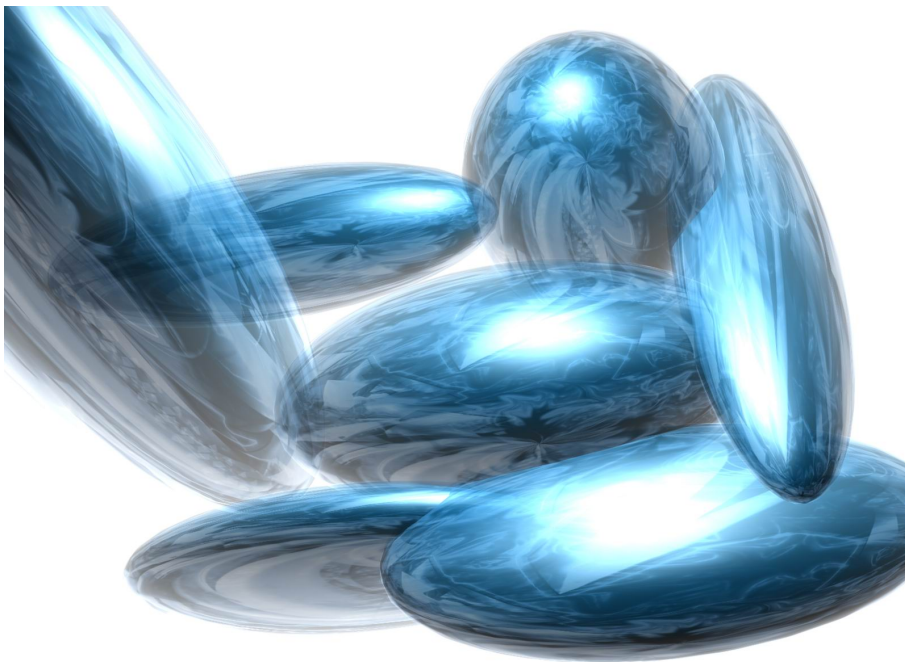


TECHNISCHE UNIVERSITÄT MÜNCHEN

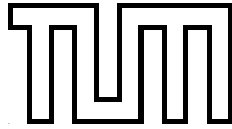
FAKULTÄT FÜR INFORMATIK

Bachelorarbeit in Informatik

Analysis and detection of virtualization-based rootkits



Hagen Fritsch



TECHNISCHE UNIVERSITÄT MÜNCHEN

FAKULTÄT FÜR INFORMATIK

Bachelorarbeit in Informatik

Analysis and detection of virtualization-based rootkits

Analyse und Erkennung von virtualisierungsbasierten Rootkits

Bearbeiter: Hagen Fritsch
Supervisor: Prof. Dr. Heinz-Gerd Hegering
Advisors: Tobias Lindinger
Nils gentschen Felde
Submission Date: 27. August 2008

Abstract

With the emergence of hardware virtualization, it was discussed if this technology gives ground for a potentially undetectable form of malware. While several claims have been made, this thesis presents this malware technology's state of the art, describes possible detection methods and demystifies the topic referencing the current literature, analyzing a sample implementation and validating results on a special testing machine. Though the 100% undetectability claim does not hold due to a variety of attack vectors that have been presented over the last two years, such malware raises the bar for detection, especially since real code detection requires ways to get raw memory access which is only possible by getting even closer to the hardware. With this thesis, a testing framework for detection using side-channel attacks is presented implementing some of the proposed detection methods.

Contents

1	Introduction	1
2	Prerequisites	3
2.1	A rootkit taxonomy	3
2.1.1	Type 0 malware	3
2.1.2	Type 1 malware	3
2.1.3	Type 2 malware	5
2.1.4	Type 3 malware	6
2.2	Mass rootkits vs. targeted attacks	6
3	HVM rootkits	9
3.1	Basic Principles	9
3.1.1	Implementing a thin hypervisor	10
3.1.2	Abusing hardware virtualization for malware	10
3.2	Technical Implementation Details	10
3.2.1	Moving the operating system into the virtual machine	11
3.2.2	Further hypervisor tasks	12
3.2.3	Details on Bluepill	13
4	On detecting HVM rootkits	15
4.1	Virtualization detection	15
4.1.1	Timing Attacks	15
4.1.2	CPU-specific behaviour	17
4.1.3	Profiling CPU resource discrepancies	18
4.1.4	Counter-based detection	21
4.1.5	Side Channel Attacks with nested virtualization	21
4.2	Explicitly detecting a malicious hypervisor	23
4.2.1	Signature Detection	23
4.2.2	Hardware-based memory acquisitions	24
4.2.3	“Insane Detection of Insane Rootkits”	25
4.3	Passive detection	25
4.4	Removal of a malicious hypervisor	25
5	On preventing HVM rootkits	27
5.1	Hardware supported authorization	27
5.1.1	BIOS support	27
5.1.2	Unlockable virtualization	27
5.1.3	TPM support	27
5.2	Installing a trusted hypervisor first	28

6	Conclusions and Prospective	29
6.1	Threat-Evaluation	29
6.2	Undetectable Stealth Rootkits	30
6.3	Applications for thin hypervisors	30
6.4	Further tasks / research	30
A	Empirical Results	33
A.1	Preparing Windows for home-brew drivers	33
A.2	Running Bluepill	34
A.2.1	Verifying Bluepill’s presence in a debug environment	35
A.3	Own Developments	36
A.3.1	Implementation of the counter-based detection	36
A.3.2	Implementation of TLB-based detection	36
A.4	Empirical results	38
	List of Figures	41
	Bibliography	43

1 Introduction

The current development of virtualization technology attracts a lot of attention and offers some great opportunities for the IT-industry. However, recent developments proved that this new kind of technology can be abused by malware that might turn out to be completely undetectable by antivirus software. Focusing on desktop system (although this is true for many servers as well) a clear domination of x86 processors can be seen. Nowadays, every newer x86 processor supports hardware virtualization and therefore virtualization malware becomes a serious risk. Evaluating the threat and developing possible counter-measures is therefore an absolute necessity for current and future computers' security. The question of the possibility of a perfect undetectable stealth rootkit, which did not exist in such a form prior to the availability of virtualization technology, is of additional academic interest.

Structure

Chapter 2 will give a short introduction into basic rootkit ideas, their goals and their shortcomings, followed by a description and analysis of hardware-assisted virtual machine (HVM) rootkits in general and the proof-of-concept rootkit Bluepill in special in chapter 3. Possible and impossible means of detecting such rootkits will be presented in chapter 4, while chapter 5 focuses on how to prevent such a rootkit's installation.

In appendix A, the necessary steps to be able to run Bluepill are described as well as the implementation of the hardware-virtualization detector that has been implemented within this thesis. Results gained by the detection methods analysed will be evaluated.

A good-founded pre-knowledge especially concerning system programming and virtualization-technology is required for the understanding of this thesis.

2 Prerequisites

This chapter is going to introduce basic rootkit concepts by classifying several types of malware, their techniques and their aims while illustrating them with well-known examples. The circle of these prerequisites is closed by pointing attention to the issue of targeted attacks, which will be of relevance on several occasions in the following chapters.

A rootkit is malware taking control over a computer system without proper authorization, usually installed after getting access to the target system and meant to preserve this access. This is accomplished by installing a backdoor allowing the attacker to reenter the system at a later time. A key feature is therefore the undetectability of such a backdoor.

Several basic techniques for “stealth” rootkits have been developed. However, as this chapter is going to point out, there are means for detecting each of those.

2.1 A rootkit taxonomy

[Rut06] presented a rootkit taxonomy for classifying the different types of rootkits from a “system compromise detection point-of-view”.

Generally speaking, a rootkit has to place a hook somewhere, so that at some point an application or the operating system would redirect control to code that is owned by the rootkit. In a modern operating system there are many places to hook. However, there are different types of hooks that can be generalized, serving as a classification for such malware.

2.1.1 Type 0 malware

Type-0-Malware comes with its own files or modifies existing files. The detection of such malware can easily be done through signature scans and integrity checks. Such methods are widely used and implemented for example in anti-virus software (signature scanning) or in Tripwire (integrity checking). Probably the most famous example of such malware is Thompson’s compiler hack [Tho84] that compiled malicious code into executables and would also compile this malicious code into the compiler when the compiler was build. Therefore even if a whole system was build from the bare sources, the compiler would still have installed the backdoor.

Also the most widely spread user-mode rootkits belong to this category. They would traditionally replace essential system binaries with their own versions presenting a false view to system administrators in order to hide the presence of processes, files and other data allowing for the detection of the rootkit.

2.1.2 Type 1 malware

Type-1-Malware modifies read-only i.e. constant data like code or static values in executables or kernel data (see figure 2.1). This is done by modifying code-sections and directly replacing

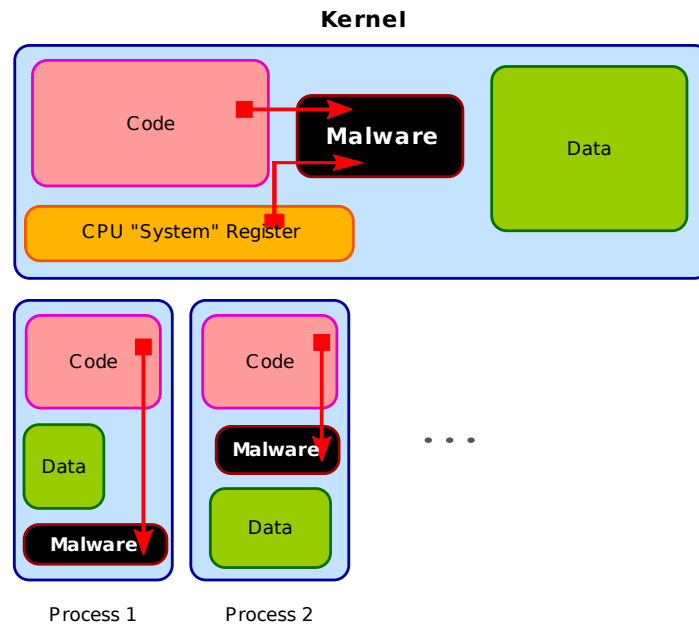


Figure 2.1: Type 1 malware [Rut06]

code (a technique known as *Inline Function Hooking*) or by modifying structures such as the import section of a binary (*Import Address Table (IAT) Hooking*). Using these techniques the control flow will be redirected to the malicious code. The equivalent of hooking the IAT in kernel mode would be to hook the *System Service Descriptor Table (SSDT)* which is a look-up table for all implemented, user-mode callable system functions. This influences every application. A typical rootkit would use these techniques to filter system API calls in order to hide for example a file, a process or a network-socket. A user-mode program would (assuming the rootkit is properly implemented) not be able to ever know about the rootkit's presence.

Integrity scans based on signed executables and hashes of code and static data could provide a wonderful way of verifying the program's state. However several more or less legitimate usages of run-time code modification prevent its application right now. Windows Kernel Patch Protection (PatchGuard [Mic]) tries to implement detection and locates modifications in the kernel space through integrity verification, but can be disabled with some effort by malicious software as shown by [MJ05] and again by [Joh06] after Microsoft tried to "harden" the protection. User-mode detection approaches try to get a low-level view of the system. If the rootkit is trying to hide a file, a detector would try to get raw disk access and parse the file system structures on its own without using the system's API.

One of the most popular hooking rootkits is *Hacker Defender* (<http://www.rootkit.com/project.php?id=5>) hiding files, directories, processes, services, ports et cetera. It is mainly a user-mode rootkit coming with some assistant code that runs in the kernel. The rootkit is well-known and its signature is included in standard antivirus software. Separate detectors and removal tools are available as well.

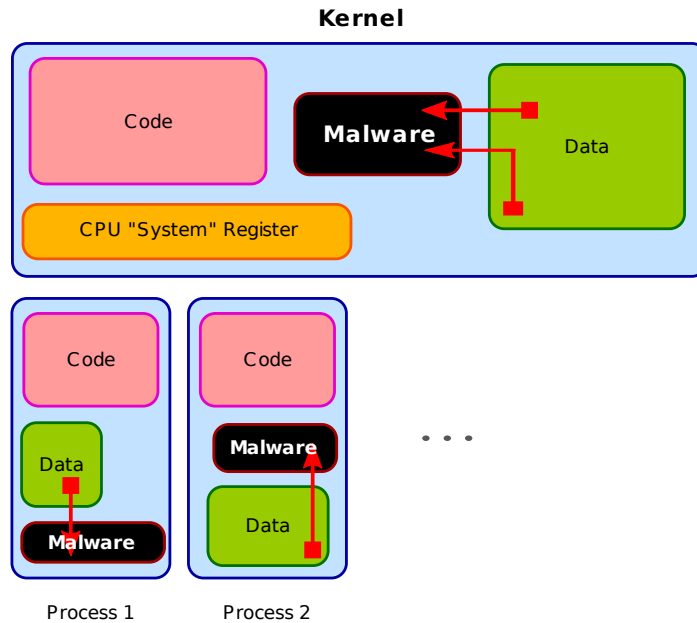


Figure 2.2: Type 2 malware [Rut06]

2.1.3 Type 2 malware

The flaw of type 1 malware is that it modifies constant data, i.e. data that is not supposed to be changed and thus the detection is straightforward. Consequently type 2 malware will only modify application's data that is supposed to change such as variables in memory (see figure 2.2). Such malware would modify certain in-memory data structures. If this technique is applied on the operating system's kernel, it is known as Direct Kernel Object Modification (DKOM). Since these data structures are often designed to be modified for legitimate uses within the operating system, a signature verification approach cannot be used to detect the unwanted modifications. However, known rootkits can still be found, since a detector would know where to look for which modifications. Also, a whole-system memory integrity scanner that knows about all data-structures that can be used for dynamic hooking, would need verify all those places and would then be able to detect the presence of such a rootkit. This is a rather theoretical assumption though, since such a scanner does not exist. However, imagining that operating systems would mark all those hooking places and produce such a tool, this would be a bad day for all type 2 malware.

The just mentioned detection works well unless the rootkit is able to prevent the scanner from scanning certain memory pages, which is exactly what the famous proof-of-concept rootkit **Shadow Walker** [SB05] does to achieve stealth. By installing a custom page fault handler and marking its own pages as non-present, it can allow execution access to the hidden page, but will provide a dummy page once read access is request. Unfortunately for the rootkit, this technique won't work for the page of the page fault handler itself and thus a modified page fault handler would be easily detectable and provide indication for malware presence on the system.

2.1.4 Type 3 malware

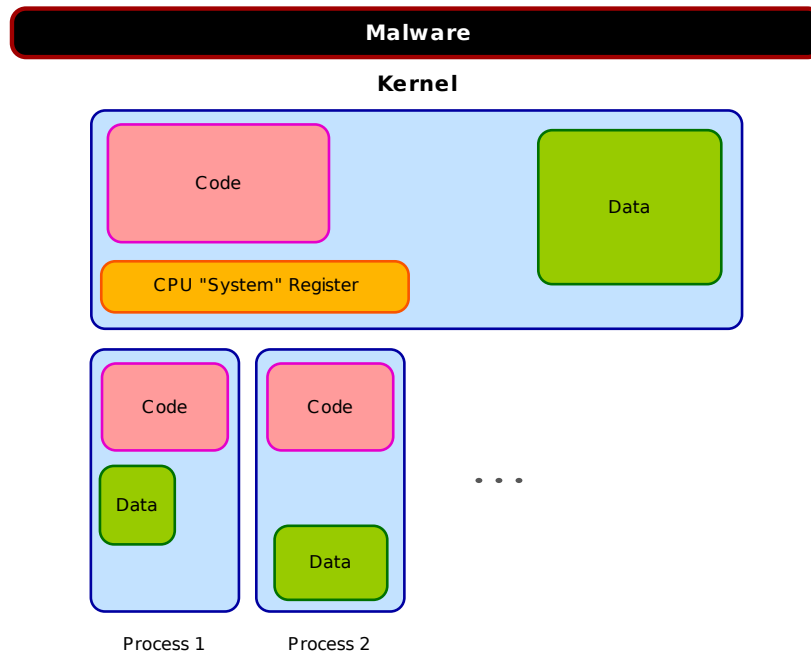


Figure 2.3: Type 3 malware [Rut06]

The logical consequence of type 1 and type 2 malware is to introduce type 3 malware which does not modify any CPU registers nor read-only nor modifiable data in the operating system (see figure 2.3). This would be the perfect malware as it seems that this kind of malware is undetectable. An example for such malware is virtualization based malware in all its forms. The next chapter is going to dive into hardware-virtualization based malware (HVM rootkits) such as Bluepill.

2.2 Mass rootkits vs. targeted attacks

Traditionally malware tried to reach as far as possible, infecting as many machines as possible. In recent years the primary motivations for malware were building bot nets for spamming or stealing personal information like credit card numbers or passwords for online games. These mass rootkits have the advantage, that antivirus vendors have easy access to samples of it which they can use to analyze and build signatures for detection and removal.

A very different aim have targeted attacks. Rather than trying to infect as many systems as possible, they *target* just a couple of machines. Sometimes there is even only one existing copy of the rootkit. Targeted malware therefore aims to spy for specific information on a very selected set of systems. They can be adjusted to bypass signature detection or can disable antivirus systems prior to launching their malicious routine. Antivirus vendors can usually not acquire samples of such malware, because the malware will not enter their honeypot systems. Also these attacks are seldom even noticed by users who could forward a sample of the rootkit to the antivirus vendor.

2.2 Mass rootkits vs. targeted attacks

Because operating systems and antivirus vendors generally verify the system's integrity by *enumerating badness* using signature scanning instead of *ensuring goodness*, targeted rootkits cannot easily be detected as long as their signature is unknown. As section 2.1.3 already concluded for type 2 malware, it is theoretically possible to protect even against targeted attacks. The arising question is therefore if type 3 malware is really undetectable or if it is possible to protect against mass rootkits or even against targeted attacks.

2 Prerequisites

3 HVM rootkits

Hypervisors run in an even more privileged mode than kernel-mode drivers that are running in ring 0 (while user-mode applications usually run in ring 0). This privileged mode for hypervisors is sometimes referred to as ring -1 and it allows for defining anything that would give clues about the rootkit's presence to be a more privileged instruction and thus makes it possible for the hypervisor to even trap ring 0 operations. This means it can theoretically intercept and cheat every aspect of guest operation.

3.1 Basic Principles

HVM rootkits are thin hypervisors making use of hardware-virtualization features of the CPU (i.e. AMD-V and Intel VT-x) and are therefore not flawed by the shortcomings software-based virtual machines have to deal with. Software-based virtualization does not allow for full virtualization, meaning that certain unprivileged instructions such as SIDT or SGDT (Store Global/Interrupt Descriptor Table Register) can be executed within the guest without the host being able to trap and emulate the execution thus allowing for straightforward virtualization-detection as described by [QS06, Rut04a, Kle]. In contrast to widely spread

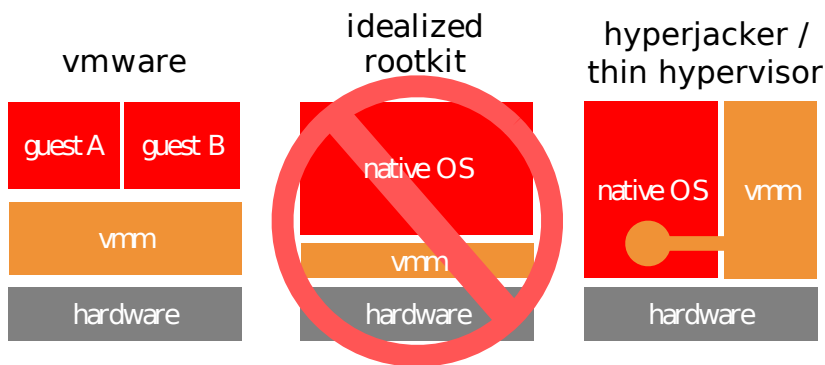


Figure 3.1: Differences between traditional and thin hypervisors (based upon [PLF07])

hypervisors such as VirtualBox or VMWare, a thin hypervisor (or a *hyperjacker* as [PLF07] calls it to distinguish from the traditional concept of a hypervisor, see figure 3.1) does not emulate a whole different hardware environment for the guest providing a strict isolation. There have however been some approaches in this direction, such as Microsoft's SubVirt [KWC⁺06] which is a persistent rootkit booting the guest operating system within a commercial hypervisor. But due to the absolute easiness of detecting such malware, this thesis is going to concentrate here solely on thin HVM rootkits which contrary to SubVirt's approach allow the guest to directly access nearly all aspects of the original hardware, most of the time without the hypervisor even trapping the access. This means the hypervisor and the

guest operating system share all except for very few resources and the performance impact of such hardware virtualization is not noticeable.

Hypervisor rootkits are non-permanent. They only reside within the computer's memory, thus uninstallation is as easy as rebooting the PC (another mean of disabling hypervisors is presented in section 4.4). The rootkit could possibly use other known techniques to become permanent. These features however remove stealth, as they have to place hooks for example in the filesystem again, and are consequently type 1 malware at best.

3.1.1 Implementing a thin hypervisor

The environment in which the virtual machine runs is defined by a control-structure specific to the virtualization extensions of the CPU. In this structure are among other information all registers that define the operations of the CPU. These are for example segment registers, the CR3 register (holding the base address of the page directory and being used for virtual memory) or the instruction pointer. General purpose and floating point registers are not included. The thin hypervisor would now clone all of the CPU's state information and put it in the virtual machine control structure/block (Intel: VMCS, AMD: VMCB). The operating system will be running outside the virtual machine at first, but when executing inside the virtual machine, the very exact same environment will be available to the operating system and as such it will work as it did before, until it executes an instruction that has to be trapped and handled by the hypervisor, which only happens very rarely.

3.1.2 Abusing hardware virtualization for malware

As described in the previous chapter, prior rootkits were flawed since they had to hook at some place in the operating system. An HVM rootkit is in this way fundamentally different, as it will not hook anything *inside* the operating system and its processor context. Instead it moves the operating system into the virtual machine, with all processor registers being stored in the virtual machine control block. Now the hypervisor can do hooking *outside* the virtual machine e.g. by setting one of the debug registers, but the guest operating system cannot notice this, since it only sees its own virtualized debug-registers. This means the hooking cannot be detected, unless there is a way to break out of the virtual machine (which is impossible by design unless AMD/Intel or the hypervisor made a mistake that would be fixed in future revisions). Because CPUs are very complex and computers even more, there are still means that allow for the detection of malicious hypervisors. This is going to be discussed in detail in chapter 4.

3.2 Technical Implementation Details

In order to better understand hardware-assisted virtualization malware, this section will depict the concept in a little more detail with focus on the technical implementation of a thin hypervisor. The description is here to give an idea and many things are simplified and involve complex procedures that are difficult to get right in a real implementation. For an in-depth reference, the corresponding manuals [AMD07a, Int07] should be consulted.

3.2.1 Moving the operating system into the virtual machine

Traditionally, hypervisors set up an isolated virtual machine, run the host inside and intercept every hardware access. Thin hypervisors use a different approach, as they do not want to interfere with most guest operations. The guest shall still be able to access all the hardware as it did before, there won't be any emulation, except for the very few instructions that have to be emulated. As described in section 3.1.1, this works by cloning the real en-

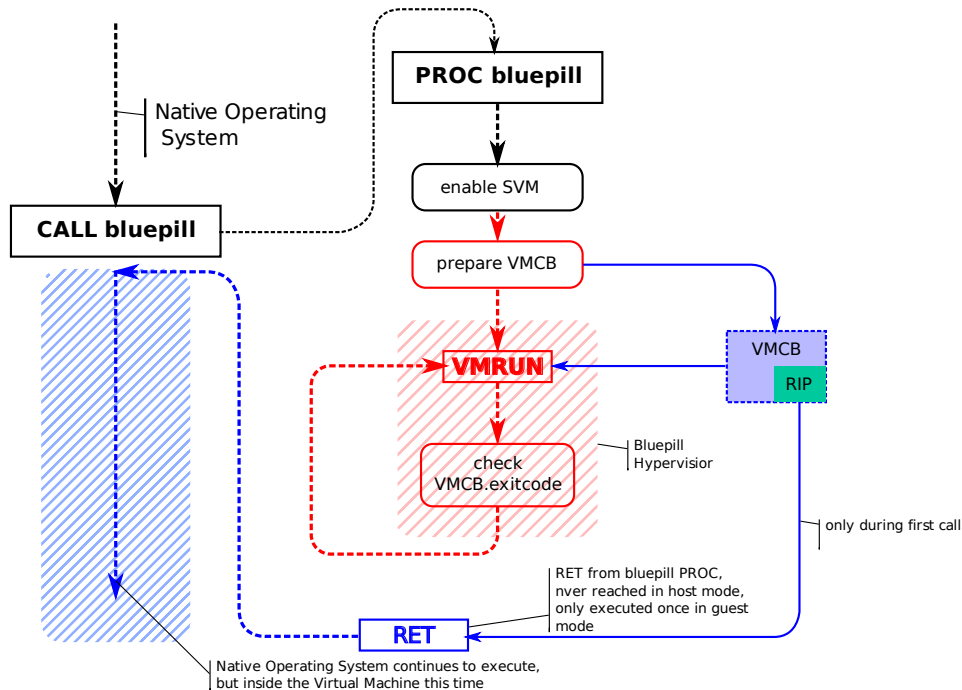


Figure 3.2: Illustration of the moving-into-VM process [Rut07b]

vironment and setting up the VM environment exactly alike. The environment of the guest is defined by its general purpose registers and the virtual machine control structure/block (Intel: VMCS, AMD: VMCB) which includes the processor state such as control registers (see [AMD07a]), entry and exit controls for defining the transition from host to guest, as well as the set of instructions to intercept and some other information.

Moving the operating system into the virtual machine happens in the following steps [AMD07a, Int07].

Preparing SVM/VTx operation. Before enabling virtualized operations, the process must be running in ring 0 which is the case for kernel-mode drivers. Next it needs to check for HVM support by running through `cpuid` (VT-x: `CPUID#0` must return bit 5 [Virtual Machine Extensions (VMX)] set in the ECX register, SVM: `CPUID#0x80000001` must return bit 2 [Secure Virtual Machine (SVM)] set in the ECX register) and then enable HVM operation by setting a bit in the extended feature register register (VT-x: `CR4.VMXE`, SVM: `EFER.SVME`). These bits are important, as they provide the

first approach for detecting HVM rootkits and the CR4/EFER register will often be referenced in chapter 4.

On Intel architecture, VMX operation needs to be explicitly entered by issuing a VMXON instruction and supplying a memory region for virtualized operations whereas for AMD-V an explicit host save area has to be supplied in the VM_HSAVE_PA register, which is a model-specific register (MSR) to support SVM operation.

Allocate and set up VMCB/VMCS. Each processor needs to have a memory region for storing host and guest data on transitions. As has just been described, these regions are set up in such a way that the guest operating system’s environment is cloned. For this to work the upcoming hypervisor has to issue a sequence of write instructions filling the region with the appropriate values. Furthermore the set of intercept conditions is defined. A minimal hypervisor needs only to intercept very few instructions (VT-x: CPUID, INVD, MOV from CR3 plus the ten VM* instructions to support VT-x operation; AMD-V: VMRUN only; [Int07, MY07]).

Prepare a #VMEXIT handler. Like interrupt handlers to which the CPU delegates control upon an occurring interrupt, the #VMEXIT handler handles guest-suspension events, which are caused by the execution of an instruction, that the hypervisor set up to be intercepted. The entry point for the handler is again specified in the VMCB/VMCS. After the handling of the event, the hypervisor reenters the virtual machine and redirects control again to the guest’s next instruction.

Switch to the guest. After setting up the environment, the #VMEXIT handler, and providing a guest entry point, the hypervisor will execute the guest-entering instruction (AMD-V: VMRUN, VT-x: VMLAUNCH) by which the control flow continues seamlessly in the native operating system without knowing that it is imprisoned within the hypervisor. Entering the guest is an event called #VMENTRY. The Intel manual [Int07] specifies 11 pages of checks on the VMCS on this event, thus entering the guest takes many CPU cycles and makes timing attacks an attractive detection approach, that will be discussed in section 4.1.1.

3.2.2 Further hypervisor tasks

As [MY07] pointed out, kernel-mode rootkits (type 1 or type 2 see chapter 2) are already “well-situated to hide from both user-mode programs and system-level security services” so the only real motivation for type 3 malware is the undetectability even under expert analysis. Therefore such a malicious hypervisor should implement certain features to avoid trivial detection. Such features begin with cheating about the “virtualization enabled”-bits in the EFER/CR4 register but continue into more advanced hiding-mechanisms such as time cheating, memory hiding and virtualization emulation. Means of detection and methods for the rootkit to counter them are going to be discussed in the following chapter 4. The most basic hypervisor-rootkits will not implement those features, the proof-of-concept rootkit Bluepill took the hard way though and implemented a couple of them.

As shown in the previous section 3.2.1, the hypervisor sets a flag in a special machine register (EFER/CR4) to enable the hardware-assisted virtualization functionality. Any in-guest software could now read the register and would know, that a hypervisor is likely to be running by checking the flag, therefore the hypervisor has to intercept access to this machine

register in order to cheat about this flag, which is the source of most of the implications of side-channel attacks discussed in section 4.1.

3.2.3 Details on Bluepill

For the analysis, the public version of Bluepill’s source-code from <http://bluepillproject.com> (version 0.32) was used. Bluepill implements the basic thin hypervisor features as described in section 3.2.1 within a generic framework supporting both Intel and AMD virtualization, though only the latter one has been tried during this thesis. In addition to the basic features, Bluepill supports nested virtualization for the AMD platform by emulating the CPU’s virtualization instructions and coping with several non-trivial implications (see also [Rut08, RT08]), thus allowing to run another hypervisor within Bluepill or even allowing to run other Bluepill instances.

RDTSB timestamp cheating (as will be presented in section 4.1.1) is not implemented anymore, probably because of the implications aroused from bigger differences in the time stamp counter (TSC) register (see empirical results in section A.4). The bare skeletons for RDTSB cheating through instruction tracing is still visible, however not functional. Due to the more effective counter-based detection (see section 4.1.4), trying to hide TSC delays does not add stealth anyways.

Virtual Memory

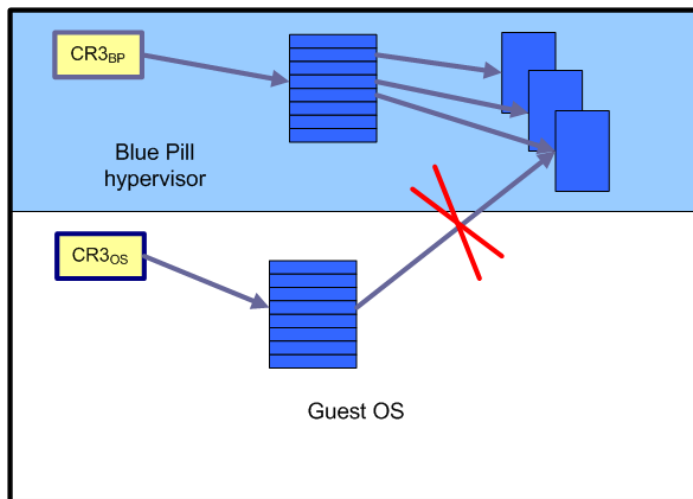


Figure 3.3: Bluepill private page tables [Rut07b]

In order to hide its own code from the operating system, Bluepill uses its own private page tables. This works since the hardware virtualization extensions allow for an own CR3-register storing the base address of the page directories. To still be able to function, Bluepill clones parts of the operating system’s page tables and is thus able to access the original kernel-space i.e. use the kernel API. Furthermore, Bluepill needs to make sure, that its pages cannot be accessed by the guest. This is according to [Rut07b] accomplished by patching the original page table entries (PTEs) to point to some garbage records (see figure 3.3). This mechanism

seems however not yet to be implemented in the publicly available version of Bluepill's source code. As a result, these versions can very well be detected through simple memory scanning.

Hypercalls

For debugging and demonstration-purposes, Bluepill provides an interface for calling the hypervisor from within the guest. Such a mechanism is known as *hypercalling*. In Bluepill's implementation it is currently only used for unloading the driver. The process is initiated in the guest and hypercalled to the hypervisor, who will then take the necessary actions to uninstall itself. A hypercall will execute a `cpuid`-instruction with a special magic parameter being subsequently intercepted and evaluated by the hypervisor. An advantage of using the unprivileged `cpuid` instruction for this task, is that it can be run in ring 3 i.e. in user-space by any program.

The `bpknock` program (see section A.2.1), which is used in debug environments to verify whether Bluepill is running or not, makes use of the hypercall principle, too.

4 On detecting HVM rootkits

Theoretically, a virtual machine should not know that it is being virtualized and in fact there is no direct and official way to find out for sure. However, a virtualized environment forces some behavioral changes and noticeable side-effects. These can be detected. Unfortunately though, the task of a detector is harder than detecting a running hypervisor. It has to distinguish between good and bad hypervisors and as such the explicit detection of HVM malware as described in second part of this chapter (section 4.2) is way more relevant than general virtualization detection that is described in the following section.

4.1 Virtualization detection

The hypervisor is in an advantage position. By design it is meant to simulate an environment to its guest, thus methods to cheat, ways to intercept instructions and hardware accesses are built-in. Accordingly the hypervisor can cheat about all obvious means of detecting a running hypervisor (see section 3.2.2). The remaining means of detection are side-channel attacks.

4.1.1 Timing Attacks

It has been pointed out, that the hypervisor should intercept access to the EFER/CR4 register in order to cheat about the SVM/VMX enabled bit (see section 3.2.2). Intercepting reads and writes will cause a `#VMEXIT` followed a `#VMENTRY` implying a delay (see section 3.2.1) in reading the EFER register, which is the base of several detection vectors.

Direct timing analysis by reading timestamp counters

A computer has many means of measuring time in different granularities. One high resolution timer is the timestamp counter (TSC) which is incremented with each CPU cycle. The register's resolution is high enough to provide an adequate mean of detecting a running hypervisor as section A.4 will show with some empirical data.

A lot of other timers like the HPET (high precision event timer), PIT (programmable interrupt timer), LAPIC (local advanced programmable interrupt timer), ACPI (Advanced Configuration and Power Interface coming with three PITs) [PLF07] provide other accurate enough time-sources. Such timers can be virtualized using interrupt interception [Rut07b] or I/O-port interception, but this has to be done explicitly by the rootkit.

As an example, measures to cheat the TSC approach are discussed in the following subsection.

RDTSC offset cheating

Since timing is such an important feature, both the AMD-V and the VT-x specification allow for easy modification of the timestamp counter (TSC) register of the guest machine. One

can even specify a constant offset in the VMCB/VMCS that is to be subtracted from the TSC after each hypervisor interception. In practice, this feature proved not to be reliable as a hypervisor stealth mechanism [MY07]. An HVM rootkit therefore needs to read the TSC register using the RDTSC (Read Timestamp Counter) instruction before and after any action and fix it manually, though it is not clear if this measure will produce a sufficient stealth-level regarding TSC-based detection methods. Furthermore, there can be special conditions, in which the counter is not reliable, such as when the CPU is changing speeds due to power-saving features [Int07] or when the RDTSC instruction is not executed linearized order [AMD07b].

Another way of acquiring the TSC value is to read it using the RDMSR 10 (Read Model Specific Register number 10) instruction. Usually the register is read using RDTSC (Read Timestamp Counter). [Rut07b] pointed out, that RDTSC and RDMSR 10 will produce inconsistent results and suggested to use instruction tracing to get a well approximation of the guest-side expected TCS value, which is a tough task considering the x86 instruction set. Eventually, this would allow the hypervisor to hide from RDTSC timing detection.

Indirect timing analysis using external clocks

Summarizing the previous paragraphs, it can be concluded, that local timers can not be trusted. Another timing detection approach tries to use clocks that can generally not be intercepted by the hypervisor. Time could for example be read from a network clock or an external stop watch. Unfortunately, these timer's resolutions are no more sufficient to detect a present hypervisor when only executing one hypervisor-intercepted instruction, because a couple of thousand CPU cycles is no noticeable time delay. Thus the detector needs to run many such instructions to get a noticeable time-delay. See listing 4.1 for an example on how such a detector could work:

Listing 4.1: Hypervisor detector using an external time source [Rut07b]

```
t1 = GetTimeViaNTP();
for (i = 0; i<BIGNUM; i++) RdMsr(EFER); //force #VMEXIT
t2 = GetTimeViaNTP();
if (t2 - t1 > tMax) printf ("Troubles!\n");
```

BlueChicken strategy to prevent detection through external clocks

There is no reason for any software to execute a long loop of RDMSR EFER instructions as it is required for the external clock based detection as presented in the previous paragraph. A malicious hypervisor can detect such long loops and once detected take measures to prevent the incurrance of timing differences. [Rut07b] proposed to uninstall the hypervisor for a moment. The RDMSR instructions can then be executed without any interception delay and the hypervisor which will reinstall itself through a timer after a very short time, would remain undetected. This strategy has been called BlueChicken and is also implemented in Bluepill.

However, this very behavior creates a fundamental vulnerability for the hypervisor, since in the moment it uninstalls, it is not working anymore and it is furthermore not hidden anymore and therefore at best type 2 malware (see section 2.1.3) having lost all its special stealth features. Although, [PLF07] mentions that the measure of uninstalling and hiding in the

kernel is appropriate, since kernel malware is much harder to detect. However this is about principle and if a measure to hide is downgrading to kernel-malware, the undetectability claim of type 3 malware just does not hold at this level.

In Bluepill’s implementation the uninstaller sets a timer to reinstall it. But assume that the antivirus will force the hypervisor to unload, disable interrupts for the moment and catch its handler before it can reinstall itself. Not only it would be known then, that the system is virtualized, but also that a malicious hypervisor had been running. This is a win-win situation for the detector. Either it detects that the system is being virtualized or it catches the rootkit in the act.

Other trusted timesources

Every computer comes with many pieces of hardware and most devices have one or more embedded timers [PLF07] with sufficient granularity additionally to the ones already mentioned in section 4.1.1. A hypervisor cannot cheat about these time-sources, otherwise it would have to intercept most device accesses and also implement tons of device driver code, possibly introducing vulnerabilities that a detector could use to detect such a hypervisor with even more ease. Also there will be the one or other device the hypervisor does not know about, so to sum up, the effort will not pay off. Following the assumption that the hypervisor does not implement device dependent timer interceptions, those provide a way of determining that the operating system runs in a virtualized environment. The caveat is however, that an antivirus vendor is not likely to make hardware-dependent products for virtualization detection. Even if it did, one could still produce a targeted version of the rootkit and implement a cheating mechanism for the special device the antivirus vendor is depending upon.

In summary, timing attacks cannot be cheated perfectly. And although most of them can possibly be cheated with enough effort, everything ends up as the usual rootkit cat-and-mouse game, so it is probably better for all parties to not implement timing based detection and counter-detection measures, but instead focus on other means of detecting an HVM rootkit.

4.1.2 CPU-specific behaviour

[Rut07b] and [PLF07] suggested to take advantage of CPU-specific bugs, of which all processors have at least some, although most of them cannot be used for the detection of a hypervisor. An example of a probably usable bug is the erratum 140 (see [AMD08]) explaining that the TSC_OFFSET which can be specified in the guest’s VMCB will only be added to the time stamp counter register (TSC) when it is via the RDTSC instruction, but not when it is read via RDMSR 10 (read model-specific register number 10). The erratum also shows a workaround of intercepting access to MSR 10 and to manually add the offset. The hypervisor has to implement this workaround, to not be vulnerable to detection using this very method. [MY07] pointed out, that the approach of relying “on processor model specific bugs rather than a generically applicable technique“ is unattractive, because it restricts detection to certain CPU models.

Other approaches in this direction include the use of undocumented CPU features such as undocumented MSRs and CSRs (model-specific control and status registers) [PLF07]. This is equally unattractive for a generic detection approach though.

4.1.3 Profiling CPU resource discrepancies

Yet another type of side-channel attacks derives from CPU-caching behaviour and the cache's modification after hypervisor interceptions. The general idea is, that a forced `#VMEXIT` leads to modifications in specially crafted CPU data-caches, because the hypervisor's code is being executed involuntarily tainting several caches. Rather than observing effects directly which is seldom possible, because caches are CPU-internal features not meant to be directly accessed, the effects have to be detected by side-channel analysis techniques such as timing analysis.

The use of CPU's performance event counters like the number of branch mispredictions which can directly be read through an I/O-port is undesired, since I/O-ports can easily be virtualized by the hypervisor [PLF07], who would intercept and fix these counters.

TLB Profiling

Another such CPU resource profiling method is TLB profiling. The Translation Lookaside Buffer (TLB) is used for caching physical addresses of virtual addresses. Usually the operating system's page tables have to be walked to get the corresponding physical address of a virtual address. If a TLB entry is present, this lookup is not necessary, thus access to memory is much faster.

TLB profiling works by filling all entries of the TLB and forcing a `#VMEXIT`. The hypervisor "must at least read the VMCB's exit code, requiring 1 data TLB entry" [Ada07]. The detector would then notice at least one modified TLB entry clearly indicating the hypervisor's presence. This is true for AMD-V having tagged TLB entries. Because the hypervisor and the guest can have the same virtual address each pointing to a different physical address the entries are tagged using an application specific identifier (ASID) to distinguish them. VT-x does not support ASIDs and therefore the TLB is flushed on each `#VMEXIT`/`#VMENTRY` thus invalidating *all* TLB entries. For this reason TLB based detection methods work better on Intel systems, because a bigger part of the cache is affected by a hypervisor interception.



Figure 4.1: Illustration of the L2 TLB organization [Rut07b]

Filling the TLB is not as trivial as accessing N different pages, instead it must be taken care of the multiple-way associativity. Each cache line can hold multiple values for the same index, uniquely identified by a tag. The index is the value of lower bits of the virtual page address (the lowest 12 bits for the page-offset are of no relevance for TLB operation and therefore discarded). The remaining address-prefix that is not included in the index-value is the tag, identifying which field of the cache line shall be used. A 4-way associative cache setup is illustrated in figure 4.1. The testing system’s processor (see appendix A.4) has two fully associative first-level (L1) TLBs (code and data) with 32 entries for 4kb-pages each. These are relatively easy to fill, but they are backed up by a second level (L2) TLB cache which is 4-way associative with 128 lines (512 entries). Additionally, the L2 TLB is shared between code and data.

Since the detector cannot just read the TLB, it has to use other means of *noticing* that a TLB entry has been modified. The easiest way is to access the same value on the page again and compare the timing to the previously measured access time, which has been implemented in a slightly modified form by Rutkowska and is included in the detection framework presented in appendix A. Unfortunately though, a malicious hypervisor can potentially cheat time measurements and consequently the following method, suggested by [Ada07], is preferred. In this approach after filling the TLB, the page table entry (PTE) for each page that

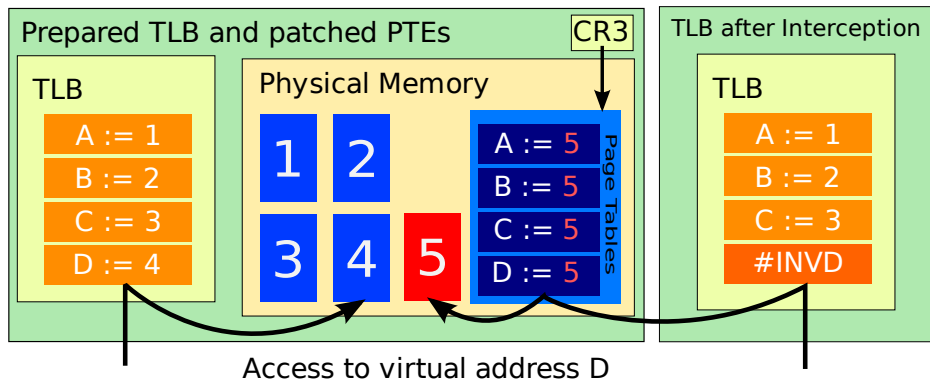


Figure 4.2: Different physical addresses returned after hypervisor interception

is assumed to be cached by the TLB is patched to a certain page with magic values prior to forcing the hypervisor to run. Afterwards, the detector tries to access each page. If it gets the old value, the TLB entry has still been present. But if it reads the magic value, the TLB has not been hit, thus the page had to be looked up in the PTE again (see figure 4.2 for an illustration). This way the detector can find out how many TLB entries have been modified while running the hypervisor. Pseudo code of this approach is also given in listing 4.2. This method is also implemented in the testing framework in appendix A.3.2, even though the results are not distinct enough to reliably assess a hypervisor’s presence (see section A.3.2 and A.4 for details).

A variation of this technique using the code TLB has been presented by [PLF07]. They also presented this method for the Intel platform with the advantage that #VMEXITs completely flush the TLB on VT-x due to the lack of ASIDs. Thus all entries are invalidated, so that the detector does not have to worry about second-level caches. The method can therefore ascertain the hypervisor’s presence.

In summary, TLB-based side-channel approaches seem to be a reliable method for hyper-

Listing 4.2: TLB Profiling [Rut07b] (spelling mistakes corrected)

```

for (i = 0; i < N; i++) page[i] = Alloc4kPage();
page22 = Alloc4kPage();
for (i = 0; i < N; i++) memset (page[i], 0x11, PAGE_SIZE);
memset (page22, 0x22, PAGE_SIZE); //page with "magic" values
for (i = 0; i < N; i++) x = page[i][0];
for (i = 0; i < N; i++) RemapPTEForPage(page[i], PhysAddr(page22));

RDMSR EFER // force #VMEXIT

for (i = 0; i < N; i++) if (page[i][0] == 0x22) break;
if (i < N) printf ("VMM might be in the system!\n");

```

visor detection in non-virtualized environments.

[Rut07b] also suggested a counter-detection measure based on Nested Pages / Shadow Page Tables (the hypervisor is basically notified when the guest modifies its page-tables), noticing a massive PTE patching and using BlueChicken (this was described in section 4.1.1) to uninstall for the moment.

This approach does not seem to be thought throughly, since a generated page-fault (#PF) when the detector writes the PTE will destroy some TLB mappings anyway, thus there should be no difference in detecting the hypervisor. Additionally the BlueChicken method yields the same problems as when used for time cheating (see section 4.1.1).

Cache Invalidation

The INVD (Invalidate Internal Caches) instruction flushes the cache discarding queued writes. Thus modifications in cache that were not yet written memory are lost. To use this method as a detection approach [PLF07] proposed to fill the cache by writing to memory, force a #VMEXIT and flush out the whole cache by executing an INVD immediately afterwards. If the modifications indeed hit memory, a hypervisor is likely to be present, because otherwise no data would have been written back to memory.

When implementing this detection method, additional care needs to be taken again due to the CPU-dependent complexity of multiple-level and multiple-way associative caches.

Return Stack Buffer (RSB)

A recent presentation [Bul08a] implemented a detection approach based on the impact of a #VMEXIT on a CPU's Return Stack Buffer which is an "internal hardware 'stack' in CPU" predicting the "target address of RET instruction before it is available from memory". This stack typically has 16 entries and mispredictions have a timing penalty. The detection approach is based on filling the stack using 16 nested call-instructions, forcing a #VMEXIT and checking the timing after all functions returned (see illustration in listing 4.3). Since the hypervisor is likely to replace RSB entries, at least some of the `rets` will miss the cache and thus take longer execution time. If the measured time exceeds a certain treshhold, a hypervisor is present.

Listing 4.3: RSB detection (RSB spy by [Bul08a])

```

func15() {
    cpuid ; #VMEXIT on VT
    rdtsc ; start measurement
    ret   ; start 16 returns
}
func14() {
    call func15
    ret
}
..
func0() {
    call func1
    ret
}
spy() {
    cli
    call func
    rdtsc ; end measurement
    sti
}

```

4.1.4 Counter-based detection

The counter based attack (first presented by [Bar07]) is an easy straightforward attack to detect the presence of a hypervisor. However, it requires at least two processor cores, since it will run a counter on one of them while the other core forces a `#VMEXIT` by executing an instruction that the hypervisor will intercept such as `RDMSR EFER` (see section 3.2.2) for AMD-V or `CPUID` for VT-x. If the counter exceeds a certain value, it can be assumed that the system is virtualized (see figure 4.3 for an illustration). The wonderful thing about this attack-vector is, that the counter will exceed the limit while the other processor is still context-switching to the hypervisor, which means as soon as the hypervisor intercepts the instruction, it is already too late for anything the hypervisor could possibly do. This attack is implemented in the test suite provided with this thesis (see section A.3.1). There is no known way to cheat this attack, unless the hypervisor takes insane measures such as instruction scanning which provides even more attack vectors and equally leads to the hypervisor's detection.

4.1.5 Side Channel Attacks with nested virtualization

Side channel attacks as described in the previous sections will not straightforwardly work, if the system is already running in a hypervisor like Xen.

If the guest-system is para-virtualized, executing instructions such as `VMRUN` will not cause a `#VMEXIT` but a `#GP` (General Protection Fault) since the guest is running in ring 3, thus the impact on the guest is *zero* overhead instructions if running on an AMD processor, because the hypervisor did not even intercept anything. On Intel processors there are obligatory intercepts like `CPUID`, `INVD` or `MOV CR3` [RT08] introducing timing differences.

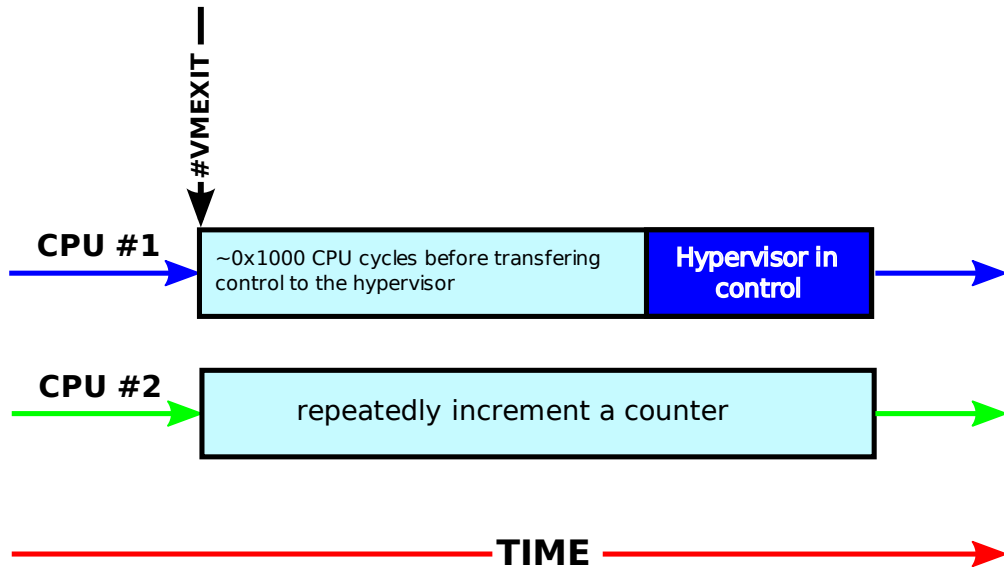


Figure 4.3: Illustration of counter-based attack. [Rut07b]

For HVM guests, in nested virtualization implementations the top-most hypervisor needs to execute another VMRUN instruction (see [Rut08] or [RT08] for details on nested virtualization) and thus introduces a noticeable timing difference ranging from 2.000 cycles to 24.000 cycles compared to the native Xen baseline of a #VMEXIT-time of 5.000 cycles [RT08]. Still though, timing based detections should not be trusted too much, as RDTSC cheating (see section 4.1.1) is still a possible counter-measure.

Cache modification based detection will not work in guest domain, because Xen will intercept privileged instructions anyways and thus already introduce changes in the CPU's caches. It has not been assessed if the additional interception of a malicious hypervisor might introduce enough changes to distinguish if an instruction has been intercepted by one or two hypervisors. Considering for example the TLB profiling method (see section 4.1.3) on the Intel platform, there is no noticeable difference, since both hypervisors completely flush the TLB.

Other side-channel attacks will only work in the hypervisor domain on instructions the HVM rootkit intercepts.

4.2 Explicitly detecting a malicious hypervisor

In the previous chapter 4.1 ways of detecting virtualization were discussed in detail. Still yet, this detection approach can not distinguish between good and bad hypervisors at any level. Knowing that the operating system is virtualized may be enough to know that something is not right (i.e. an HVM rootkit is running) in special situations when you do not expect to be in a virtualized environment. Unfortunately though, with growing importance of virtualization technology even on desktop systems, it is not uncommon that an operating system is running within a hypervisor and therefore the information gained in the previous section will be worthless in many situations. It is naive to restrict a detector to unvirtualized machines, thus explicitly detecting the malicious hypervisor is of even greater importance.

Furthermore should be noted that the hypervisor should not prevent virtualization detection, otherwise it can be easily found. This is because a detector finding out that it resides in a virtualized environment, would look for discrepancies like RDTSC cheating or a BlueChicken implementation (see section 4.1.1). If such a measure is found, the detector can be relatively sure, that malware is running, since there seems to be no reason a legitimate hypervisor would implement these techniques.

4.2.1 Signature Detection

An ideal malicious hypervisor would use its own pagetables and would therefore be disconnected from the virtual address space of the guest OS (see the illustration in figure 3.3 on page 13). Consequently the guest has no clue of the existence of the hypervisor within its virtual address space.

In order to scan for a hypervisor's signature however, the detector needs first of all to be able to read memory that is invisible to it due to the disconnected address-spaces. Thus to actually access the hypervisor-code, the guest has to somehow get raw access to memory and there are basically two approaches to do that:

PTE patching The easiest method to access a raw page, is to modify a page table entry (PTE) for a certain virtual address to point to the desired physical address. Then the detector can access and scan the page for a known signature. The hypervisor can defeat this by using *Shadow Page Tables* that trigger a page fault (#PF) on write access and allow the hypervisor to monitor access to the guest's page tables. Rather than using Shadow Page Tables, which are according to [Rut07b] difficult to implement correctly, one can use *Nested Page Tables* (introduced into AMD-V with AMD Barcelona) which are a much easier to implement hardware-assisted solution for Shadow Page Tables that can effectively hinder the guest from patching PTEs to hypervisor code pages.

Hardware-based acquisition will be discussed in section 4.2.2.

Even when getting raw memory access, the scanner is still in the disadvantage of seeing non-connected scattered pages in the physical address space. Scanning for a given code signature is possible, but obfuscation techniques such as polymorphic code can be used by targeted attacks to avoid signature detection and it is generally known that there is no program that can verify what another program does. It should be pointed out though, that techniques to avoid signature detection can be used by classical rootkits, too. The fundamental difference however is, that classical rootkits could theoretically be knocked out

by formal verification methods of the operating system's state, which does not work for their HVM siblings.

Instead of scanning for code signatures, a scanner can look for the signature of the VM-CB/VMCS, which cannot be obfuscated very well. This has the additional advantage of now having access to all information the hypervisor has, such as its CR3 register value allowing to access its page tables and seeing its virtual memory non-scattered anymore. Also the entry point for the `#VMEXIT` handler is provided by the VMCB/VMCS.

Raw memory access is not directly possible if the rootkit implements techniques to prevent it and even when being able to access raw memory, the scanner still has to find the code and distinguish between good and bad hypervisors. For all that, HVM rootkits are in the game again.

4.2.2 Hardware-based memory acquisitions

As described in the previous section, the OS can trick software-based solutions for raw memory access (and this is good, otherwise real hypervisors would be in trouble), so direct hardware-access to memory is the best bet for detecting a malicious hypervisor.

Direct Memory Access (DMA) allows devices to access system memory without the CPU's intervention. In the past years DMA based memory acquisition got a lot of attention in computer forensics, especially because of the FireWire bus' capability for DMA [BDK05] easing memory dumps to a new level. If FireWire is not available, other DMA capable PCI devices can be used. Dedicated devices for this task like Tribble or CoPilot by Komoku have been developed, but no such device seems to be publicly available. According to [PLF07] other PCI devices can be abused to read memory for the detector. They also claim that the GART (Graphics Address Remapping Table), which is the graphic card memory mapping unit, can be used for this task.

In the near future however, these methods will no longer work, because with AMD's IOMMU (input/output memory management unit) and Intel's VT-d (Virtualization Technology for Directed I/O), new CPUs will allow for memory mapping of DMA. This is important for virtualization because virtualized machines have usually no direct access to physical memory and are likewise not capable of using DMA devices. These technologies bring back the control to the hypervisor, sadly enough rendering another method of HVM rootkit detection impossible.

Additionally there are methods for defeating hardware-based memory acquisitions by modifying the North Bridge's memory mapped I/O address dispatch table [Rut07a]. With this technique the CPU still has access to the memory while all other devices are redirected to garbage records or can even be used to crash the machine, when someone tries to access raw memory.

In summary, neither software-based nor hardware-based RAM acquisition can be trusted. Although virtualization can be detected, a properly implemented HVM rootkit can defeat *all feasible* methods of memory acquisition (exceptions are discussed in the following section 4.2.3). This is not only a problem for detection: Even if it is known that such a rootkit is running on the system, there is no way of getting its code-base after it has been loaded, thus the rootkit cannot be analyzed and signatures cannot be developed. This is probably the root for the undetectability claims that have been made for HVM malware. With some preparation though, acquiring the malware's codebase could still be accomplished using some

ideas discussed in section 6.4).

4.2.3 “Insane Detection of Insane Rootkits”

A rootkit can do its best, to hide its code and to defeat anyone from accessing it, but it can never succeed completely. One can for example use crafted memory controllers that explicitly allow for an external acquisition. But one does not have to build special hardware to detect a rootkit in a system because there is already enough hardware inside: Very recently [Bul08b] presented a detection approach based on programming embedded micro controllers in the chipset again using DMA to get raw access to memory to scan for the rootkit. Contrary to methods using PCI hardware the chipset firmware is able to access the DMA remapping unit and disable it for the detection program. It was demonstrated, that a hypervisor with a known signature can be found and even be disabled (removal of hypervisors will be discussed in section 4.4) using this technique. The caveat however is, that the software needs to be chipset specific getting even closer to the hardware than the hypervisor and has yet only been presented for the Intel platform.

A similar (yet less insane) position to place a detector is the System Management Mode (SMM) which “is a relatively obscure mode on Intel processors used for low-level hardware control” [ESZ08], has direct memory access, is relatively invisible to the operating system and is thus a perfect place for a detector to scan the system’s memory. On a sidenote, SMM is also very attractive for rootkit developers due to associated stealth features and [ESZ08] already presented such a SMM-based proof-of-concept rootkit. However, SMM rootkits are really outside the scope of this document.

Eventually though, using very low-level hardware features, detectors *can* acquire raw memory, thus the arms race continues.

4.3 Passive detection

For the sake of completeness, be it mentioned, that network traffic introduced by any rootkit with network functionality (which usually is desired), can be detected. However, secret, undetectable covert-channels provide a relatively easy way to hide (see [Rut04b] for details).

4.4 Removal of a malicious hypervisor

One step beyond detection the question arises on how to get rid of the rootkit. A simple method is to reboot the computer and unless the rootkit has some permanence features, it will be gone immediately.

Another method could be to read the VMEXIT entry point address from the VMCB/VMCS and patch the VMEXIT handler in such a way, that it restores the guest state in the hypervisor privilege level, effectively pulling the native operating system out of the virtual machine. The rootkit’s code and data will still be present in system memory, but will not be executed again.

5 On preventing HVM rootkits

The previous chapter focussed on the detection of HVM rootkits. The 100% undetectability claim does not hold, however detection of such rootkits is very hard or nearly impossible once it installed itself as a hypervisor. Therefore it becomes important to focus on how it is possible to prevent these rootkits from installing and becoming a hypervisor in the first place.

Windows Vista 64 requires all kernel-mode drivers to be digitally signed, thus no untrusted code can be loaded. Except not quite, because malware can exploit trusted buggy drivers as has been illustrated for example by [Rut07b]. Consequently the protection does not hinder from getting kernel-mode access and installing a hypervisor.

Consequently, the basic idea of preventing HVM rootkits is the restriction of CPU's virtualization extensions to authorized software. There are several ways to do that.

5.1 Hardware supported authorization

Since virtualization needs to be explicitly enabled on the hardware, the hardware can be the first line of defense against the installation of an untrusted hypervisor.

5.1.1 BIOS support

Several BIOS allow to disable virtualization extensions. For example on the Intel platform, the CR4.VMXE (AMD: EFER.SVME) bit must be set to 1 prior to enabling virtualization extensions using VMXON (see section 3.2.1 for details on enabling virtualization). However, the bit cannot be modified anymore once the LOCK bit is set to 0 in the IA32_FEATURE_CONTROL (AMD: VM_CR) model specific register (MSR), thus the BIOS can set CR4.VMXE=0 and LOCK=0 to disallow the usage of virtualization extensions, which makes it impossible for a HVM rootkit to install, but also disables legitimate usages of virtualization. AMD-V comes with a similar feature, described in the next section.

5.1.2 Unlockable virtualization

In addition to the basic disable and lock feature, which has been described in section 5.1.1, AMD-V allows furthermore to re-enable virtualization extensions using a 64-bit key even if the LOCK bit is set (see section 15.29 in [AMD07a]). Therefore, the user would need to enter the key for the valid hypervisor who can then re-enable SVM and install itself, while hindering a malicious hypervisor from doing so because of the missing key.

5.1.3 TPM support

With Intel's Trusted Execution Technology (TXT, formerly LaGrande) and AMD's SVM (Presidio) both vendors introduce a trusted computing environment with hardware support

through a Trusted Platform Module (TPM) with instructions (Intel: GETSEC[SENTER], AMD: SKINIT) that can effectively reboot the CPU into a trusted state using a TPM verified secure loader. For example Intel has a flag in the IA32_FEATURE_CONTROL MSR that allows the execution of VMXON (to enable virtualization extensions) only in SMX (Safer Mode Extensions) operation and causes a general protection fault (#GF) otherwise [Int07]. SMX operation can be entered through the just mentioned GETSEC[SENTER] instruction, thus effectively restricting virtualization extensions to TPM verified software.

These approaches may be combined with the installation of a trusted hypervisor, which is going to be described in the next section.

5.2 Installing a trusted hypervisor first

An easy to implement idea of protecting against installation of a malicious hypervisor, is to install a *trusted hypervisor* first. Once it is running a HVM rootkit is effectively hindered from installing itself, because there can only be one hypervisor in the system. Unfortunately

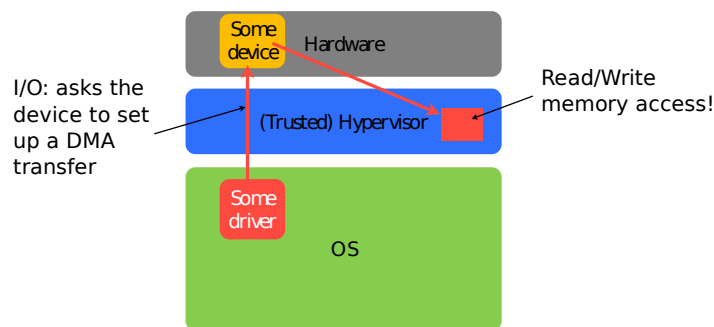


Figure 5.1: DMA flaw in trusted hypervisors [Rut08]

for this technique, without proper IOMMU/VT-d support, the rootkit could still use the same techniques that were earlier proposed to detect a running hypervisor (see section 4.2.1) to render the trusted hypervisor useless. Using a DMA transfer-channel with a device whose operation is not virtualized, the rootkit can write on the hypervisor's pages in memory and take over control (see figure 5.1).

However, since IOMMU/VT-d technology is upcoming (the first desktop systems with IOMMU support were according to [RW08] available around October 2007), this won't be a problem in the future.

6 Conclusions and Prospective

In the previous chapters rootkit ideas were introduced (chapter 2) to give a small overview of the terrain this thesis is dealing with. Hypervisor rootkits and their implementation were presented (chapter 3) and provide the required technical background for understanding and evaluating the technology on which base numerous detection approaches that were suggested in the last two years could be explained and evaluated (chapter 4). Also the difference between virtualization detection (section 4.1) and explicit rootkit detection (section 4.2) was pointed out, concluding that detection can only happen using reliable memory access methods. Technologies and ideas to hinder a rootkit from installing itself as a hypervisor were eventually discussed (chapter 5), giving a surround image of all aspects of HVM rootkits allowing for a knowledge founded evaluation of the technologies' threat.

6.1 Threat-Evaluation

As HVM rootkits have not yet been seen in the wild in real attacks (besides some proof-of-concept reference implementations), their impact is still hard to assess. While the technology is frightening, it is equally unlikely to have a working fully-armed HVM rootkit implementation with complete protection against memory acquisition due to the hardware-dependent issues that need to be taken care of. Building such a rootkit is however possible and the enemy should never be underestimated. With these prerequisites, HVM rootkit prevention becomes important.

It is very well possible to detect a virtualized system (i.e. a running hypervisor). All the presented techniques for detecting and hiding from detections basically fall back to the usual cat and mouse game that can be observed with traditional rootkits, too, with the hypervisor still being in the disadvantage since the proposed detection approaches require a lot of effort to be cheated. Hence we can conclude, that detection of virtualization works fine, if the system is not to be virtualized. In contrast, as has been shown in the previous chapter (section 4.2), it is the detector who is in a disadvantage situation once the system is already supposed to be virtualized.

Since reference implementations for basic HVM rootkits already exist for both the AMD and the Intel platform, the likelihood of basic virtualization-based rootkits increases, especially because no detection techniques have yet been incorporated into antivirus software. The hardware-dependence is still a huge drawback though, as those rootkits only run on newer AMD or Intel processors. Also the non-permanence of these rootkits is unattractive for the infection of user-desktop systems.

Furthermore should be noted, that there is no real need for type 3 malware right now, because formal integrity verification procedures are hard to implement and do not yet exist. Therefore targeted attacks of non type 3 malware have still no troubles with infecting systems and avoiding detection.

6.2 Undetectable Stealth Rootkits

One of this thesis' motivations is the question of the potential possibility of writing completely undetectable malware. Fortunately, nothing is 100% undetectable. Everything leaves traces and with the right tools, hardware can be disassembled and monitored. Nevertheless such measures are usually not feasible. Thus the question remains, how high the bar for detecting malware can be raised. One can imagine several types of type 3 malware of which HVM rootkits discussed in this thesis are just one. Side-channel attacks and lower level hardware allow for their detection. However future rootkits could use the very places that are used for detecting HVM rootkits, to hide themselves or trick detectors.

While the classical detection approaches do not work anymore for such malware, the rootkits' arms race will certainly continue with rootkits finding new places to hide (as happened recently with Intel's System Management Mode [ESZ08]) and antivirus vendors investigating new ways to track them down.

6.3 Applications for thin hypervisors

Besides the virtualization of machines and the abuse of hardware virtualization for malware, there is still a great potential for other uses based on such thin hypervisors.

Analysing traditional malware can be a very intricate task if the malware uses anti-debugging techniques of which there are very powerful ones, defeating local debuggers and reverse engineering tools with ease. While virtual machine debuggers that can bypass some of the measures already exist, undetectable hypervisor-assisted debuggers could be more powerful and effective especially since para-virtualization can easily be detected and be implemented into anti-debugging techniques.

The idea of trusted hypervisors (see section 5.2) can be extended to trusted hypervisors implementing nested virtualization, which can furthermore be a powerful tool for monitoring system resources and setting up honeypots for HVM rootkits. If more of these rootkits are seen in the wild, such systems will provide a handy environment for analysis and acquirement of the malware's code-base.

6.4 Further tasks / research

In chapter 4.1 several approaches on detecting virtualization were presented, while only some of them come with implementations. Since most of them are based on similar concepts there is no need to implement all these methods, especially because they only allow for virtualization detection and not for explicit detection of a malicious hypervisor. A much more useful investment of time and resources are the direct detection approaches, like a signature scanner based on patching page table entries (as described in section 4.2.1), which raise the bar for implementing such rootkits and make them more unattractive.

Current research on HVM malware focused solely on the x86 processors. Other architectures (such as SPARC) also have support for hardware virtualization, thus the threat of virtualization malware needs to be evaluated on these architectures, too, although the impact is certainly significantly less than on x86 due to the widespread dominance of x86 processors.

Furthermore is the concluding tenor of this thesis the need for an effective way of prevention. If a CPU features virtualization extensions, a trusted hypervisor should be running to prevent infection with HVM malware. If the system is already running in a hypervisor, it needs to be taken care of verifying the hypervisors integrity. Projects like HyperGuard [RW08] that run in System Management Mode (and should therefore be tamper-proof), aim to verify that there is no untrusted code in the hypervisor. The chipset approach (see section 4.2.3) could be of additional assistance for this task.

Theoretically, the techniques exist. They just need to be properly implemented and employed, so that no one has to fear virtualization technology.

A Empirical Results

In order to evaluate Bluepill, implement and test detection methods and conduct own measurements, a testing system has been set up. This practical part of the thesis describes necessary steps and actions used to prepare the testing machine for running Bluepill. In section A.3 the implementation of the detection methods implemented in the testing framework is described. Section A.4 analyses several tests that were conducted evaluating those detection methods.

All material associated with this thesis including source code for the testing framework is available online at:

<http://www.nm.ifi.lmu.de/pub/Fopras/frit08/>

A.1 Preparing Windows for home-brew drivers

Windows Vista 64 requires all drivers to be digitally signed with Microsoft-trusted certificates. This makes it hard to run own kernel-mode code in the first place, but using hardware-virtualization requires ring 0 privileges and thus needs to be run in kernel-mode. [RT06] proposed to abuse buggy kernel-drivers to load some shellcode into ring 0 due to the fact that there would be no sense for a malicious software author to obtain a certificate from Microsoft. However, for the scope of the document it is only required to run the drivers on the local machine and therefore an unsigned certificate can be used.

For this to work, a certificate needs to be created and added to the trusted publisher and root store [Chi] as shown in listing A.1.

Listing A.1: Adding a trusted self-signed certificate

```
makecert -sr localMachine -ss PrivateCertStore -n CN=BLUEPILL \  
    testcert.cer  
  
certmgr /add /c /s /r localMachine CA /n "Root Agency" /s /r \  
    localMachine root  
  
certmgr -add testcert.cer -s -r localMachine trustedpublisher
```

This certificate can now be used to sign drivers with the command shown in listing A.2.

Listing A.2: Signing a driver with a self-signed certificate

```
SignTool sign /v /s PrivateCertStore /n BLUEPILL /t \  
    http://timestamp.verisign.com/scripts/timestamp.dll driver.sys
```

However, Windows Vista 64 will even refuse to load these self-signed drivers unless special boot flags are set. Therefore TestSigning needs to be enabled and integrity checks need to be disabled using BCDedit as shown in listing A.3.

Listing A.3: Enabling TestSigning und disabling integrity checks

```
bcdedit /set TestSigning on
bcdedit /set nointegritychecks on
```

In order to verify, that Windows in fact allows loading of own code now, a simple Hello World driver is created. This is illustrated in the following listings A.4, A.5 and A.6. The Windows Driver Kit (WDK) needs to be obtained through obscure Microsoft websites prior to being able to compile this demo.

Listing A.4: hello.c – The Hello World Driver Source

```
#include <ntddk.h>
NTSTATUS NTAPI DriverUnload(IN PDRIVER_OBJECT DriverObject)
{
    DbgPrint("Driver unloading\n");
    return STATUS_SUCCESS;
}
NTSTATUS DriverEntry(PDRIVER_OBJECT DriverObject,
                   PUNICODE_STRING RegistryPath)
{
    DbgPrint("Hello, World\n");
    return STATUS_SUCCESS;
}
```

Listing A.5: makefile – The Hello World Driver Makefile

```
!INCLUDE $(NTMAKEENV)\makefile.def
```

Listing A.6: sources – The Hello World Driver sources-file

```
TARGETNAME = hello
TARGETPATH = obj
TARGETTYPE = DRIVER

INCLUDES   = %BUILD%\inc
LIBS       = %BUILD%\lib

SOURCES    = hello.c
```

The driver can now be built using the `build`-command from the WDK which can be accessed through the “Windows Vista and Windows Server Longhorn x64 Checked Build Environment”. The driver needs to be signed afterwards as shown in listing A.2. Since Windows has no `modprobe`-like utility, a third party driver loader has to be used. Here the tool `w2k_load.exe` from the `w2k-internals` package was used. The `DbgPrint` statements can now be observed through some debugging tool such as `Debugview` from `SysInternals`¹.

A.2 Running Bluepill

The Bluepill source can be obtained on <http://bluepillproject.com>. Once unpacked the tool can be build using the `build`-command in the build environment and needs to be signed

¹<http://technet.microsoft.com/en-us/sysinternals/bb896647.aspx>

again (see listing A.2). For debugging-purposes Bluepill uses a shared memory region with a debug-client driver, which means that in order to see Bluepill’s debug messages this driver has to be compiled, signed and loaded too. Loading works again using the `w2k_load`-utility.

Once Bluepill loads, all the hypervisor-magic happens silently and unless the Debugview program is running, there is exactly nothing to observe.

A.2.1 Verifying Bluepill’s presence in a debug environment

To verify that Bluepill is running, one can:

Use the Debugview program. It will show Bluepill’s DbgPrints if Bluepill is configured to use the debug client.

Knock at Bluepill’s door. For this, the Bluepill-package includes a tool named `bpknock` which executes `cpuid` with a special magic parameter. If Bluepill is indeed running (and the knock-command has been compiled into Bluepill) a special magic value will be returned. These values are defined in `nbp-0.32-public/common/common.h`. Listings A.7 and A.8 illustrate this call.

Listing A.7: `bpknock` (Bluepill running)

```
$ bin/i386/bpknock.exe 0xbabecafe
knock answer: 0x69696969
```

Listing A.8: `bpknock` (Bluepill not running)

```
$ bin/i386/bpknock.exe 0xbabecafe
knock answer: 0
```

View memory pool usage. During this thesis’ analysis of Bluepill’s source code, it has been found out, that Bluepill allocates its pool memory through Windows memory allocation API with a special tag. Therefore the `poolmon`-utility can be used to monitor this usage. The tag-value (named `ITL_TAG` defaulting to `'LTI'`) is again defined in `nbp-0.32-public/common/common.h`. For monitoring and analysis reasons Bluepill was modified to use different tags for different types of memory allocations. The tag `'BLUE'` is used for general memory, `'BLUP'` for allocated 4kb-pages and `'BLPB'` for bigger page allocations. This produces the following results when Bluepill is running:

Listing A.9: `poolmon -iBLUE -iBLUP -iBLPB`

Tag	Type	Allocs	Frees	Diff	Bytes
BLPB	Nonp	27 (27)	0 (0)	27	249856 (249856)
BLUE	Nonp	188 (188)	0 (0)	188	15040 (15040)
BLUP	Nonp	109 (109)	0 (0)	109	446464 (446464)

There are 188 small buffers in use. 109 4kb pages plus 27 bigger pages were allocated while cloning the operating system’s page tables.

Detect Bluepill for example with one of the detection tools provided in the next section.

A.3 Own Developments

In order to ease scanning the system, a testing framework for several detection methods was implemented within this thesis. This framework is a simple driver exposing the detection mechanisms through an IOCTL-interface, with the IOCTLs as listed in figure A.1. [One03] has been a great help in implementing this.

IOCTL_CODE	Value	Function
IOCTL_BLUEPILL_COUNTER_RDMSR	0x222002	Counter-based detection using RDMSR EFER
IOCTL_BLUEPILL_COUNTER_CPUID	0x222006	Counter-based detection using CPUID
IOCTL_BLUEPILL_TIMING_RDTSC	0x222042	Timing-based detection using RDTSC
IOCTL_BLUEPILL_TIMING_RDTSCP	0x222046	Timing-based detection using RDTSCP
IOCTL_BLUEPILL_TIMING_MSR10	0x22204A	Timing-based detection using MSR 10
IOCTL_BLUEPILL_TLB_HIT_RDMSR	0x222082	TLB misses based detection using RDMSR EFER
IOCTL_BLUEPILL_TLB_HIT_CPUID	0x222086	TLB misses based detection using CPUID
IOCTL_BLUEPILL_TLB_TIME_RDMSR	0x2220C2	TLB cache timing detection using RDMSR EFER
IOCTL_BLUEPILL_TLB_TIME_CPUID	0x2220C6	TLB cache timing detection using CPUID

Figure A.1: IOCTL-Codes for the testing framework

A.3.1 Implementation of the counter-based detection

The counter-based detection has been discussed in section 4.1.4 and its implementation is straightforward. The driver is running in kernel-mode, starts a second kernel thread and orders each thread to acquire one fixed CPU using `KeSetSystemAffinityThread`. In order not to be disturbed by context-switches to other threads/processes the hardware-priority is raised to `DISPATCH_LEVEL` using `KeRaiseIrqlToDpcLevel`. Synchronisation of both threads happens through a busy waiting strategy, which is fine for the short waiting interval. Therefore the hypervisor-calling thread waits until the counting thread started. In this implementation there is no hard threshold in the counter thread which might safely assume virtualization after more than a couple of hundred cycles (see section A.4). This would just be a minor modification though. However, the real values are for the time being of greater interest, thus the counting thread continues until the hypervisor is finished. Listing A.10 (page 37) shows the source code of this method in a slightly simplified manner.

A.3.2 Implementation of TLB-based detection

TLB profiling methods have been discussed in section 4.1.3. The method implemented for the testing framework (additionally to the cache timing based method which was developed by Rutkowska and also integrated into the framework) works as follows:

Allocate pages. There are two TLBs on the testing machine. The L1 TLB has space for 32 entries of 4kb-data-pages, whereas the L2 TLB is 4-way associative with 128 entries for each set resulting in a total of 512 entries.

Figure 4.1 on page 18 illustrates that in order to fill each L2 TLB entry, four pages of each index with different tags each are required (see section 4.1.3 for a description of multiple-way associativity). Luckily if one manages to allocate a continuous chunk of

Listing A.10: counter-based detection procedure (simplified)

```

ULONG globalSMPCounter;
ULONG smpStartValue;
ULONG smpEndValue;

VOID Thread1(void) {
    KIRQL OldIrql;
    CCHAR cProcessorNumber = 0;

    KeSetSystemAffinityThread ((KAFFINITY) (1 << cProcessorNumber));
    OldIrql = KeRaiseIrqlToDpcLevel ();

    while(globalSMPCounter==0) ; //wait until the other thread started

    smpStartValue = globalSMPCounter;
    read_msr_efer(); //say hello to mister hypervisor
    smpEndValue = globalSMPCounter;

    KeLowerIrql (OldIrql);
    KeRevertToUserAffinityThread ();
}

VOID Thread2(void) {
    KIRQL OldIrql;
    CCHAR cProcessorNumber = 1;

    KeSetSystemAffinityThread ((KAFFINITY) (1 << cProcessorNumber));
    OldIrql = KeRaiseIrqlToDpcLevel ();

    while(smpEndValue == 0) globalSMPCounter++;

    KeLowerIrql (OldIrql);
    KeRevertToUserAffinityThread ();
}

ULONG RunTest(void) {
    globalSMPCounter = 0;
    smpEndValue=0;

    StartThread(Thread2);
    Thread1();

    return smpEndValue-smpStartValue;
}

```

A Empirical Results

512 4kb-pages, these requirements are already fulfilled, though it turned out, that it is not that easy to convince Windows to allocate such a big chunk of 4kb-pages since it prefers to go for 2MB pages which are of no interest for the detection approach. Another 32 pages are allocated for the L1 TLB.

Write Pattern. Using `memset()` all pages are filled with the value 0x22. One additional trap page is filled with 0x11.

Fill TLB. To provide results as accurate as possible, the TLB is flushed first. Now each memory page is accessed in order to cache its physical address in the TLB: at first the 512 pages that are supposed to be saved into the L2 cache, followed by the 32 pages that fill the L1 cache.

Remap PTEs. Eventually the PTEs of all those pages are remapped to point to the additional 0x11-filled trap page.

Force #VMEXIT. The hypervisor intercepts and accesses pages in memory tainting the prepared caches.

Check TLB state. Each page is accessed again. If 0x22 is read, it is known that the TLB cached this page's physical address. But if 0x11 is read, there was a TLB miss hence the CPU walked the page tables and read the patched physical address pointing to the trap page. The number of replaced TLB entries can then be acquired by counting all such TLB misses.

There are however several problems in this implementation, which help explain, that even when not executing any instructions, there are still around 50 TLB misses (see the empirical data for this method in section A.4):

Variables & Functions. There are quite a number of local and global variables that need to be accessed within the testing procedure. These also include the operating-system's page-tables filling a couple of entries in the TLB.

Undefined L2 Cache operation. The available documentation on the L2 TLB cache of the processor is insufficient, as it does not explain under which circumstances a L1 TLB entry will or will not replace a L2 TLB entry. Accordingly the detector tries its best to fill the cache, but cannot be sure.

Interrupts. The detector already works with a hardware-priority of `DISPATCH_LEVEL`, but still occurring interrupts force execution of other code accessing other data.

The source code is provided with this document, but not printed here due to its complexity.

A.4 Empirical results

For being able to see the effectiveness of the detection methods, a couple of tests were run, clearly showing that a running hypervisor can definitely be detected by these measures. The tests are the same as presented in section A.3. The testing machine is running Windows Vista 64 on an AMD Athlon(tm) 64 X2 Dual Core Processor. The test was conducted twice: On a clean system (figure A.2) and on a system running Bluepill (figure A.3). The graphs

show the clock-cycles which the processor used during the execution of the RDMSR EFER instruction (plus the overhead for acquiring the time using either RDTSC (read time-stamp counter), RDTSCP (read time-stamp counter and process id) or RDMSR 10 (read time-stamp counter using a RDMSR instruction)). As has been explained in section 4.1.1, the RDTSC instruction may be executed in non-linearized order and thus yield wrong results. This is why the additional method of reading the TSC register using RDTSCP, which is a linearizing instruction, is provided as well. For the counter-based detection, the graph shows the number of increments which the counting thread was able to do, while the other one is executing either RDMSR EFER or CPUID which are both likely to be trapped by the hypervisor (the latter one is even forced to be trapped by VT-x specification).

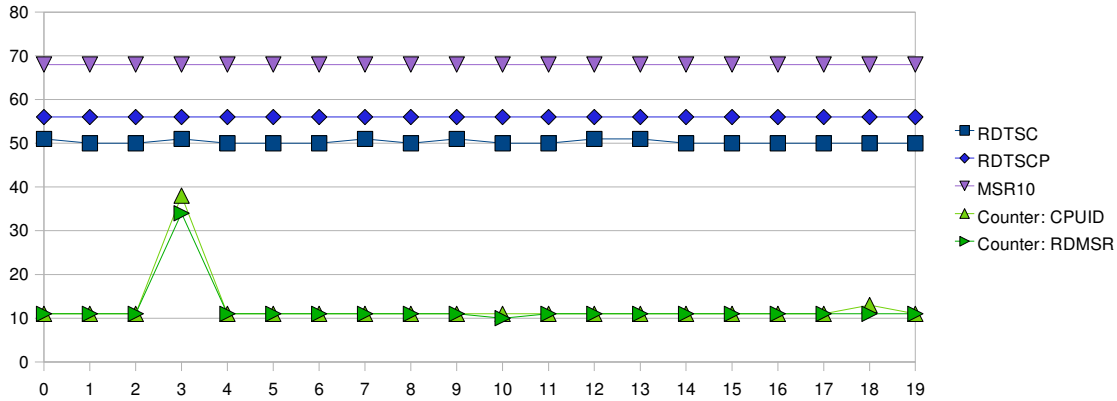


Figure A.2: Empirical timing and counter results. Bluepill disabled.

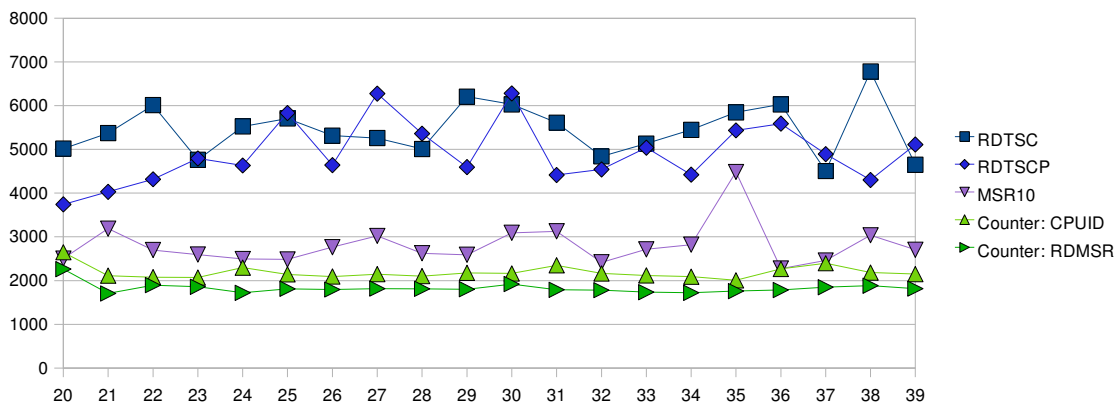


Figure A.3: Empirical timing and counter results. Bluepill enabled.

Since the available implementation of Bluepill has no more a working implementation of RDTSC offset-cheating, this feature could unfortunately not be tested and evaluated into the graphs. The huge differences within one row of acquired timing values when Bluepill is running, suggests however that the task of providing a realistic approximate for the offset to be subtracted is much harder than originally assumed (as already discussed in section 4.1.1) and most likely a malicious hypervisor could be detected through more sophisticated timing analysis anyways.

The only caveat for the counter-based detection is that it requires two CPUs or two cores

A Empirical Results

within the CPU. However, since this is already very much standard on modern processors with support for virtualization extensions, this is only a small drawback. The value-range in the graphs clearly indicates the reliable detection of an interception of the executed instructions.

For the TLB-based detection that was implemented, the results are not as clean and show a greater disparity due to the reasons discussed in section A.3.2. Nevertheless the noise can

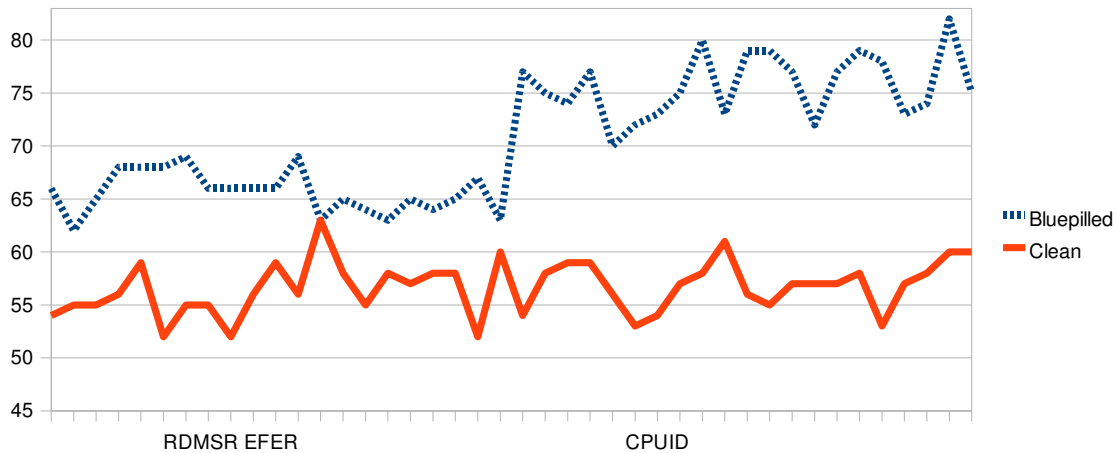


Figure A.4: Invalidated TLB entries.

be filtered by averaging a couple of conducted test's results, although it is probably still hard to find fixed values to distinguish a clean from an infected system. Since I lack access to other systems I could compare the timing to, this cannot be evaluated right now.

The graph also indicates that when Bluepill is running RDMSR EFER has a lesser influence on the TLB than the CPUID instruction, which is confirmed by the results of the TLB cache timing method, which works by using RDTSC to get the timing differences of TLB accesses: At first data is accessed, so that the L1 TLB fills up, then this data is read again this time measuring the access time. After a forced #VMEXIT some entries have been but back into the L2 TLB cache. This introduces 1-3 extra cycles when fetching the L2 TLB entry. Special care needs to be taken, that the actual data is stored in the L1 data cache, otherwise a real memory hit occurs introducing an unpredictable amount of cycles rendering the method useless. This method has been verified to very reliable detect a running hypervisor (see figure A.5). The values are constant and do not vary for a given Bluepill implementation.

	L1 TLB misses
Clean	2
Bluepilled: RDMSR EFER	8
Bluepilled: CPUID	16

Figure A.5: TLB misses in several setups

List of Figures

2.1	Type 1 malware [Rut06]	4
2.2	Type 2 malware [Rut06]	5
2.3	Type 3 malware [Rut06]	6
3.1	Differences between traditional and thin hypervisors (based upon [PLF07])	9
3.2	Illustration of the moving-into-VM process [Rut07b]	11
3.3	Bluepill private page tables [Rut07b]	13
4.1	Illustration of the L2 TLB organization [Rut07b]	18
4.2	Different physical addresses returned after hypervisor interception	19
4.3	Illustration of counter-based attack. [Rut07b]	22
5.1	DMA flaw in trusted hypervisors [Rut08]	28
A.1	IOCTL-Codes for the testing framework	36
A.2	Empirical timing and counter results. Bluepill disabled.	39
A.3	Empirical timing and counter results. Bluepill enabled.	39
A.4	Invalidated TLB entries.	40
A.5	TLB misses in several setups	40

List of Figures

Bibliography

- [Ada07] Keith Adams. BluePill detection in two easy steps, 2007. <http://x86vmm.blogspot.com/2007/07/bluepill-detection-in-two-easy-steps.html>.
- [AMD07a] AMD. *AMD64 Architecture Programmer's Manual; Volume 2: System Programming*, September 2007. http://www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/24593.pdf.
- [AMD07b] AMD. *AMD64 Architecture Programmer's Manual; Volume 3: General-Purpose and System Instructions*, September 2007. http://www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/24594.pdf.
- [AMD08] AMD. *Revision Guide for AMD NPT Family 0Fh Processors*, February 2008. http://www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/33610.pdf, Revision 3.30.
- [Bar07] Edgar Barbosa. Detecting Bluepill, 2007. <http://rapidshare.com/files/42452008/detection.rar.html>, Presentation on SyScan Conference, 2007.
- [BDK05] Michael Becher, Maximillian Dornseif, and Christian N. Klein. FireWire: all your memory are belong to us, 2005. <http://md.hudora.de/presentations/firewire/2005-firewire-cansecwest.pdf>, Presentation on CanSecWest Vancouver, 2005.
- [Bul08a] Yuriy Bulygin. CPU side-channels vs. virtualization rootkits: the good, the bad, or the ugly, April 2008. http://www.c7zero.info/stuff/hyper-channel_toorcon_seattle.ppt, Presentation on ToorCon, Seattle 2008.
- [Bul08b] Yuriy Bulygin. Insane Detection of Insane Rootkits: Chipset Based Detection and Removal of Virtualization Malware, August 2008. <http://www.c7zero.info/stuff/bh-usa-08-bulygin.ppt>, Presentation on Black Hat Conference, Vegas 2008.
- [Chi] Ramesh Chinta. Kernel Mode Code Signing on Windows Vista and Windows Longhorn Server. <http://download.microsoft.com/download/0/5/0/050a2d04-7432-4325-a5c3-dcbd54cf6695/KernelModeCodeSigningonWindowsVistaandWindowsServerLonghorn.ppt>.
- [ESZ08] Shawn Embleton, Sherri Sparks, and Cliff Zou. SMM Rootkits: A New Breed of OS Independent Malware, September 2008. <http://www.eecs.ucf.edu/~czou/research/SMM-Rootkits-Securecom08.pdf>.
- [Int07] Intel. *Intel® 64 and IA-32 Architectures Software Developer's Manual; Volume 3B: System Programming Guide, Part 2*, November 2007. <http://download.intel.com/design/processor/manuals/253669.pdf>.

Bibliography

- [Joh06] Ken Johnson. Subverting PatchGuard Version 2. December 2006. <http://www.uninformed.org/?v=6&a=1&t=pdf>.
- [Kle] Tobias Klein. Trapkit: Scoopy Doo – VMware Fingerprint Suite. <http://www.trapkit.de/research/vmm/scoopydoo/index.html>; accessed Apr 21, 2008.
- [KWC⁺06] Samuel T. King, Yi-Min Wang, Peter M. Chen, Chad Verbowski, Helen J. Wang, and Jacob R. Lorch. SubVirt: Implementing malware with virtual machines, May 2006. <http://www.eecs.umich.edu/~pmchen/papers/king06.pdf>, Proceedings of the 2006 IEEE Symposium on Security and Privacy.
- [Mic] Microsoft Corporation. Patching Policy for x64-Based Systems. <http://www.microsoft.com/whdc/driver/kernel/64bitPatching.msp>; accessed Jul 26, 2008.
- [MJ05] Matt Miller and Ken Johnson. Bypassing PatchGuard on Windows x64. December 2005. <http://www.uninformed.org/?v=3&a=3&t=pdf>.
- [MY07] Michael Myers and Stephen Youndt. An Introduction to Hardware-Assisted Virtual Machine (HVM) Rootkits. August 2007. <http://www.crucialsecurity.com/documents/hvmrootkits.pdf>.
- [One03] Walter Oney. *Programming the Microsoft Windows Driver Model, Second Edition*. Microsoft Press, Redmond, WA, USA, 2003.
- [PLF07] Thomas Ptacek, Nate Lawson, and Peter Ferrie. Don't Tell Joanna, The Virtualized Rootkit Is Dead, August 2007. https://www.blackhat.com/presentations/bh-usa-07/Ptacek_Goldsmith_and_Lawson/Presentation/bh-usa-07-ptacek_goldsmith_and_lawson.pdf, Presentation on Black Hat Conference, Vegas 2007.
- [QS06] Danny Quist and Val Smith. Detecting the Presence of Virtual Machines Using the Local Data Table. March 2006. <http://www.offensivecomputing.net/files/active/0/vm.pdf>.
- [RT06] Joanna Rutkowska and Alexander Tereshkin. Subverting Vista Kernel For Fun And Profit, 2006. <http://www.blackhat.com/presentations/bh-usa-06/BH-US-06-Rutkowska.pdf>, Presentation on Black Hat Conference, Vegas 2006.
- [RT08] Joanna Rutkowska and Alexander Tereshkin. Bluepillling the Xen Hypervisor, August 2008. <http://invisiblethingslab.com/bh08/part3.pdf>, Presentation on Black Hat Conference, Vegas 2008.
- [Rut04a] Joanna Rutkowska. Red Pill... or how to detect VMM using (almost) one CPU instruction. November 2004. <http://invisiblethings.org/papers/redpill.html>.
- [Rut04b] Joanna Rutkowska. Security Challenges in Virtualized Enviroments, December 2004. <http://www.ccc.de/congress/2004/fahrplan/files/223-passive-covert-channels-linux.pdf>, Presentation on Chaos Communication Congress, Berlin 2004.

- [Rut06] Joanna Rutkowska. Introducing Stealth Malware Taxonomy. November 2006. <http://www.invisiblethings.org/papers/malware-taxonomy.pdf>.
- [Rut07a] Joanna Rutkowska. Beyond The CPU: Defeating Hardware Based RAM Acquisition, February 2007. <http://www.invisiblethings.org/papers/cheating-hardware-memory-acquisition-updated.ppt>, Presentation on Black Hat Conference, DC 2007.
- [Rut07b] Joanna Rutkowska. IsGameOver(), Anyone?, August 2007. <https://www.blackhat.com/presentations/bh-usa-07/Rutkowska/Presentation/bh-usa-07-rutkowska.pdf>, Presentation on Black Hat Conference, Vegas 2007.
- [Rut08] Joanna Rutkowska. Security Challenges in Virtualized Enviroments, April 2008. <http://invisiblethings.org/papers/Security%20Challanges%20in%20Virtualized%20Enviroments%20-%20RSA2008.pdf>, Presentation on RSA Conference, San Francisco 2008.
- [RW08] Joanna Rutkowska and Rafał Wojtczuk. Preventing and Detecting Xen Hypervisor Subversions, August 2008. <http://invisiblethingslab.com/bh08/part2.pdf>, Presentation on Black Hat Conference, Vegas 2008.
- [SB05] Sherri Sparks and Jamie Butler. Shadow Walker, Raising The Bar For Rootkit Detection, 2005. <http://www.blackhat.com/presentations/bh-jp-05/bh-jp-05-sparks-butler.pdf>, Presentation on Black Hat Conference, Japan 2005.
- [Tho84] Ken Thompson. Reflections on trusting trust. *Commun. ACM*, 27(8):761–763, 1984. <http://cm.bell-labs.com/who/ken/trust.html>.