

Code of legitimate kernel modules can be modified by rootkits to subvert their behavior. This technique can be at least as effective as function hooking and more difficult to be detected, because modifications can occur at any position inside the code of a kernel module. It requires a deep understanding of how a module works. Additionally, each different version of a kernel module needs to be modified in a different way.

4.2.5 Direct Kernel Object Manipulation

Rootkits can subvert a system by modifying data structures within the kernel. This technique is called Direct Kernel Object Manipulation (DKOM). It is considerably more difficult to be detect than code related modifications, yet it can be effectively used by a rootkit to perform malicious tasks such as hiding processes, kernel modules, and network connections. Altering kernel data structures can also lead the operating system to stealthily execute rootkit code. This attack scenario has been studied for Network Driver Interface Specification (NDIS) kernel modules (39).

A classical example of how DKOM can be used is shown in Figure 9. The Windows kernel keeps track of the currently running processes in a specific double linked list of *EPROCESS* data structures. Unlinking an element from such list causes the corresponding process to be hidden from any Windows APIs. However, since the Windows scheduler is based on threads (that are enumerated using other kernel lists), the process can still run.

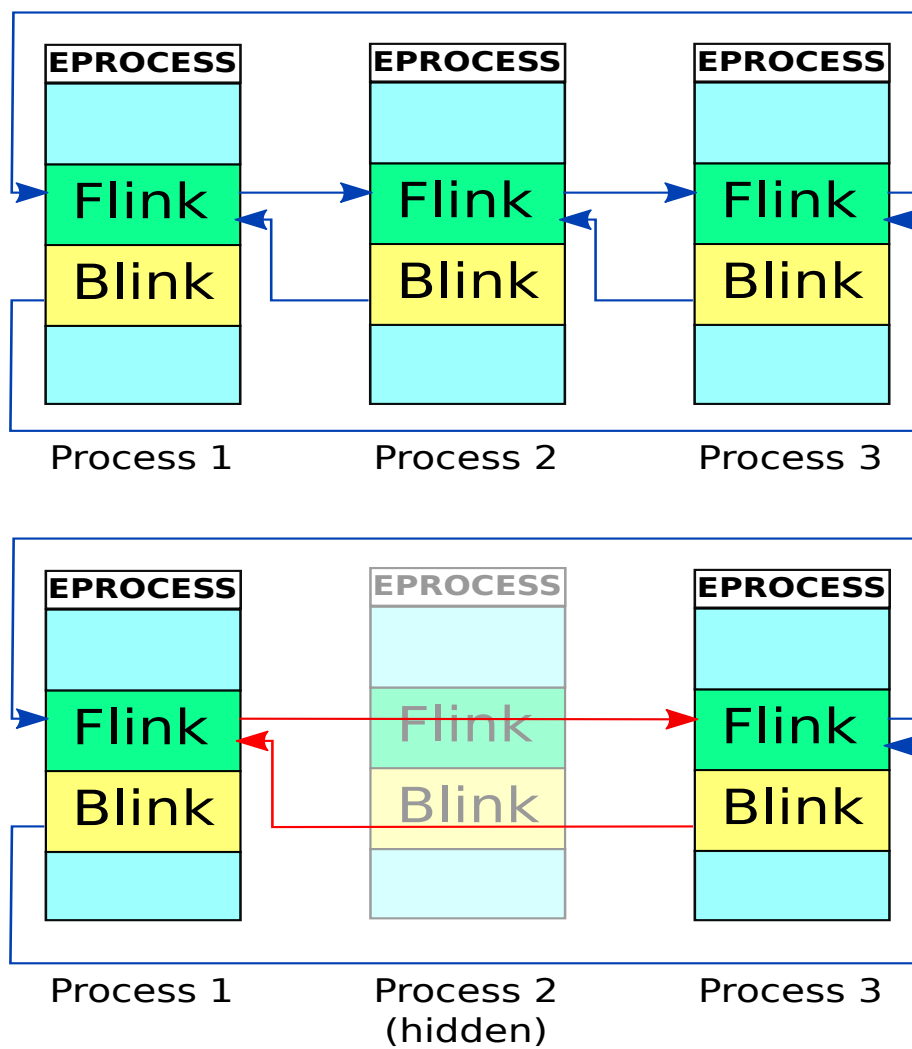


Figure 9: Process hiding using DKOM.

Process 1 FLINK and Process 3 BLINK fields are modified, effectively hiding Process 2.

4.3 Common rootkit uses

Even if a rootkit can work independently from any other software, it often operates in conjunction with other user-mode components.

4.3.1 Rootkit and malware relationship

In many case rootkits are used to cloak other malicious applications. This possibility allows malware writers to develop their applications without worrying about hiding their presence, since this facility is provided by the installed rootkit. In this scenario a rootkit can be used to protect malicious applications not only by hiding their presence, but also interfering with detection software or with system updating mechanisms.

Another typical use case is when a rootkit is used as a stealth backdoor, allowing remote access to an infected system. In this case the rootkit can send collected sensitive information or install other software on demand.

4.3.2 Legitimate software using “rootkit-like” techniques

The same techniques used by rootkits are sometimes used by legitimate software. A classical example are Antivirus tools: they often hook several system functions to intercept filesystem accesses, process creations, and network activities.

Additionally, some legitimate applications can filter the use of critical APIs to implement self-defense techniques. For instance, some tools do not allow any other applications to terminate their execution.

CHAPTER 5

MEMORY DUMP ACQUISITION AND LOW-LEVEL ACCESS

In a typical usage scenario three main components are used in the analysis framework:

- A method to dump physical memory and the swap area. Even though any available method can be used, we have actively developed and used two of them.
- The Volatility Forensics Framework, with some additional plugins and modifications.
- *Blacksheep* software itself.

5.1 Dumping method used

5.1.1 Custom dumping software

We have developed a custom application to dump the content of the physical memory and the swap area (i.e. *pagefile.sys* file) of a running Windows XP machine. Our goal has been to create a software that could be used in a large network of computers without interfering with users' activities. The typical usage scenario is a set of similarly configured machines, where users log in when they start using a machine and log out when they finish. This scenario is common in many different situations, such as offices, libraries, and computer laboratories in schools.

This application consists of two different components: a kernel module and a user-mode system service, running automatically in background when the system starts. An installer

has been developed too, allowing easy and automatic installation and configuration for both components. The dumping tool has been tested on several physical and virtual machines running 32-bit version of Windows XP (Service Pack 3).

5.1.2 The user-mode service

The system service is responsible for dumping physical memory and, if present, the swap file (*pagefile.sys*). To get a handle associated to the physical memory and the *pagefile.sys* file, the kernel module is invoked because it is not possible for an user-mode process to access these two resources directly. Once proper handles have been obtained and opened, it maps them (using *MapViewOfFile* API) and copies them to a file.

The dumping process is performed without interfering with users' activities. For this reason, the dumping procedure is started only when an user log off and, if during the dumping procedure an user log on into the system, it is immediately stopped. Since rootkits usually start during the system boot process, they are still running even when no users are logged into the system, so we think that this policy is able to minimize the impact on the usability of a machine, while still allowing us to dump memory when a rootkit is running.

Communication with the kernel module is performed using the standard Windows IRP mechanism. The system service assigns a specific name to each created memory dumps, encoding the machine name and the creation time. It is also responsible for uploading them to a remote server using an encrypted connection.

5.1.3 The kernel module

The kernel dumping module is responsible for getting the handle of the physical memory and the *pagefile.sys* file. The handle of the physical memory is obtained by opening device `\\Device\\PhysicalMemory` using *ZwOpenSection* API. The procedure we use to get the handle of the *pagefile.sys* file is more complex, and its pseudocode is in 3 (Windows APIs used are in comments).

```

1  getSwapFileHandle(){
2      c_process = getCurrentProcess() //PsGetCurrentProcess
3      s_process = getSystemProcess() //PsInitialSystemProcess
4      all_handles = getAllHandles() //ZwQuerySystemInformation
5      for each(handle in all_handles){
6          p_process = getParentProcess(handle) //PsLookupProcessByProcessId
7          handle_type = getHandleType(handle) //ObReferenceObjectByHandle
8          if(    p_process == s_process and
9              handle_type == FILE and
10             handle.fileName == "pagefile.sys"){
11
12             dup_handle = duplicateHandle(handle, c_process) //ZwDuplicateObject
13             return (dup_handle)
14         }
15     }

```

Listing 3: Procedure used to get the handle of the swap file.

Notice that the *getCurrentProcess* API returns the retrieved handle to the currently running process, that, in practice, is the system service we have developed, since it is the only process that interacts with this kernel module. The handle is duplicated inside the caller process, before being returned to it.

5.1.4 QEMU virtual machine introspection

An alternate method we have used to dump physical memory is virtual machine introspection. We have used QEMU with KVM to run a Windows 7 SP1 32-bit virtual machine.

If it is not necessary to get a copy of the swap file, it is possible to simply use the following command:

```
pmemsave 0x0 <physical memory size> <dump file>
```

This command automatically suspends the virtual machine while memory is dumped.

Otherwise, to also get a copy of the swap file, the following procedure is necessary:

1. suspend the virtual machine
2. dump physical memory
3. mount the hard disk image used by the virtual machine (in read-only mode)
4. locate the disk partition used by the guest operating system
5. mount the disk partition used by the guest operating system (in read-only mode)
6. copy *pagefile.sys* file
7. unmount the partition and the the hard disk image
8. resume the virtual machine

5.2 Volatility integration

We use Volatility (version 2.0) to retrieve crucial information that *Blacksheep* needs to perform its analyses. Volatility takes as an input a memory dump (current version does

not handle the swap file) and command line parameters (e.g. the analysis to be performed, the operating system version from which the memory dump has been acquired, parameters specific to the analysis to be performed) and returns a report (written in plain text). The Volatility version we use is compatible with any Windows versions and Service Pack from Windows XP Service Pack 2 to Windows 7 Service Pack 1, but no 64-bit version is supported yet.

We have developed additional plugins for Volatility and modified some of the existing ones to get all the information we need. Additionally, we have also used the *malware* analysis module (24).

Volatility implements an internal cache. For instance, locations of several data structures are saved once they are located. In this way any further analyses requiring to retrieve the same data structures can be speed up.

5.2.1 Volatility plugins

5.2.2 mappingandswap

This plugin returns mapping information needed to translate virtual addresses to physical ones for kernel address space. As explained in Section 3.2.3, it is possible to consider as unique the mapping between virtual and physical addresses. This plugin analyzes paging tables of several processes, in particular, mapping information is extracted from PTE and PDE records.

The output is a list of tuples in the following format:

```
<virtual address> <physical address> <type>
```

For simplicity, all addresses are always returned with 0x1000 granularity, so, even if large pages are used, they are split in several tuples.

The *type* field can have one of the following values: *VALID*, *TRANSITION* or *SWAP*. *VALID* is used when the memory at such virtual address is stored in the physical memory dump, *TRANSITION* when it is still stored in the physical memory dump, but it is in *TRANSITION* state, whereas *SWAP* is returned when it is stored in the swap file. In the first two cases, the returned *physical address* is an offset inside the physical memory dump, in the third it is an offset inside the swap file.

The plugin receives as a command line parameter the number of processes for which paging tables are analyzed. We found that some processes do not completely map the kernel address space. For instance, the *System* process does not map the *win32k.sys* kernel module, since this module is loaded to session space and the process *System* is created before any session is opened. For this reason, we use the policy of returning the union of all mapping information obtained analyzing the first five processes (in creation time order). We have found that this policy is a good trade-off between mapping completeness and processing time. In case that inconsistencies are found between mappings obtained from different processes, only the first acquired mapping is considered as valid. However, the number of such inconsistencies is negligible.

We have found that most of the inconsistencies are due to pages in transition state. We think that this is due to the fact that such pages are going to be moved to swap file. For this reason we have decided to use page in transition state, only if their mapping is present and consistent in all the analyzed processes.

5.2.3 modscan

This plugin is embedded in Volatility. It returns a list of all found kernel modules in a memory dump. We have slightly modified it to also return the CRC field of the found kernel modules. It works by scanning all memory and looking for *LDR_DATA_TABLE_ENTRY* data structures. This data structure is allocated by Windows when a kernel module is loaded.

The found kernel modules are returned as a tuple:

<physical address of the LDR_DATA_TABLE_ENTRY structure>

<module full name> <module virtual base address>

<module size> <module name> <module crc>

5.2.4 idt

This plugin is embedded in the *malware* module. It works by parsing the Kernel Processor Control Region (KPCR) structure. This is a crucial Windows kernel data structure, containing information about the state of a processor (so, for each available processor a different KPCR is present). It returns function pointers handling each interrupt that can be raised.

5.2.5 driverirp

This plugin is embedded in the *malware* module. It works by scanning all memory looking for kernel modules and returning the associated IRP handlers for each of them. Such handlers are invoked when a module receives a specific IRP. For each IRP handler, it also returns the pointed-to kernel module. Additionally, disassembling the pointed-to code, it tries to understand if the pointed location has been hooked by any other module.

5.2.6 ssdt

This plugin is embedded in Volatility, it returns a list of all system calls detected. It looks for pointed SSDTs in all threads, so it is able to find system calls in non canonical SSDTs as well. For each found system call, it returns its name, the pointed virtual address, and the kernel module where such address is located.

5.2.7 gdt

This plugin is embedded in the *malware* module. It works by parsing the KPCR structure to locate the GDT location. It returns information retrieved from the GDT, such as memory segment limits, memory segment attributes, and call gates (if present).

5.2.8 sysenter

We have created this plugin to retrieve the virtual address where the execution jumps when a *SYSENTER* instruction is executed. In a running machine, this value is stored in the MSR register 0x176, so it is not necessarily present in memory. In a live system this value can be read with the *RDMSR* assembler instruction (that must be executed by

kernel-mode code). Using a dumping driver or virtual machine introspection, it is possible to retrieve this value. However, it could be impossible with other methods.

We have found a workaround to this problem that we have used in the development of this plugin. We noticed that a copy of this value is always present in memory and used by the *KiTrap01* kernel function, so we locate this function (it is always set as a handler for the Interrupt 1) and then we look for the following assembler instructions:

```
mov     ecx, [ebp+68h]

cmp     ecx, <copy of MSR 0x176 register>
```

to locate the copy of the MSR 0x176 value.

We have successfully tested this approach with Windows XP Service Pack 3 machines both non-infected and infected by the *Rustock.b* rootkit (this rootkit modifies the MSR 0x176 value to hook all system calls). In both cases we have been able to retrieve the correct value. However, it is likely that this approach only works with specific Windows versions.

5.3 Blacksheep overview

Blacksheep consists of approximately 4700 lines of code and 270 lines of comments. It is written in Python 2.6 and it has been tested on an Ubuntu 10.4 machine (even though the code is substantially platform independent).

We will now describe briefly main classes and files that compose *Blacksheep*:

5.3.1 MemDump

This class represents a memory dump. Its main aim is to provide transparent methods to access memory locations at specific virtual addresses.

5.3.2 MemDumpFileAnalyzer

This class is used to perform preliminary analyses of a memory dump and it is instantiated during the initialization of a **MemDump** object. It also manages the cache of Volatility results and invokes Volatility if necessary. The values returned by methods of this class are used to fill a **MemDump** object.

5.3.3 Utils

This file collects several utility functions such as the **MemDump** objects initialization method, functions managing folders and files, and mathematical functions.

5.3.4 CodeAn

This file collects all functions related to the analysis of the code of a set of memory dumps.

5.3.5 DataAn

This file collects all functions related to the analysis of the data of a set of memory dumps.

5.3.6 RootAn

This file collects all functions related to the analysis of the kernel entry points of a set of memory dumps.

5.3.7 DiffMatrix

This class represents a distance matrix. Functions to perform and to show results of untrained analyses are methods of this class.

5.3.8 Driver

This class represents a kernel module.

5.3.9 Pointer

This class represents a generic pointer within the kernel address space. For example, kernel entry points are stored as pointers.

5.3.10 FVirtualPage

This class contains the data related to a virtual page (always with 0x1000 granularity). It can be filled with data read from a memory dump during the initialization of a **MemDump** object or on demand.

5.3.11 DataDescr

This class describes a DWORD (a 32-bit unsigned integer) inside analyzed data. A DWORD can have different types such as number, string, pointer to a data structure.

5.3.12 InvariantComparer

This class compares two dictionaries of **DataDescr** objects, enumerating differences among them.

5.3.13 Logger

This class implements a static method to log text messages generated by *Blacksheep*.

A message is logged to a file, standard output or both according to its priority.

5.3.14 PickleCache

This class implements a manager for partial results. Many partial results are saved to be used in further analyses without recomputing them. Partial results (encapsulated in objects) are saved to a file and loaded when necessary, using the CPickle¹ library.

5.4 Low level access layer

The aim of the low level access layer of *Blacksheep* is to initialize **MemDump** instances, allowing fast and transparent access to memory at any virtual address within kernel space. After its initialization, a **MemDump** object contains data from a memory dump. The initialization procedure and the internal representation have been developed with two main goals:

1. Provide methods to quickly retrieve data at a given kernel virtual address. On a running system, address translation is performed in hardware by processor MMU and TLB, but it is not possible to use such processor features in off-line processing of memory dumps. In addition, during an analysis translation functions are called thousands of times. For these reasons, particular attention has been given to performance while developing address translation functions.

¹docs.python.org/library/pickle.html

2. Allow different initialization procedures depending on which data are needed during the analysis to be performed. For instance, it is not necessary to retrieve the list of kernel entry points when only code is analyzed.

5.4.1 Parsing Volatility reports

Since the execution of Volatility commands can be time consuming (especially for the *mappingandswap* and *modscan* plugins), their results are saved in a folder used as a cache. The method *exec_command* in **MemDumpFileAnalyzer** class manages this cache. It receives a command to be executed by Volatility, if the result of such command is already available it is retrieved from the cache. On the contrary, if the command has been never executed before, it calls Volatility with the command, saves its result into the cache and returns it to the caller.

5.4.2 Initialization of a MemDump object

The initialization of a **MemDump** object is performed by the function *Utils.initMemdump*. The UML sequence diagram of this function is shown in Figure 10. During the initialization of a **MemDump** instance a corresponding **MemdumpFileAnalyzer** object is created. Its main tasks are to invoke Volatility (or retrieve results from the cache) and to parse its responses. Three sub-procedures are optional and they are performed only if specified (as function parameters) when calling *Utils.initMemdump*.

5.4.3 *fillInternalRepresentation*

If a lot of data needs to be read from a memory dump (for instance, during a data analysis), all mapped kernel memory is read from RAM and swap file during the initialization of the corresponding **MemDump** object. Usually, hundreds of megabytes of data need to be read from the RAM file. On the contrary, except in heavily loaded systems, only a few megabytes of kernel memory are stored in the swap file.

Retrieving such data from the RAM file needs sparse accesses, slowing the acquisition procedure. For this reason, since a copy procedure is performed using sequential reads, it is possible to increase performance by temporarily copying the RAM file to the `/dev/shm/` virtual device¹ and accessing this copy instead of the original. A circular buffer is implemented inside `/dev/shm/` virtual device: when this device is full, a previously copied RAM file is deleted.

Notice that the internal representation is always created, even when it is not filled. In such case, data will be read from RAM or swap file only when necessary (see Section 5.4.6)

5.4.4 *getKernelEntryPoints*

Kernel entry points are retrieved only if necessary, typically during kernel entry point analysis.

¹Linux 2.6 kernel creates by default a RAM disk mapped to `/dev/shm/` folder. It is a virtual device which content is stored in system memory and can be accessed by any application.

5.4.5 *findAllKernelPollAllocations*

If data analysis needs to be performed, kernel pool allocations are searched in the memory dump during its initialization. We describe how kernel pool allocations are detected in Section 7.1.

During the initialization procedure all kernel modules loaded in a memory dump are retrieved and their PE structure is parsed and stored.

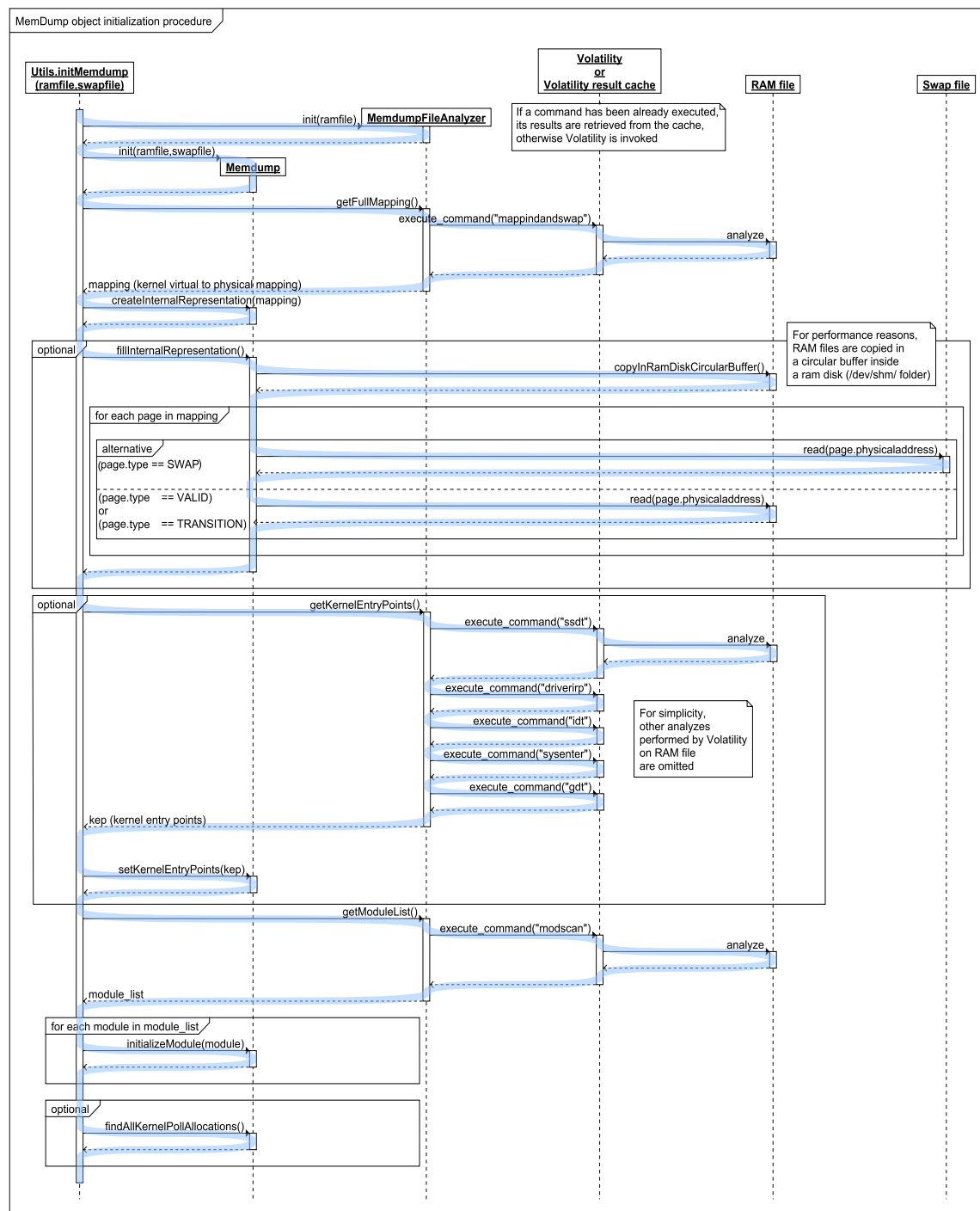


Figure 10: UML sequence diagram of the initialization of a MemDump instance.

5.4.6 Internal representation

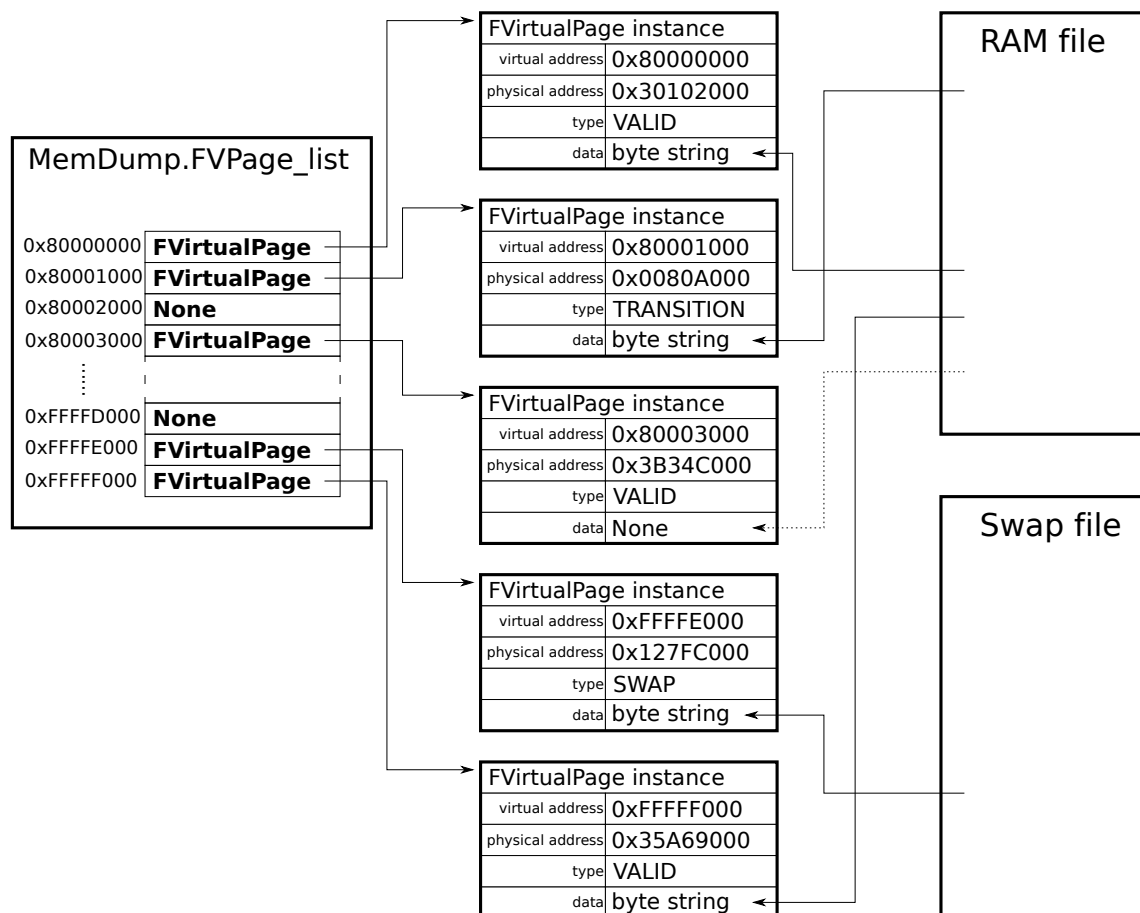


Figure 11: An example of the internal representation of a memory dump.

Kernel memory is stored in a **MemDump** object as a tuple of 0x80000 elements¹ (called *FVPage_list*). Each element corresponds to a 0x1000 byte long page inside the kernel memory. For instance the third element corresponds to virtual addresses between 0x80003000 and 0x80003FFF, because:

$$0x80003000 = 0x80000000 + (0x1000 * 3)$$

$$0x80003FFF = 0x80003000 + 0x1000 - 1$$

Three different situations are possible (refer to Figure 11):

- A *FVPage_list* element is None (0x80002000 and 0xFFFFD000 pages in the example).

In this case, the corresponding memory page is unmapped.

- A *FVPage_list* element is a **FVirtualPage** instance which *data* field is a binary string (0x80000000, 0x80001000, 0xFFFFE000, 0xFFFFF000 pages in the example).

In this case the corresponding memory page is mapped and its content is stored in RAM file (if its type is VALID or TRANSITION) or swap file (if its type is SWAP).

The content of the memory at such location has been already read from RAM/swap file, so no access to the disk is necessary when accessing data within these memory pages.

¹In this chapter, unless otherwise specified, we will always assume that */3GB* split is disabled.

- A *FVPage_list* element is a **FVirtualPage** instance which *data* field is None (0x80003000 page in the example).

As in the previous case the corresponding memory page is mapped to RAM or swap file. However, the content of the memory at such location has been never read from RAM/swap file, so an access to disk is necessary when accessing these data. Once data have been retrieved from disk, they are stored in the *data* field of the **FVirtualPage** instance, allowing faster future access.

This internal representation allows fast access to memory at any kernel virtual address. Additionally, it has enough flexibility to store only needed data during the initialization phase, retrieving the content of other pages of memory only when necessary.

In 4, the pseudocode of the method to access a DWORD of memory at a given virtual address is shown. The implemented access methods are:

- *getDwordAtVirtualAddress(vaddr)*. It returns DWORD at *vaddr* virtual address.
- *getRangeOfMemory(start,size)*. It returns memory region between virtual addresses *start* and *start + size*.
- *getDriverRVA(driver,rva)*. It returns DWORD at virtual address *driver.base_address+rva*.

If any of the required bytes are unmapped, None is returned.

```

1  getDwordAtVirtualAddress(vaddr){
2      //kernel_space_start_address = 0x80000000
3      fvpage = self.FVPage_list[(vaddr - kernel_space_start_address)>>12]
4      if(fvpage == None){
5          return None //this location is unmapped
6      }else{
7          if(fvpage.data == None){
8              //this location has been never read from disk
9              fill(fvpage)
10         }
11     }
12     start_offset = vaddr & 0xfff
13     return (fvpage.data[start_offset:start_offset+4])
14 }
15
16 fill(fvpage){
17     physical_page_start = fvpage.physical_address & 0xFFFFF000
18     if(fvpage.type == VALID or fvpage.type == TRANSITION){
19         fvpage.data = read_page_from_ram_file(physical_page_start)
20     }else if(fvpage.type == SWAP){
21         fvpage.data = read_page_from_swap_file(physical_page_start)
22     }
23 }

```

Listing 4: Procedure used to get a DWORD at a specified virtual address.

CHAPTER 6

HIGH-LEVEL ANALYSES

Blacksheep performs three different types of comparisons between any two memory dumps:

- Code comparison
- Kernel entry point comparison
- Data comparison

The results of such comparisons are then used to perform trained analyses and untrained analyses. So, in total, six different analyses are performed.

Trained analysis works on two sets of memory dumps: a training set, assumed to be non-infected, and a set of memory dumps of acquired from machines to be checked. Each memory dump to be checked is compared against the training set, looking for modifications that are likely to have been generated by a rootkit infestation. The result of each trained analysis is a report stating if the checked memory dump is likely to be infected.

Untrained analysis works on a single set of memory dumps. Results returned from code, data and kernel entry point comparisons are used to compute the distance between each pair of memory dumps. Then, hierarchical clustering (using SciPy library (40)) is performed on the computed distance matrices. Untrained analyses assume that most of

the analyzed memory dumps have been acquired from non-infected machines. The result of each untrained analysis is a hierarchy of clusters.

6.1 Code analyses

6.1.1 Code comparison

Code comparison examines code of two instances of the same kernel module in two different memory dumps. The way in which kernel modules are identified is described in Section 7.3, whereas the method by which code inside a module is identified is detailed in section 7.4.

The comparison proceeds by performing a byte-per-byte comparison of the code between the two memory dumps. Some differences are considered as *benign* (i.e. they are not caused by any malicious modification of the code). For instance, benign differences can exist due to relocation, and many pointers have different values among the training set. For this reason, the analysis verifies if the differences found are caused by any of the well-known modifications that can occur when a module is loaded in memory.

Input parameters of the code comparison function are:

- The two memory dumps to be compared.
- The two kernel module instances to be compared, inside the given memory dumps.
- (*optional*) A list of memory pages to be checked. If this parameter is set, only the given memory pages are compared.

As an output, this function returns:

- The number of differences found, classified according to their type.
- A set of memory pages in which *non-benign* differences have been found.
- A set of RVAs where *non-benign* differences have been found.
- A boolean flag indicating whether results of the comparison are reliable.

6.1.2 Benign difference types

In the following paragraphs we will detail the types of differences we consider as *benign* and how we detect them. If a difference is not classified as any of these types, it is considered to be *non-benign*, since it is likely that it has been caused by a rootkit infection. The next paragraphs are ordered in the same order that we check for explanations of a found difference. For performance reasons, the chosen order is from the most common explanation to the rarest one.

Relocation

Kernel modules are not loaded in memory at a fixed address and their code is not position independent. For these reasons, when a module is loaded, the loader applies relocations: it patches pointers inside the module code to be consistent with the base address which the module has been loaded at.

Where and how to apply these relocations is specified by the compiler inside a well-defined section of the module file (36), usually named *.reloc*. We do not rely on this PE section to detect relocations, but we simply verify if the difference found matches with the difference of the base addresses that the compared module instances have been loaded at.

Imported and exported addresses

Inside a kernel module, many pointers to other modules are present. Because of module relocation, these pointers have different values in different dumps. For this reason, when a difference is found between two bytes, we treat the surrounding DWORDs as pointers and we check if they are equal. Two pointers are considered to be equal if they point to the same RVA (Relative Virtual Address)¹ within the same kernel module.

PE header modifications

Modifications inside the PE header of a kernel module are suspicious, since malformed PE headers are often used to confuse analysis tools (41). However, during the development of *Blacksheep* we have observed that sometimes Windows loader modifies some specific fields when a kernel module is loaded from the disk to the memory.

For this reason, we ignore modifications that occur in the following fields:

- ImageBaseAddress
- PointerToRelocations (for each PE section)
- NumberOfRelocations (for each PE section)

Hooks

The use of kernel hooking is described in section 4.2.3. During code comparison we need

¹In Windows terminology, an RVA is a virtual address inside a loaded module. Its value is equal to the distance between the pointed location and the base address of the module that the location lies within.

to be able to detect if a hook is *equal* between two dumps (i.e. it makes the execution divert to the same function). In the case of hooks pointing to other loaded kernel modules, we simply consider them as *equal* if they point to the same RVA within the same module. (This detection is already performed while dealing with imported and exported addresses).

However, there are more complicated situations to deal with, such as when a hook is created by making a function pointer point to code written inside dynamically allocated memory. To deal with such cases, found differences are checked to see if they are caused by one of the following hooking techniques:

- Changing a code pointer used by an indirect **call** or an indirect **jmp** instruction.
- Modifying the argument of a direct **call** or **jmp** instruction.
- Modifying the argument of an instruction **push**, followed by a **ret**.

In these three situations, the addresses pointed by the hooks in the two dumps are extracted and it is checked if, at such addresses, the same function is present. In this case, the two hooks are considered as *equal* and the difference found is classified as *benign*. How the code of two pointed functions is compared is described in Section 7.6.

6.1.3 Special cases

There are some special cases to deal with during code analysis. **Unmapped pages**
When a kernel module (and any PE file in general) is loaded in memory by Windows, its content is read from disk and loaded into physical memory only partially (see 3.2.2). For

this reason, while comparing two module instances, we compare only memory pages that are mapped in both dumps.

Wrongly mapped pages

Since the memory dumping process can lead to partially inconsistent dumps, it is possible that some pages inside an analyzed kernel module are incorrectly mapped in the memory dump. These incorrectly mapped pages can cause *Blacksheep* to compare wrong locations during code comparison procedure. For this reason, we assume that, when more than 60% of a memory page is different between two compared module instances, such differences are caused by an error in the extracted mapping information and they are ignored.

Unreliable comparison

If too many wrongly mapped pages are detected, the comparison is stopped and the compared module instance is ignored in further analyses.

6.1.4 Unstable pages

During the development of *blacksheep* we have noticed that, even in non-infected machines, some kernel modules have locations inside their code that change among different memory dumps. We have detected this in:

- Some locations inside **ntoskrnl.exe** and **peauth.sys** modules.
- Kernel modules used by security software (i.e. antivirus, personal firewall).
- Kernel modules used by Digital Right Management software.

There are different reasons why kernel modules change their own code:

- Dynamically generated code.
- Code obfuscation used to prevent reverse engineering.
- Code obfuscation used by security software self-protection mechanisms.

To deal with such code locations, we create a list of memory pages in which the code could change, even in non-infected systems. During the analysis, if differences are found inside these pages, they are not considered to be caused by an infection. However, a non-critical message is reported to allow user to conduct further investigations.

6.1.5 Trained analysis

Trained code analysis examines each kernel module in the dump to be checked. It works in three different phases.

First, *Blacksheep* verifies that each module within the dump it is processing also exists in at least one dump of the training set. If it does not exist in any training dump, it is considered a critical anomaly, since it is likely that this new kernel module has been loaded by a rootkit.

Then, each module is compared against the corresponding module in every dump in the training set, searching for a corresponding module in the training set with no *non-benign* differences with it. If such a module is found, the examined module is considered as non-infected and the analysis goes on comparing the next module.

However, if no such module within the training set is identified, we enumerate the memory pages in which *non-benign* differences have been found and check if these pages

are stable among the training set (see 6.1.4). If they are unstable even in the training set, this means that differences found are not caused by a malicious infection. Otherwise the module is considered as infected. A memory dump is considered to be non-infected if all of its kernel modules are non-infected (and present in at least one dump in the training set). Otherwise, it is considered as infected and a critical message is reported to the user.

The result of this analysis is a report (in the form of a text file). Additionally, module instances classified as infected and modules not found in the training set are extracted from the analyzed memory dump for further investigations.

6.1.6 Untrained analysis

Untrained analysis works by using the number of *non-benign* differences found between two given dumps as a distance metric. Differences whose distance is less than four bytes are merged and counted as one, because it is likely that they have a common cause. For instance, when a pointer is hooked, usually two or three adjacent bytes are modified. During this analysis, only kernel modules that are present in every memory dump are taken into consideration since, without having a training set, it is not possible to have a list of trusted modules.

Code comparison is performed on each possible pair of analyzed memory dumps. Since it is symmetric, the number of necessary comparisons is halved. The linkage function used during clustering is *single*: the distance between two non-singleton clusters is equal to the minimum distance between elements of each cluster. This policy allows us to ignore

differences caused by wrongly mapped pages (since they are likely to be present only in a single memory dump within a cluster).

The output of untrained code analysis is a hierarchy of clusters with analyzed memory dumps as leaves.

6.2 Kernel entry point analyses

6.2.1 Kernel entry point comparison

Kernel entry point comparison examines differences in function pointers used when the execution switches to kernel mode. The comparison proceeds by collecting all kernel entry points of the two examined memory dumps and then comparing them.

The input parameters of the kernel entry point comparison function are the two memory dumps to be compared. As an output it returns differences found, classified according to their type.

Kernel entry points are collected using different Volatility plugins: **ssdt**, **idt**, **driverirp**, **gdt**, **sysenter**. These values are internally stored as pointers and the following tuple is saved for each of them: (*<ID>*, *<Value>*). *ID* is a unique string used to identify a kernel entry point, for instance the *ID* of a SSDT entry is *SSDT:<function_number>* and the *ID* of an IRP is *IRP:<module_name>:<IRP_name>*. *Value* is the location where it is pointing at. All kernel entry points from a memory dump are compared with those with the same *ID* of another memory dump. Found differences are classified in one of the categories listed below.

6.2.2 Classification categories

In this paragraph we will refer to kernel entry points as pointers, the memory dump to be verified will be called *m_verify*. The dump in the training set which *m_verify* is compared to is called *m_reference*.

added inside

The pointer in *m_verify* is not present in *m_reference* and it points inside a kernel module.

added outside

The pointer in *m_verify* is not present in *m_reference* and it points outside any kernel module.

changed from inside to inside

The pointer in *m_verify* points to a different location than in *m_reference*. In both memory dumps it is pointing inside a kernel module.

changed from inside to a new module

The pointer in *m_verify* points to a different location than in *m_reference*. In both memory dumps it points inside a kernel module, but the module pointed in *m_verify* is different from that pointed in *m_reference*.

changed from inside to outside

The pointer in *m_verify* points to different locations than in *m_reference*. In *m_reference* it points to a kernel module, whereas in *m_verify* it points to a location outside any kernel module.

changed to generic IDT handler

The pointer in *m_verify* points to the generic interrupt handler used by Windows to manage unexpected interrupts.

changed from outside outside - equal

The pointer in *m_verify* points to the same location than in *m_reference*. In both memory dumps it points outside any kernel module.

changed from outside to outside - different

The pointer in *m_verify* points to a different location than in *m_reference*. In both memory dumps it points outside any kernel module.

Any difference classified as **“added outside”**, **“changed from inside to outside”**, **“changed from outside to outside - different”**, **“changed from inside to a new module”** or **“changed from inside to inside”** is considered as *suspicious* since it is likely that it has been caused by a rootkit infecting *m_verify*.

6.2.3 Trained analysis

Trained analysis searches, for each dump to be verified, if it exists a memory dump with no *suspicious* difference. If such a memory dump is not found, the analyzed one is classified as infected.

The result of this analysis is a report (in the form of a text file). For each analyzed memory dump it is reported whether it has been classified as infected or as non-infected.

Additionally, it provides details about modified kernel entry points detected in memory dumps classified as infected.

6.2.4 Untrained analysis

Untrained analysis works by using as a distance metric the following formula:

$$distance = \max(comparison(dump1, dump2), comparison(dump2, dump1))$$

Where *comparison(dump1, dump2)* returns the number of *suspicious* differences found, using *dump1* as *m_verify* and *dump2* as *m_reference*. Linkage function used during clustering is *single*: the distance between two non-singleton clusters is equal to the minimum distance between elements of each cluster.

The output of untrained data analysis is a hierarchy of clusters with analyzed memory dumps as leaves.

6.3 Data analyses

6.3.1 Data comparison

Data comparison works in three different phases. First of all, kernel pool allocations and double linked lists are searched within kernel memory stored in a memory dump (these steps are described in Sections 7.1 and 7.2). Then a description is given to each DWORD allocated in data sections of all kernel modules (how data sections are identified is described in Section 7.4). This step is performed by **MemDump.DataToDescr** method. Finally, two dictionaries of DWORD descriptions relative to two different memory dumps are compared by **InvariantComparer.computeDiffs** method.

We will now details how these functions work and how results are used by trained and untrained analyses.

6.3.2 Memdump.DataToDescr method

This method takes as an input a DWORD and it returns its description as a **DataDescr** object. Using this method **DataDescr** objects describing each DWORD inside any data section of all kernel modules loaded in a memory dump are created and collected in the **DataDescriptions** dictionary. This dictionary uses as values **DataDescr** objects and as keys the locations (encoded as *<kernel module>:<RVA>*) of the corresponding described DWORDs. Even double linked lists found previously are stored in this dictionary, in such a case list heads are used as keys and a **DataDescr** objects of type *DLLIST* are used as values.

A **DataDescr** object can have one of the following types, according to how the described DWORD is classified by **Memdump.DataToDescr** method.

ZERO DWORD value is 0x0 (usually it is a NULL pointer or an unused memory region)

NUMERIC The DWORD has a numeric value. We heuristically assume a value as numeric if and only if: $\text{value} < 0x80000000$ (*kernel_space_start_address*) or $\text{value} > 0xFFD00000$.

STRING The DWORD is a (part of a) string. We heuristically assume a value as a string if it contains only printable characters eventually interleaved with 0x00 byte (for Unicode).

PSTRING The DWORD is a pointer to a string.

POOL_HEADER The DWORD is the first 4 bytes of a kernel pool allocation header (see Section 3.4.2).

POOL_TAG The DWORD is a kernel pool allocation tag (see Section 3.4.2).

FLINK and BLINK The DWORD is the *FLINK/BLINK* field of a double linked list data structure (see Section 3.4.1).

POINTER The DWORD is a pointer to a RVA within a kernel module.

POINTER_TO_DLLIST The DWORD is a pointer to an element of a double linked list.

ANY This type is used when we are unable to find any suitable description for a given DWORD

The three following types are called *structured* because **DataDescr** objects of these types recursively contain (in their *data* field) references to other **DataDescr** objects. The maximum depth at which we limit our analysis is 3. How pointers to data are detected is described in Section 7.5.

POINTER_H The DWORD is a pointer to a pool allocation. In the *data* field a list of **DataDescr** objects describing each DWORD of the pointed data structure is stored. The starting address of the pointed data structure and its size are inferred analyzing the *POOL_HEADER* data structure.

POINTER_S The DWORD is a pointer to allocated data within kernel space, but we are unable to find a surrounding allocation. In the *data* field a list of **DataDescr** objects describing the pointed DWORD and the surrounding ones is stored. How many DWORDs are analyzed around the pointed one can be set with specific settings (by default 10 DWORDs are analyzed).

DLLIST **DataDescr** objects of this type are created while kernel memory is scanned looking for double linked lists (see 7.2). In their *data* field a **DataDescr** object describing all elements of the list is stored. It is created by “merging” descriptions of the single elements using **DataDescr.compare** method.

POINTER_V When a pointer to data is found but the reached depth is already the maximum allowed, a **DataDescr** object of this type is created.

The field *value* of a **DataDescr** object has different meanings depending on its type. For instance, for *NUMERIC* type, it is the actual value the DWORD has, for *PSTRING* it is the pointed string, for *POINTER* it is the tuple (*<pointed kernel module>*, *<RVA>*). In **DataDescr** objects of type *POINTER_TO_DLLIST* the field *value* is the head of the pointed double linked list (see 7.2)

Listing 5 shows how a detected double linked list is internally represented by *Blacksheep* (many lines are skipped). The found list is composed by 5 elements, all of *KTHREAD* type. Two important properties that are invariant on all the elements of this list are identified:

- All the elements point to a process inside the system list of running processes (line 11).
- All the elements point to the same *SERVICE_DESCRIPTOR_TABLE* (line 18).

See Windows WDK and NirSoft website (42) for the definitions of some kernel data structures.

```

1  DLLIST head: 0x8055b1e0L id: .DLLIST:( 'ntoskrnl.exe', 541152L, '.data') nelements: 5
2  POINTER_H      size: 0x9dL
3      -0x8  POOLHEADER 0xa4f004f - PType:0x5L - BSize:0x4f - PIndex:0x0 - PSize:0x4f
4      -0x4  POOLTAG Thre 0x65726854L //pool tag suggests that this is a KTHREAD data structure
5      0x0   VALUE 0x1
6      0x4   ZERO  0x0
7      0x8   POINTER_TO_DLLIST .DLLIST:ObjT_24_440_120
8      0xc   VALUE 0x22000000
9      0x10  VALUE 0x1
10 // ...
11      0x5c  POINTER_TO_DLLIST .DLLIST:( 'ntoskrnl.exe', 537176L, '.data')
12 //.DLLIST:( 'ntoskrnl.exe', 537176L, '.data') is the head of the system KPROCESS list
13      0x60  ZERO  0x0
14      0x64  GVALUE
15 // ...
16      0xf4  ZERO  0x0
17      0xf8  POINTERS  0x80553060L size: 0xb
18 //pointer to KeServiceDescriptorTableShadow (SERVICE_DESCRIPTOR_TABLE)
19      -0x8  ZERO  0x0
20      -0x4  ZERO  0x0
21 //SERVICE_DESCRIPTOR_ENTRY #1 (SST)
22      0x0   pointer  ( 'ntoskrnl.exe', 175036L, '.text') //KiServiceTable (SSDT)
23      0x4   ZERO  0x0
24      0x8   VALUE 0x11c
25      0xc   POINTER  ( 'ntoskrnl.exe', 176176L, '.text') //KiArgumentTable
26 //SERVICE_DESCRIPTOR_ENTRY #2 (SST)
27      0x10  POINTER  ( 'win32k.sys', 1690496L, '.data') //W32pServiceTable (SSDT)
28      0x14  ZERO  0x0
29      0x18  GVALUE
30      0x1c  POINTER  ( 'win32k.sys', 1693840L, '.data') //W32pArgumentTable
31 // ...
32      0x130  FLINK
33      0x134  BLINK
34 // ...

```

Listing 5: An example of a detected double linked list.

6.3.3 DataDescr.compare method

This method is used to get a common description of two different DWORDs. It takes as an input two **DataDescr** objects and it returns a new **DataDescr** object giving a description suitable for both. This method is invoked in two different scenarios:

- To get an unique **DataDescr** object describing all the elements of a double linked list.
- To compute a set of invariant properties that hold in the training set during trained analysis.

It works by finding the “less generic” description suitable for both the compared **DataDescr** objects. For instance, if they are both of type *NUMERIC*, but they have different values it returns a **DataDescr** object of type *GVALUE* (generic numeric value) or if they are both of type *POINTER* and they point to the same kernel module but to different RVAs a **DataDescr** object of type *POINTER* with value (*<kernel module>,GENERIC-RVA*) is returned. If no suitable description is found a **DataDescr** object of type *ANY* is returned.

Structured types are recursively compared starting from their leaves, using a depth-first approach.

6.3.4 InvariantComparer.computeDiffs method

This method compares two **DataDescriptions** dictionaries and returns their differences. It is used to compare each pair of **DataDescr** objects with the same key in both

dictionaries. If they have different types, such difference is classified in one of the following categories:

different_structure The two compared objects have both a *structured* type, but it is different.

no_structure One of the compared object has a *structured* type, but not the other.

zero_to_structure One of the compared object has a *structured* type, whereas the type of the other is *ZERO*.

zero_to_something One of the compared object has a not *structured* type, whereas the type of the other is *ZERO*.

different_pointer The two compared objects are both of type *POINTER*, but their values are different.

no_pointer One of the compared objects is of type *POINTER*, but not the other.

no_pointer_to_dlist One of the compared objects is of type *POINTER_TO_DL-LIST*, but not the other.

other_difference The two compared objects have different types and they do not fit in any of the categories previously mentioned.

These types of differences have been chosen because they are related to modifications that could be generated by a rootkit. This method returns the differences found, classified according to one of the categories listed above.

6.3.5 Trained analysis

Trained analysis works in three steps. First of all, for each memory dump in the training set, **DataDescriptions** dictionaries are created. Then they are compared, applying **DataDescr.compare** method to values with the same key. In this way an *invariance set* is created: a dictionary describing data properties that are *invariant* among all the memory dumps in the training set. Finally, **DataDescriptions** dictionaries of each dump to be verified are compared, using **InvariantComparer.computeDiffs** method, to the *invariance set* and found differences are logged.

Data trained analysis does not classify memory dumps in infected and non-infected, but it returns a list of data differences that can be manually analyzed. In particular for each difference found the following information are logged:

- The type of the difference.
- A textual representation of the two compared **DataDescr** objects.
- A textual representation of the two data structures in which the compared objects are located (if they are located within **DataDescr** objects of *structured* types).

Some differences are expected even when non-infected memory dumps are analyzed, however in infected memory dumps some peculiar differences may be found (see Sections 8.4.2 and 8.6.2).

6.3.6 Untrained analysis

Untrained analysis works by using as a distance metric the number of *zero_to_structure* differences found when comparing **DataDifferences** dictionaries of two memory dumps (those found inside double linked lists are not counted). This specific type of difference has been chosen, because it is the most likely to be caused by a rootkit infection.

Data comparison is performed for each possible pair of analyzed memory dumps. Since it is symmetric the number of necessary comparisons is halved. Linkage function used during clustering is *average* (also called UPGMA): the distance between two non-singleton clusters \mathcal{A} , \mathcal{B} is equal to: $\frac{1}{|\mathcal{A}| \cdot |\mathcal{B}|} \sum_{x \in \mathcal{A}} \sum_{y \in \mathcal{B}} d(x, y)$

The output of untrained data analysis is a hierarchy of clusters with analyzed memory dumps as leaves.

CHAPTER 7

TECHNICAL DETAILS

7.1 Kernel pool allocation detection

Kernel pool allocations are searched within a memory dump during the initialization of the corresponding **MemDump** object, by the procedure in Listing 6. This procedure scans kernel virtual addresses looking for a *POOL_HEADER* data structure (see Section 2). This data structure is detected using rules listed in (25). In addition to rules shown in Listing 6 we also exploit the fact that within the same memory page, only pool allocations of the same type (*p_{type}* variable in the code) can be present. This allows us to reduce the number of false matches. Virtual addresses between 0xC0000000 and 0xE1000000 are skipped, since, according to Windows kernel memory layout, no kernel pool allocation is stored in this range. This procedure is able to detect only allocations smaller than 0x1000 bytes, because bigger ones are stored using a different mechanism.

Detected allocations are saved within the **FVirtualPage** corresponding to the memory page where they have been found. This allows a fast implementation of the function that, given a kernel virtual address, returns the surrounding pool allocation (if present).

```

1  IsAPoolHeader(vaddr){
2  //check if at a given virtual address a pool header is present
3  //if a pool header is found, returns its size, otherwise returns None
4
5      dword = getDwordAtVirtualAddress(vaddr)
6      poffset = vaddr & 0xFFF
7      psize = 0x01FF & dword
8      bsize = (0x01FF0000 & dword) >> 16
9      ptype = (0xFE000000 & dword) >> (16 + 9)
10
11     //these rules are taken from:
12     //subs.emis.de/LNI/Proceedings/Proceedings97/GI-Proceedings-97-9.pdf
13     if( (bsize > 0 and bsize <= 511)
14         and ((vaddr >> 12) == (((vaddr + ((bsize - 1) * 8) - 1)) >> 12))
15         and (ptype in (2,3,4,5,6,7,8,33,34,35,36,37,38,39))
16         and (((psize == 0) and ((vaddr % 0x1000) == 0)) or psize > 0)
17         and ((psize * 8) <= 0x1000) and ((bsize * 8 + poffset) <= 0x1000)){
18
19         //check pool tag
20         pool_tag = getDwordAtVirtualAddress(vaddr + 4)
21         if((pool_tag & 0x00808080)!=0){
22             return None
23         }else{
24             invalid_characters = countNumberOfNonPrintableCharacters(pool_tag)
25             if(invalid_characters > 1 and pool_tag != 0x00000000){
26                 return None
27             }else{
28                 return (bsize * 8)
29             }
30         }
31
32     }else{
33         return None
34     }
35 }

```

Listing 6: Procedure used to detect *POOL_HEADER* data structures.

7.2 Double linked list detection

Double linked lists are found during the first step of every data comparison. This procedure scans kernel virtual addresses looking for *LIST_ENTRY* data structures correctly linked (see Listing 1 and Figure 7). As in kernel pool allocation detection, virtual addresses between 0xC0000000 and 0xE1000000 are skipped. While looking for double linked lists we check if the following conditions are true:

- *FLINK* pointers create a closed loop.
- *FLINK*s and *BLINK*s point coherently to the next/previous element.
- A list head exists.
- All the elements of the list are within a kernel pool allocation.

If the above conditions are met, found list is saved.

We identify a double linked list using its list head. To give to each found list a head unique and coherent among different memory dumps we use the following rules:

- If a list head is within a kernel module, it is identified using the tuple (*<module>*,*<RVA>*) describing the virtual address where the list head is.
- Otherwise, we use the tuple (*<tag_str>*,*<flink_offset>*,*<element_size>*,*<head_offset>*), where:

- *tag_str* is the poll tag of the elements of the list.
- *link_offset* is the offset the *FLINK* field has, within each element of the list.

- *element_size* is the size of each list element.
- *head_offset* is the offset the list head has, within the kernel pool allocation in which it is located.

7.3 Kernel module identification

Kernel modules are enumerated using a modified version of Volatility **modscan** plugin. We have chosen to identify any kernel module using the tuple (*<module size>*,*<CRC>*).

Module size is the size a module has when loaded in memory (it is always a multiple of 0x1000). *CRC* is a 20-bit value computed by the compiler when a module is created, used to verify its integrity. It is computed using a custom hashing function on the content of the module. We believe that the probability that two kernel modules have the tuple (*<module size>*,*<CRC>*) equal by chance is extremely low.

We do not use the module name, because we have discovered that the same kernel module can have different names. We have noticed this behavior in two different situations: kernel modules used during system hibernation process¹ and the kernel module used by SCSI Pass-Through Direct driver² (it changes its name at every reboot).

¹<http://msdn.microsoft.com/en-us/library/windows/hardware/ff564084%28v=vs.85%29.aspx>

²<http://www.duplexsecure.com/en/downloads>

A rootkit could easily modify its content to have the same *module size* and *CRC* of a legitimate kernel module. However, this would lead our analysis to compare the rootkit module with a legitimate one, detecting differences between the two.

7.4 Detecting code and data sections within a PE file

A PE file is divided in sections, typically some of them contains code, others data, others information used by the loader. Unfortunately, there is no a precise and universally used convention to understand if a section contains code or data. However, it is possible to use the name and the flags assigned to each section to decide with an almost perfect approximation if a section contains code.

Listing 7 shows the pseudocode of the procedure we use to understand if a given RVA within a kernel module is inside a code section. Rules used are similar to those used by System Virginty Verifier tool¹. It is worth to note that even if a PE section has a flag to indicate if it is executable (*IMAGE_SCN_MEM_EXECUTE*), its value is not considered by the loader. For instance, even using hardware where it is available, the *XD* bit is not used, allowing code to be executed in sections flagged as not executable too. Additionally, some specific rules are necessary for sections within *ntoskrnl.exe* and *hal.dll* kernel modules.

¹http://www.woodmann.com/collaborative/tools/index.php/System_Virginty_Verifier

```

1  isInsideCode(module,RVA){
2  //returns True if given RVA within module is in a code section
3  //returns False otherwise.
4
5      section = module.RVAtoSection(RVA)
6      name == section.getName()
7      notExec = not section.hasFlag(IMAGE_SCN_MEM_EXECUTE) //boolean
8
9      if(notExec and (name==" .data" or name==" .npdata" or name==" .bss")){
10         return False
11
12     }else if(notExec and (name.find("PAGE")>-1 and name!="PAGELK")){
13         return False
14
15     }else if(module.name=="ntoskrnl.exe"
16         and (notExec or name=="PAGEVRFY" or name=="PAGESPEC"
17         or (name=="INIT" and section.hasFlag(IMAGE_SCN_MEM_DISCARDABLE)))){
18         return False
19
20     }else if(module.name=="hal.dll"
21         and (notExec or (name=="INIT" and
22         section.hasFlag(IMAGE_SCN_MEM_DISCARDABLE)))){
23         return False
24
25     }else{
26         return True
27     }
28 }

```

Listing 7: Procedure used to decide if an address is inside a code section.

7.5 Data pointer detection

We use the following heuristics rules to determine if a given DWORD is a pointer within kernel memory:

1. $0x80000000$ (*kernel_space_start_address*) \leq (DWORD value) $\leq 0xFFD00000$
2. DWORD location is four-byte aligned
3. DWORD value is four-byte aligned

It is worth to note that, even if rules 2 and 3 are not strictly enforced by x86-compatible processors in 32-bit mode, they are always followed by compilers, since unaligned memory accesses have big performance penalties.

In several different situations *Blacksheep* needs to determine if two pointers are *equal*. We consider two pointers in two different memory dumps as *equal* if they point to the same RVA within the same kernel module.

7.6 Function code comparison

We have developed a procedure to understand if two functions inside two different memory dumps have the same code. The problem of understanding if two set of instructions have the same behavior is, in general, undecidable. This procedure simply deals with specific modifications that occur when code is relocated inside dynamically allocated memory regions. Still it is able to compare correctly complex hooks installed by some security tools.

Pseudocode is shown in 8. Instructions are decompiled using diStorm¹ Python library.

¹<http://code.google.com/p/distorm/>

```

1  compareFunctionCode(vaddr1,vaddr2,dump1,dump2){
2  //returns True if code of the functions at vaddr1 and vaddr2 is equal,
3  //returns False otherwise
4
5      LIMIT = 100 //an arbitrary limit
6      while(analyzed_instructions<LIMIT){
7          analyzed_instructions += 1
8          instr1 = getNextInstruction(vaddr1,dump1) //use diStorm3
9          instr2 = getNextInstruction(vaddr2,dump2) //use diStorm3
10         //instr.mnemonic is the assembler instruction name (e.g. MOV, JMP)
11         //instr.operand is the numeric value (e.g. 0xFF in MOV EAX, 0xFF)
12         if(instr1==instr2){ //their disassembled code is exactly the same
13             if(instr1.mnemonic=="RET"){ return True }else{ continue }
14         }else{
15             if(instr1.mnemonic!=instr2.mnemonic){
16                 return False
17             }else{ //checks if it is a direct pointers
18                 pointer1 = instr1.operand
19                 pointer2 = instr2.operand
20                 if(comparePointers(pointer1,pointer2,dump1,dump2)){
21                     continue
22                 }else{ //check if it is an indirect pointers
23                     pointer1_ind = dump1.getDwordAtVirtualAddress(pointer1)
24                     pointer2_ind = dump2.getDwordAtVirtualAddress(pointer2)
25                     if(comparePointers(pointer1_ind,pointer2_ind,dump1,dump2)){
26                         continue
27                     }else{
28                         return False
29                     }
30                 }
31             }
32         }
33     }
34     return False
35 }
36
37 comparePointers(pointer1,pointer2,dump1,dump2){
38 //Returns true if and only if pointer1 and pointer2 point to the same RVA
39 //within the same kernel module in dump1 and dump2 respectively.
40 }

```

Listing 8: Procedure used to compare the code of two functions.

7.7 Performance optimizations

7.7.1 Python optimizations

The code is run using Psyco library (43), a Python extension module working as a just-in-time compiler. It is enabled by inserting the following line at the beginning of the `--main--` function:

```
psyco.profile(0.01)
```

7.7.2 Caching

The use of `/dev/shm/` device and the Volatility result cache have been already explained in Section 5.4.1 and Section 5.4.2.

In addition, during an analysis several partial-results are saved to a file. In this way further analyses can reuse such partial-results without recomputing them. Saving and loading operations are managed by **PickleCache** class that, in turn, uses CPyickle Python library. Examples of cached partial-results are: **DataDescriptions** dictionaries and all the distances between two memory dumps computed during untrained analyses.

7.7.3 Other optimizations

Many internal methods are invoked several times using contiguous input parameters and, most of the times, they return the same results. For instance the method that, given an RVA in a kernel module, returns the PE section where such RVA is located always returns the same result if it is invoked with RVAs within the same 0x80-byte aligned memory region

(because PE sections are always 0x80-byte aligned). As shown in Listing 9, this property is exploited to avoid searching for a PE section every time this method is called.

```
1 Module_RVAtToSection(module, rva){
2     static last_module = None
3     static last_rva = None
4     static last_result = None
5
6     if(last_module == module
7     and (last_rva >> 7) == (rva >> 7)){ //because 2^7 == 0x80
8         return last_result
9     }else{
10         //search for the PE section at module:rva location
11         //...
12         last_module = module
13         last_rva = rva
14         last_result = found_section
15         return found_section
16     }
```

Listing 9: An example of speed optimization.

CHAPTER 8

EVALUATION

8.1 Used data sets

To evaluate *Blacksheep* we have used two sets of memory dumps acquired from differently configured machines. Characteristics of the utilized machines are summarized in Table III. In this chapter we will always refer to the two sets as *WinXP* and *Win7*.

TABLE III: THE SETS OF MEMORY DUMPS USED TO EVALUATE BLACKSHEEP.

Set name	Operating system	Last OS update	Installed programs	Machine configuration	Dumping method used
WinXP	Windows XP SP3 Professional	31 Oct 2011	<ul style="list-style-type: none">– Firefox– Process Explorer– Visual Studio	<ul style="list-style-type: none">– VirtualBox– 2.8 GHz, 1 32-bit Core– 1GB RAM	Custom dumping driver
Win7	Windows 7 SP1 Professional	31 Jan 2012	<ul style="list-style-type: none">– Default applications only	<ul style="list-style-type: none">– QEMU– 2.8 GHz, 1 32-bit Core– 1GB RAM– Swap file disabled	Virtual machine introspection

8.2 Tested rootkits

We have tested *Blacksheep* with different Windows kernel rootkits. Their droppers (the executable files used to install them) have been download from the Contagio blog¹ and Offensive Computing² websites.

¹<http://contagiodump.blogspot.com/>

²<http://offensivecomputing.net/>

TABLE IV: ROOTKITS USED TO TEST BLACKSHEEP.

The name we use during testing is listed in Name column.

Name	Avast classification	McAfee classification	Symantec classification
blackenergy	Win32:MalOb-M [Cryp]	BlackEnergy.gen.e	Packed.Generic.265
mebroot	Win32:Akan	Generic Packed.g	Trojan.Mebroot
r2d2	Win32:R2D2-E [Trj]	BackDoor-FCA	Backdoor.R2D2
rustock	Win32:Costrat-T [Trj]	Artemis!93BB8478468E	Backdoor.Rustock.B
stuxnet	Win32:StuxX-B [Wrm]	Stuxnet	W32.Stuxnet
tdl3	Win32:Jifas-FB [Trj]	DNSChanger.bf	Packed.Vuntid!gen1
tdss	Win32:Alureon-MT [Rtk]	Generic Dropper.va.gen.m	Trojan.Gen.2
zeroaccess	Win32:MalOb-IJ [Cryp]	Artemis!3FD3D439553B	Trojan.Gen









Table IV lists rootkits used during the evaluation of *Blacksheep*. Classifications provided by popular Antivirus have been retrieved by uploading droppers to the VirusTotal¹ website.

¹<https://www.virustotal.com/>

Assigning the right classification to a rootkit is an open problem. We have given names to tested rootkits using the following criteria. In case that a manual analysis is available, we have used the name provided in the analysis, otherwise we have chosen the most common name returned by VirusTotal automatic analyses.

In this chapter every infected memory dump is named with the name of the rootkit installed, followed by a three digit random number. Memory dumps acquired from machines in a non-infected state are named as *non-infected*.

TABLE V: ROOTKITS USED TO TEST BLACKSHEEP: WINDOWS 7 COMPATIBILITY AND HASHES.

Name	Works in Windows 7 ?	Hash (MD5, SHA-1, SHA-256)
blackenergy		45f3085144f6875b27ae173f22856198 f20b4a1ad872b2b78505c94c70e7f19a50438e12 58db0a680d11647fa5a74fcda1936ff54484a259300e1d92d7d8f656f62bdb96
mebroot		de53c6367f284bba1137783c6041ee18 5549fff5f70d019c4a6f94e73a0e9e01b5c0188f f318330548dedc76b1ce17bf53f7358e7a840f0358dd04e60764f02779257849
r2d2		309ede406988486bf81e603c514b4b82 a6a0f45180f5b3390ee2ef21fe4b89813ed641f4 021da2f5e892265cafd1642a44fe258ee56cf6e1393f6e0dc79add99fed1f15f
rustock		93bb8478468e9dce3506f1f8b6655c20 4a3f4448038f042bee221747500aafb867a78156 f1bf0e9cb488074ad7a394dd4cf5e2299e241660454679c982a0d2be3abb4e48
stuxnet		74ddc49a7c121a61b8d06c03f92d0c13 0ccbc128dd8bf73dc7b3922fb67d26bbcdcbcaa89 743e16b3ef4d39fc11c5e8ec890dcd29f034a6eca51be4f7fca6e23e60dbd7a1
tdl3		44cd40833af9bf801999217c9247bc56 6a3f9c09c438c4cdd43895cf81bcbebcd18f37a3 dab1fa7cfc4dc2562b1f60491ecbbbf3d51f328948a1ae211294d3f8b91af95
tdss		31db7a22df02e1a91db9afda4f02f3bf 6ede4482be1b06c90cca93bedf3e363c096102f5 ba670c68a7e481c324bdc2e8c5c8c1c8ddc4a2772e991826771350ea8e03f2ce
zeroaccess		3fd3d439553b6dd40b25ef28c07337ed 43c2f41d6e4d49e7082f2401b2d027565051d96f 2e0f78f3b8452e10d118e2d668fb7c5005ec8cd02e9deb7eef2e255eb0188ccf

Some tested rootkits do not work in Windows 7, since their droppers crash immediately after their opening. For some rootkits, after that the execution of their droppers starts, Windows 7 asks for user confirmation before performing system-critical operations. This is due to how application permissions are set by default in Windows 7 User Access Control (UAC). In this case we always give to UAC dialog forms all the confirmations required. In addition all droppers have been executed with administrative privileges.

8.3 Methodology

From a machine in a non-infected state 20 memory dumps have been acquired. Then, for each tested rootkit, it has been infected, rebooted and 4 memory dumps have been acquired. Since we have used virtual machines it has been possible to infect it while always starting from exactly the same non-infected state.

Memory dumps have been acquired in different operating system sessions, in particular, each system have been always rebooted after the acquisition of two memory dumps. In *Win7* set the dumping procedure has been started as soon as the operating system boot has finished. In *WinXP* set, before acquiring memory dumps, some installed programs have been randomly opened.

In trained analyses 10 out of 20 memory dumps acquired from machines in non-infected state have been used as the training set. The others have been analyzed to determine the false positive rate.

Untrained analyses have been applied on sets of 14 dumps: 10 acquired from machines in a non-infected state, the others from machines with a specific infection. The hierarchical

sets of cluster obtained as a result have been flattened using the *fcluster*¹ function of SciPy mathematical library. If the flattening procedure returns more than two clusters, the smaller ones are merged together to have only two clusters. The biggest flat cluster obtained is considered as that containing non-infected memory dumps. Ideally, the biggest cluster should comprise all non-infected memory dumps (and only those dumps), leaving all (and only) memory dumps acquired from machines in an infected state in the other cluster. If only one cluster is returned (spanning all analyzed memory dumps), all dumps are considered as non-infected.

¹<http://docs.scipy.org/doc/scipy/reference/generated/scipy.cluster.hierarchy.fcluster.html>

Parameters used:

criterion = 'maxclust' t = 5.0 for code and kernel entry point untrained analyses.
criterion = 'maxclust' t = 2.0 for data untrained analysis.

8.4 Data set 1 - Trained Analyses

8.4.1 Code and Kernel Entry Point analyses

TABLE VI: OVERVIEW OF THE RESULTS IN *WINXP* SET, USING TRAINED ANALYSES.

Name	Code Analysis	Kernel Entry Point Analysis	Total ^a	Note
blackenergy	4/4	4/4	4/4	<ul style="list-style-type: none"> – Modifications in <i>ntoskrnl.exe</i> – 1 module installed (randomly named)^b – 14 modified SSDT entries
mebroot	4/4	4/4	4/4	<ul style="list-style-type: none"> – Modifications in <i>CLASSPNP.SYS</i> – 22 modified IRPs
r2d2	4/4	0/4	4/4	<ul style="list-style-type: none"> – 1 module installed: <i>winsys32.sys</i>
rustock	4/4	4/4	4/4	<ul style="list-style-type: none"> – Modifications in <i>ntoskrnl.exe</i>, <i>tcpip.sys</i>, <i>wanarp.sys</i> – SYSENTER and Interrupt 0x2E handlers changed
stuxnet	4/4	0/4	4/4	<ul style="list-style-type: none"> – 2 modules installed: <i>mrxnet.sys</i>, <i>mrxcls.sys</i>
tdl3	4/4	4/4	4/4	<ul style="list-style-type: none"> – 1 module installed: <i>compbatt.sys</i>^c – 28 added IRPs and 1 modified IRP
tdss	4/4	4/4	4/4	<ul style="list-style-type: none"> – Modifications in <i>hal.dll</i> – 1 module installed: <i>kdcom.dll</i>^d
zeroaccess	4/4	4/4	4/4	<ul style="list-style-type: none"> – Modifications in <i>ntoskrnl.exe</i>, <i>atapi.sys</i> – 1 module installed: <i>netbt.sys</i> – 28 added IRPs
total	32/32	24/32	32/32	100% detection rate using both analyses
non-infected	0/10	0/10	0/10	0% false positive rate

- a In this column memory dumps classified as infected by code analysis *OR* kernel entry point analysis are counted.
- b Module name format is: *00000XXX* where X is any hexadecimal digit.
Module size and CRC are always *0x0000A0C9* and *0x00021E88*.
- c *compbatt.sys* is a legitimate Windows module, the installed one has the same CRC, but a different size.
- d *kdcom.dll* is a legitimate Windows module, the installed one has the same CRC, but a different size.

8.4.2 Data analysis

The example in Listings 10 shows a difference found by trained data analysis on *WinXP* set. In a memory dump infected by the *blackenergy* rootkit, the pointer to the SSDT used by some *KTHREAD* elements has been changed. In this way the corresponding threads use a modified SSDT, in which some function pointers have been modified by the rootkit.

```

1  Difference found: different_structure
2
3  //memory dump infected by blackenergy rootkit
4  POINTER_H    0x81ed6208L size: 0x11L
5  //this SERVICE_DESCRIPTOR_TABLE has been created by the rootkit
6      -0x8  HEAP_HEADER 0xa090002 - PType:0x5L - BSize:0x9 - PIndex:0x0 - PSize:0x2
7      -0x4  HEAP_TAG  None
8      0x0   POINTER_V   0x80599a74L //pointer to the hooked SSDT installed by blackenergy rootkit
9      0x4   ZERO        0x0
10     0x8   VALUE       0x11c
11     0xc   POINTER     ('ntoskrnl.exe', 176176L, '.text')
12     0x10  ZERO        0x0
13  //...
14     0x38  ZERO        0x0
15
16  //invariant set (created using 10 non-infected memory dumps)
17  POINTER_S    0x805530a0L size: 0xb
18  //legitimate SERVICE_DESCRIPTOR_TABLE
19     -0x8   ZERO        0x0
20     -0x4   ZERO        0x0
21     0x0    POINTER     ('ntoskrnl.exe', 175036L, '.text') //legitimate SSDT
22     0x4    ZERO        0x0
23     0x8    VALUE       0x11c
24     0xc    POINTER     ('ntoskrnl.exe', 176176L, '.text')
25     0x10   ZERO        0x0
26     0x14   ZERO        0x0
27     0x18   ZERO        0x0
28     0x1c   ZERO        0x0
29     0x20   ZERO        0x0

```

Listing 10: A detected data difference caused by a rootkit.

8.5 Data set 1 - Untrained Analyses

8.5.1 Code and Kernel Entry Point analyses

TABLE VII: OVERVIEW OF THE RESULTS IN *WINXP* SET, USING UNTRAINED ANALYSES.

Name	Code Analysis		Kernel Entry Point Analysis		Total ^a	
	True	False	True	False	True	False
	Positives	Positives	Positives	Positives	Positives	Positives
blackenergy	4/4	0/10	4/4	0/10	4/4	0/10
mebroot	4/4	0/10	4/4	0/10	4/4	0/10
r2d2	0/4	0/10	0/4	0/10	0/4	0/10
rustock	4/4	0/10	4/4	0/10	4/4	0/10
stuxnet	0/4	0/10	0/4	0/10	0/4	0/10
tdl3	0/4	0/10	4/4	0/10	4/4	0/10
tdss	4/4	0/10	4/4	0/10	4/4	0/10
zeroaccess	4/4	0/10	4/4	0/10	4/4	0/10
total	20/32	0/80	24/32	0/80	24/32	0/80
	62.5%	0%	75%	0%	75%	0%

- a In this column memory dumps classified as infected by code analysis *OR* kernel entry point analysis are counted.

8.5.2 Data analysis

TABLE VIII: OVERVIEW OF THE RESULTS IN *WINXP* SET, USING UNTRAINED DATA ANALYSIS.

Name	Data Analysis	
	True Positives	False Positives
blackenergy	0/4	1/10
mebroot	1/4	2/10
r2d2	0/4	1/10
rustock	0/4	1/10
stuxnet	0/4	1/10
tdl3	0/4	1/10
tdss	0/4	1/10
zeroaccess	2/4	2/10
total	3/32	10/80
	9.4%	12.5%

8.6 Data set 2 - Trained Analyses

8.6.1 Code and Kernel Entry Point analyses

TABLE IX: OVERVIEW OF THE RESULTS IN WIN7 SET, USING TRAINED ANALYSES.

Name	Code Analysis	Kernel Entry Point Analysis	Total ^a	Note
r2d2	4/4	0/4	4/4	– 1 module installed: <i>winsys32.sys</i>
stuxnet	4/4	2/4	4/4	– 2 modules installed: <i>mrnet.sys</i> , <i>mrcls.sys</i> – 20 added IRPs (only in 2 dumps)
tdl3	4/4	4/4	4/4	– 1 module installed: <i>netbt.sys</i> ^b – 29 added IRPs
tdss	4/4	4/4	4/4	– 1 module installed: <i>kdcom.dll</i> ^c – 28 added IRPs
zeroaccess	4/4	4/4	4/4	– Modifications in <i>ntoskrnl.exe</i> , <i>ataport.SYS</i> – 1 module installed: <i>cdrom.sys</i> ^d – 56 added IRPs
total	20/20	14/20	20/20	100% detection rate using both analyses
non-infected	0/10	0/10	0/10	0% false positive rate

- a In this column memory dumps classified as infected by code analysis *OR* kernel entry point analysis are counted.
- b *netbt.sys* is a legitimate Windows module, the installed one has the same size, but a different CRC.
- c *kdcom.dll* is a legitimate Windows module, the installed one has the same CRC, but a different size.
- d *cdrom.sys* is a legitimate Windows module, the installed one has the same size, but a different CRC.

8.6.2 Data analysis

By manually inspecting differences found between the analyzed memory dumps and the *invariance set*, it is possible to notice that for memory dumps infected by rootkits of the TDSS family (*zeroaccess*, *tdss*, *tdl3*), many are located within the *CI.dll* kernel module. This module is responsible for verifying the signatures of all loaded kernel modules. It is well known that these rootkits force Windows to disable signature verification on modules (for details see (9)). They achieve this result by booting Windows with special boot flags or tampering its boot procedure. For these reasons we think that modifications detected within the *CI.dll* kernel module are caused by activities performed by these rootkits. Listing 11 shows three of these modifications.

```

1 -----
2 Difference found: zero_to_structure
3 //location CI.dll:0x0000406C --> PVOID g_UnsignedHash
4
5 //in invariant set:
6 ZERO 0x0
7 //in infected memory dump:
8 POINTER_H 0x91d5b1a8L size: 0x49L
9     -0x8 HEAP_HEADER 0x625020d - PType:0x3L - BSize:0x25 - PIndex:0x1 - PSize:0xd
10    -0x4 HEAP_TAG C1cr
11    0 VALUE 0x2a
12    //...
13 -----
14 Difference found: zero_to_something
15 //location CI.dll:0x00004090 --> unsigned long g_UnsignedHashChanged
16
17 //in invariant set:
18 ZERO 0x0
19 //in infected memory dump:
20 VALUE 0x1
21 -----
22 Difference found: zero_to_something
23 //location CI.dll:0x00004094 --> unsigned long g-ulUnsignedHashCount
24
25 //in invariant set:
26 ZERO 0x0
27 //in infected memory dump:
28 VALUE 0x1

```

Listing 11: Three examples of data modifications within the *CI.dll* kernel module caused by *zeroaccess* rootkit activities.

Names assigned by Windows debugging symbols to the three locations shown are printed in comments (after the \longrightarrow symbol).

8.7 Data set 2 - Untrained Analyses

8.7.1 Code and Kernel Entry Point analyses

TABLE X: OVERVIEW OF THE RESULTS IN *WIN7* SET, USING UNTRAINED ANALYSES.

Name	Code Analysis		Kernel Entry Point Analysis		Total ^a	
	True	False	True	False	True	False
	Positives	Positives	Positives	Positives	Positives	Positives
r2d2	4/4	0/10	0/4	0/10	4/4	0/10
stuxnet	0/4	0/10	2/4	0/10	2/4	0/10
tdl3	4/4	0/10	4/4	0/10	4/4	0/10
tdss	0/4	0/10	4/4	0/10	4/4	0/10
zeroaccess	4/4	0/10	4/4	0/10	4/4	0/10
total	12/20	0/50	14/20	0/50	18/20	0/50
	60.0%	0%	70.0%	0%	90.0%	0%

- a In this column memory dumps classified as infected by code analysis *OR* kernel entry point analysis are counted.

8.7.2 Data analysis

TABLE XI: OVERVIEW OF THE RESULTS IN *WIN7* SET, USING UNTRAINED DATA ANALYSIS.

Name	Data Analysis	
	True Positives	False Positives
r2d2	1/4	3/10
stuxnet	1/4	3/10
tdl3	4/4	0/10
tdss	4/4	0/10
zeroaccess	4/4	0/10
total	14/20	6/50
	70.0%	12.0%

8.8 Other results

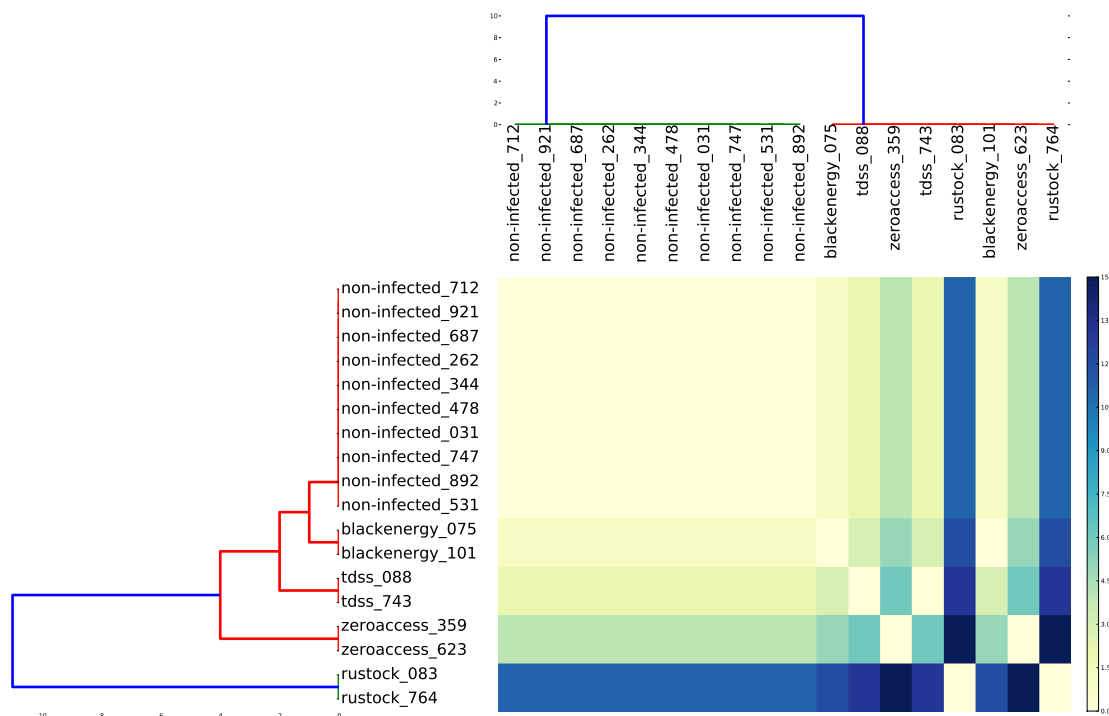


Figure 12: An example of code untrained analysis on a set of memory dumps with multiple infections.

In the graph on the left the hierarchy of clusters returned by untrained code analysis is shown, in the graph on the top the two clusters obtained after the flattening procedure are shown. In the center, the difference matrix is drawn.

All memory dumps belong to the WinXP set.

8.9 Execution time and memory consumption

The execution time of *Blacksheep* is strictly linked to the type and the number of the performed comparisons. Table XII shows the number of comparison needed by each type of analysis.

TABLE XII: NUMBER OF COMPARISONS NEEDED BY EACH TYPE OF ANALYSIS.

n : number of memory dumps used in untrained analysis.

c : number of memory dumps to be checked by trained analysis.

t : number of memory dumps in the training set.

Analysis type	Number of comparisons
Code untrained analysis	$\frac{n(n-1)}{2}$
Kernel entry point untrained analysis	$n(n-1)$
Data untrained analysis	$\frac{n(n-1)}{2}$
Code trained analysis	between c and ct
Kernel entry point trained analysis	between c and ct
Data trained analysis	$c + t$

Figure 13 shows execution time of each type of comparison. Comparing memory dumps already used in other comparisons is significantly faster, because of the several caching mechanisms implemented in *Blacksheep*. The analyses have been performed on an Intel Core i7 CPU (2.8 GHz quad-core) running Ubuntu 10.4 32-bit. Results shown in Figure 13 have been acquired by comparing 10 different couples of memory dumps from the *Win7* set and computing the average execution time.

In addition to the implemented caching mechanisms, all partial results of untrained analyses are saved and reused in further analyses. For this reason, when a memory dump is added to a set of n already compared memory dumps, only n (for data or code untrained analyses) or $2n$ (for kernel entry point untrained analysis) comparisons are necessary. The execution time needed to perform clustering is negligible.

Memory consumption is maximum during data analyses. The maximum value recorded while comparing memory dumps from the *Win7* set is 2.5 GB.

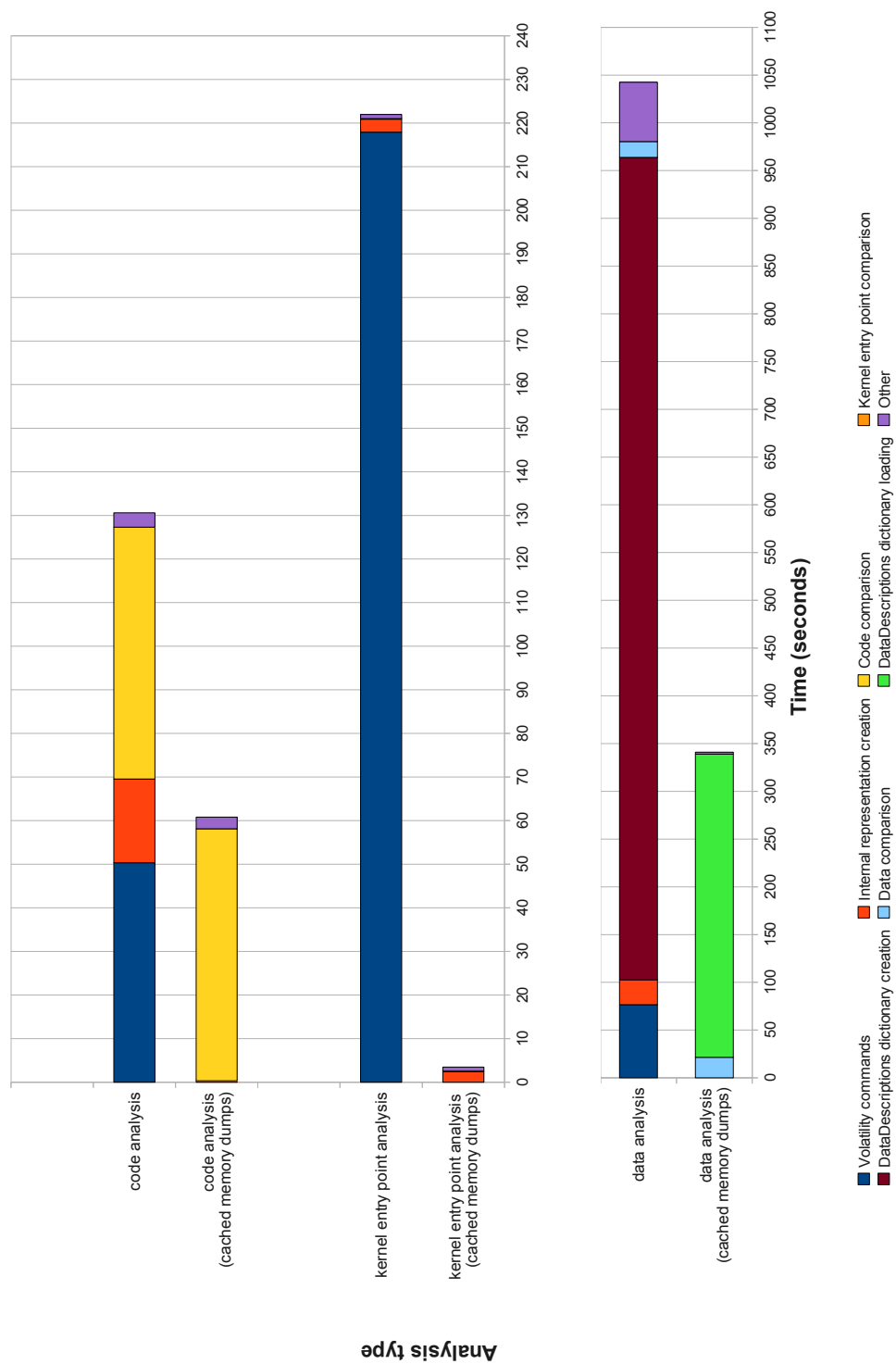


Figure 13: Execution time during different types of comparisons.

8.10 Discussion

8.10.1 Trained analyses

In both *WinXP* and *Win7* sets, *Blacksheep* is able to detect all the tested rootkits in any memory dump acquired from an infected machine. Furthermore all non-infected memory dumps are classified correctly. These results have been obtained using a relative small set of 10 memory dumps as the training set. Additionally *Blacksheep* provides detailed information of the anomalies detected that can be used for a preliminary analysis of the behaviors of the installed rootkits. All kernel modules detected as infected are extracted from the memory dump where they have been found to allow further investigations.

An interesting common behavior some detected rootkits have is that they install kernel modules with names equal (e.g. *kdcom.dll*, *cdrom.sys*) or similar (e.g. *winsys32.sys*) to legitimate ones. In one case, a rootkit (*blackenergy*) installs a kernel module with a randomized name. In some cases, they also try to mimic CRC and size values. However in all the infection we have tested they fail to do so correctly. From these results we can state that the way in which *Blacksheep* identifies kernel modules is effective in dealing with kernel modules installed by the analyzed rootkits.

Most of the rootkits add or modify IRPs to interfere with operating system behavior. Two other interesting techniques have been detected too. *Blackenergy* modifies SSDT entries and *rustock* modifies SYSENTER and Interrupt 0x2E handlers. Patching code

within Windows kernel modules is a widely used technique. The most targeted modules are *ntoskrnl.exe* and *hal.dll*.

Two rootkits (*stuxnet* and *r2d2*) only install new kernel modules, without performing any other detectable modification to the OS. For this reason they cannot be detected correctly by kernel entry point analysis, but the modules they install are still detected by trained code analysis.

Trained data analyses reveal interesting information too, even if manual intervention is necessary to spot modifications related to malicious activities. In *WinXP* set it is possible to understand in detail how *blacksheep* rootkit modifies SSDT entries. It does not directly patch them, instead it modifies the pointer to the *SERVICE_DESCRIPTOR_TABLE* of some threads. We think that this technique is used to deceive security tools that only check entries within legitimate SSDTs. In *Win7* set many fields within the *CI.dll* module that have ZERO value in non-infected memory dumps change their values in memory dumps infected by rootkits of the TDSS family. We think that these modifications are caused by the fact that rootkits of this family force the OS to disable kernel module signature verification. Further investigations should be carried out to discover the exact way in which these modifications are caused.

8.10.2 Untrained analyses

Untrained analyses on code and kernel entry points works effectively in detecting most of the rootkits in *WinXP* and *Win7* sets. Two of them (*stuxnet* and *r2d2*) cannot be detected because neither they modify kernel modules nor kernel entry points. They can be

detected during trained analysis because they install new kernel modules, however newly installed modules are ignored during untrained analysis. A possible solution would be to consider differences in loaded kernel modules while clustering memory dumps. However a major issue needs to be addressed: some kernel modules are loaded only on demand by operating system or applications so they are not always present. In trained analysis this problem is solved having a sufficiently large training set, but in untrained analysis it causes instability in the results.

Additionally, in some cases untrained analysis is able to produce a hierarchy of clusters in which infected memory dumps are separated from the non-infected ones, but this hierarchy is not flattened correctly.

Untrained data analysis achieves good results in *Win7* set. Many of the detected modifications are within the *CI.dll* kernel module and they have been manually inspected using trained data analysis too. In *WinXP* set results are not satisfactory mainly because the dumping method used introduces too much instability in the acquired data structures.

With some infections untrained code and kernel entry point analyses are able to generate correct clusters even when working with a set of dumps infected by multiple rootkits (see Figure 12). This could be particularly useful to give a preliminary automatic classification of rootkit samples based on how they modify Windows kernel.

CHAPTER 9

CONCLUSIONS AND FUTURE WORK

In this thesis we have developed *Blacksheep* a tool for automatic kernel rootkit detection and we have tested it on two different sets of memory dumps. This tool performs different analyses on sets of memory dumps. Best results are achieved with trained analyses based on the comparison of code and kernel entry points. In this case we are able to detect all infected memory dumps without any false positive.

In addition, *Blacksheep* is able to provide useful information on how rootkits, infecting the analyzed memory dumps, operate. So it can also be used as a tool to provide a preliminary automatic analysis of rootkit behaviors.

The current version of *Blacksheep* can be improved in several ways.

A very interesting feature that may be added is the ability to analyze 64-bit versions of Windows too. A limited number of security tools currently support these operating systems, whereas the number of rootkits that can infect them is increasing (e.g. the TDSS rootkit family (9)). Differently from many Antivirus software, *Blacksheep* is not effected by the presence of Kernel Patch Protection because it does not need to modify the kernel in any way. Version 2.1 of Volatility Framework (currently in alpha stage) introduces 64-bit compatibility and it could be used by *Blacksheep* as well as the current one.

There is a considerable room for improvements in data comparison. Using existing tools such as Daikon (29) it would be possible to have a more extensible and precise generation of invariant properties, following the approach already implemented in Gibraltar (30). Some modifications are necessary to make Daikon able to deal with memory dumps because it has been originally developed to analyze invariant properties on pre-conditions and post-conditions of function parameters. Data analysis could be further improved by the availability of Windows data structure definitions. Even without having the source code, it is possible to infer many Windows data structures parsing debugging symbols that are publicly available. Many extracted data structures are already available, for instance on NirSoft website (42).

Clustering algorithms used could be improved in several ways. An ad-hoc linkage function could be developed and used while comparing two clusters of memory dumps, taking into account differences caused by wrongly mapped memory pages or not loaded kernel modules. Also the flattening procedure, used to obtain two clusters, one with the infected memory dumps and the other with the non-infected ones, should be improved.

Many performance improvements are possible. The way in which addresses are translated could be implemented using a native language (i.e. C/C++) to maximize speed. Even the way in which DWORD descriptions are stored could be improved by using optimized data structures to minimize memory consumption. Comparisons needed by untrained analyses are performed independently so they could be easily computed in parallel on a distributed system.

Finally, *Blacksheep* could be further tested both with other rootkits and with other sets of memory dumps acquired from machines with different configurations.

CITED LITERATURE

1. Kapoor, A. and Mathur, R.: Predicting the future of stealth attacks. Virus Bulletin conference, October 2011.
2. Treit, R.: Some Observations on Rootkits. <http://blogs.technet.com/b/mmpc/archive/2010/01/07/some-observations-on-rootkits.aspx>, January 2010.
3. Rutkowska, J.: Thoughts on DeepSafe. <http://theinvisiblethings.blogspot.com/2012/01/thoughts-on-deepsafe.html>, January 2012.
4. Rutkowska, J.: Security Challenges in Virtualized Environments. RSA Conference, April 2008.
5. Davis, M. A., Bodmer, S. M., and LeMasters, A.: Hacking Exposed, Malware and Rootkits. McGraw-Hill Osborne Media, 2009.
6. Arnold, T. M.: A comparative analysis of rootkit detection techniques. Master's thesis, University of Houston-Clarke Lake, May 2011.
7. Li, Z., Sanghi, M., Chen, Y., yang Kao, M., and Chavez, B.: Hamsa: fast signature generation for zero-day polymorphic worms with provable attack resilience. In SP 06: Proceedings of the 2006 IEEE Symposium on Security and Privacy, pages 32–47. IEEE Computer Society, 2006.
8. Griffin, K., Schneider, S., Hu, X., and cker Chiueh, T.: Automatic Generation of String Signatures for Malware Detection.
9. Rodionov, E. and Matrosov, A.: The Evolution of TDL: Conquering x64. ESET, June 2011.
10. Jacob, G., Debar, H., and Filiol, E.: Behavioral detection of malware: from a survey towards an established taxonomy. Journal in Computer Virology, 4:251–266, 2008. 10.1007/s11416-008-0086-0.

11. Maggi, F., Matteucci, M., and Zanero, S.: Detecting Intrusions through System Call Sequence and Argument Analysis. IEEE Transactions on Dependable and Secure Systems., 7(4), December 2010.
12. Microsoft: Kernel Patch Protection: FAQ. <http://msdn.microsoft.com/en-us/windows/hardware/gg487353>, September 2007.
13. Russinovich, M. E., Solomon, D. A., and Ionescu, A.: Windows Internals. Microsoft Press, 5th edition, June 2009.
14. Rutkowska, J.: Rootkits vs. Stealth by Design Malware. <https://www.blackhat.com/presentations/bh-europe-06/bh-eu-06-Rutkowska.pdf>, 2006.
15. Wang, Z., Jiang, X., Cui, W., and Ning, P.: Countering Kernel Rootkits with Lightweight Hook Protection. In ACM Conf. on Computer and Communications Security, November 2009.
16. Yin, H., Poosankam, P., Hanna, S., and Song, D.: HookScout: Proactive Binary-Centric Hook Detection. In Proceedings of the 7th Conference on Detection of Intrusions and Malware & Vulnerability Assessment, Bonn, Germany, July 2010.
17. Seshadri, A., Luk, M., Qu, N., and Perrig, A.: SecVisor: A Tiny Hypervisor to Provide Lifetime Kernel Code Integrity for Commodity OSes, 2007.
18. McAfee: McAfee DeepSAFE. <http://www.mcafee.com/us/solutions/mcafee-deepsafe.aspx>, 2011.
19. Blunden, B.: The Rootkit Arsenal. Wordware Publishing, 2009.
20. Garcia, G. L.: Forensic Physical Memory Analysis: an overview of tools and techniques. In TKK T- 110.5290 Seminar on Network Security, 2007.
21. Burdach, M.: Finding Digital Evidence in Physical Memory. In Black Hat Federal Conference, 2006.
22. HBGary: HBGary Responder Pro. <http://www.hbgary.com/responder-pro>.

23. Walters, A.: The Volatility framework: Volatile memory artifact extraction utility framework. <https://www.volatilesystems.com/default/volatility>.
24. Ligh, M. H.: Volatility malware plugins. <http://code.google.com/p/malwarecookbook>.
25. Schuster, A.: Pool Allocations as an Information Source in Windows Memory Forensics. In Pool Allocations as an Information Source in Windows Memory Forensics, 2006.
26. Schuster, A.: Searching for processes and threads in Microsoft Windows memory dumps. In Digital Investigation, 2006.
27. Kornblum, J. D.: Using Every Part of the Buffalo in Windows Memory Analysis. Digital Investigation, March 2007.
28. Petroni, N. L., Timothy, J., Aaron, F., William, W., and Arbaugh, A.: An architecture for specification-based detection of semantic integrity violations in kernel dynamic data. In Proceedings of the USENIX Security Symposium, pages 289–304, 2006.
29. Ernst, M. D., Perkins, J. H., Guo, P. J., McCamant, S., Pacheco, C., Tschantz, M. S., and Xiao, C.: The Daikon system for dynamic detection of likely invariants. Science of Computer Programming, 69, December 2007.
30. Baliga, A., Ganapathy, V., and Iftode, L.: Detecting Kernel-Level Rootkits Using Data Structure Invariants. IEEE Transactions on Dependable and Secure Computing, Vol. 8, No. 5, September 2010.
31. Carbone, M., Lee, W., Cui, W., Peinado, M., Lu, L., and Jiang, X.: Mapping kernel objects to enable systematic integrity checking. In ACM Conf. on Computer and Communications Security, 2009.
32. Lin, Z., Rhee, J., Zhang, X., Xu, D., and Jiang, X.: SigGraph: Brute Force Scanning of Kernel Data Structure Instances Using Graph-based Signatures. In the 17th Network and Distributed System Security Symposium, 2011.
33. Kornblum, J. D.: Exploiting the Rootkit Paradox with Windows Memory Analysis. International Journal of Digital Evidence, 2006.

34. Rutkowska, J.: Beyond The CPU: Defeating Hardware Based RAM Acquisition (part I: AMD case). In Black Hat DC, 2007.
35. Intel: Intel 64 and IA-32 Architectures Software Developer's Manual - Volume 3A. September 2010.
36. Microsoft: Microsoft PE and COFF Specification. <http://msdn.microsoft.com/en-us/windows/hardware/gg463119>, September 2010.
37. Hoglund, G. and Butler, J.: Rootkits: Subverting the Windows Kernel. Addison Wesley Professional, August 2005.
38. Matrosov, A., Rodionov, E., Harley, D., and Malcho, J.: Stuxnet Under the Microscope. ESET, January 2011.
39. Tereshkin, A.: Rootkits: Attacking Personal Firewalls. CODEDGERs, 2006.
40. Jones, E., Oliphant, T., Peterson, P., et al.: SciPy: Open source scientific tools for Python. <http://www.scipy.org/>, 2001–.
41. ReversingLabs: Undocumented PECOFF. Black Hat USA, 2011.
42. NirSoft: Windows Vista Kernel Structures. http://www.nirsoft.net/kernel_struct/vista/.
43. Rigo, A.: Representation-based just-in-time specialization and the Psycho prototype for Python. In Proceedings of the 2004 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-based Program Manipulation, pages 15–26. ACM Press, 2004.

VITA

Name: Antonio Bianchi

Education:

March 2009 – May 2012 **M.Sc. in Computer Science**

University of Illinois at Chicago

Final GPA: 3.71/4

September 2008 – April 2012 **Laurea Specialistica degree (equivalent to M.Sc.)**

in Computer Engineering

Politecnico di Milano

Grade: 110/110 cum laude

September 2005 – September 2008 **Laurea degree (equivalent to Bachelor)**

in Computer Engineering

Politenico di Milano

Grade: 108/110

Work experience:

August 2011 – October 2011 **Research Assistant**

Security Lab; University of California, Santa Barbara

Santa Barbara, CA, USA

October 2010 – June 2011 **Tutor for foreign students**

Politecnico di Milano

Milan, Italy

January 2006 – June 2008 **Web developer**

Liceo Scientifico Niccolò Machiavelli

Pioltello, Italy