J Comput Virol (2011) 7:1–21
DOI 10.1007/s11416-009-0129-1

ORIGINAL PAPER

Enforcing kernel constraints by hardware-assisted virtualization

Éric Lacombe · Vincent Nicomette · Yves Deswarte

Received: 16 December 2008 / Accepted: 30 July 2009 / Published online: 21 August 2009

Abstract This article deals with kernel security protection. We propose a characterization of malicious kernel-targeted actions, based on how the way they act to corrupt the kernel. Then, we discuss security measures able to counter such attacks. We finally expose our approach based on hardware-virtualization that is partially implemented into our demonstrator *Hytux*, which is inspired from *bluepill* (Rutkowska in subverting vista kernel for fun and profit. In: Black Hat in Las Vegas, 2006), a malware that installs itself as a lightweight hypervisor—on a hardware-virtualization compliant CPU—and puts a running Microsoft Windows Operating System into a virtual machine. However, in contrast with *bluepill*, Hytux is a lightweight hypervisor that implements protection mechanisms in a more privileged mode than the Linux kernel.

1 Introduction

1.1 Context and issue

Everybody agrees now that the use of computers (in particular through the Internet) has become essential in everyday life. People use computers to work, to exchange information, to make purchases, etc. Unfortunately, malicious computer activities are also regularly growing and try to exploit vulnerabilities which are more and more numerous due to the inherent complexity of the software. Malwares may target application software installed on the system but also the operating system itself and particularly its kernel. Corrupting the kernel of an operating system itself is particularly interesting from the attacker point of view because it signifies corrupting potentially all the software that run upon this kernel. In particular, *kernel rootkits* [2] are a kind of malware dedicated to perform such corruption. In order to operate, these malwares need kernel security flaws in order to execute malicious code inside the kernel. These kernel security flaws are particularly spread across device drivers.[1]

As the corruption of the kernel of the operating system provokes the corruption of all the sofware running upon it, the kernel of an operating system needs strong protection mechanisms. But, protecting the kernel in an efficient way is particularly tricky because it is extremely difficult to make the protection mechanisms impossible to escape. Regarding software that run in user-space for example, it is possible to implement effective user-space security mechanisms because they can be implemented inside the kernel and act in a more privileged mode that the entities they monitor. Now, the issue is how to effectively protect the kernel against malicious code execution? It has to be done from a more privileged mode than the kernel itself and it has to be tamper-proof (from the kernel, the user-space or hardware devices).

In this article, we present a mechanism that satisfies these prerequisites thanks to hardware-assisted virtualization.

É. Lacombe (✉) · V. Nicomette · Y. Deswarte
CNRS, LAAS, 7 Avenue du Colonel Roche,
31077 Toulouse, France
e-mail: eric.lacombe@laas.fr; eric.lacombe@security-labs.org

É. Lacombe · V. Nicomette · Y. Deswarte
UPS, INSA, INP, ISAE, LAAS, University of Toulouse,
31077 Toulouse, France
e-mail: vincent.nicomette@laas.fr

Y. Deswarte
e-mail: yves.deswarte@laas.fr

[1] The main reasons for this are that first, the main part of a kernel is constituted of device drivers; second, the rules that regulate device drivers integration into the Linux vanilla kernel—with regard to code quality—are less strict than the ones applied on main kernel subsystem updates.

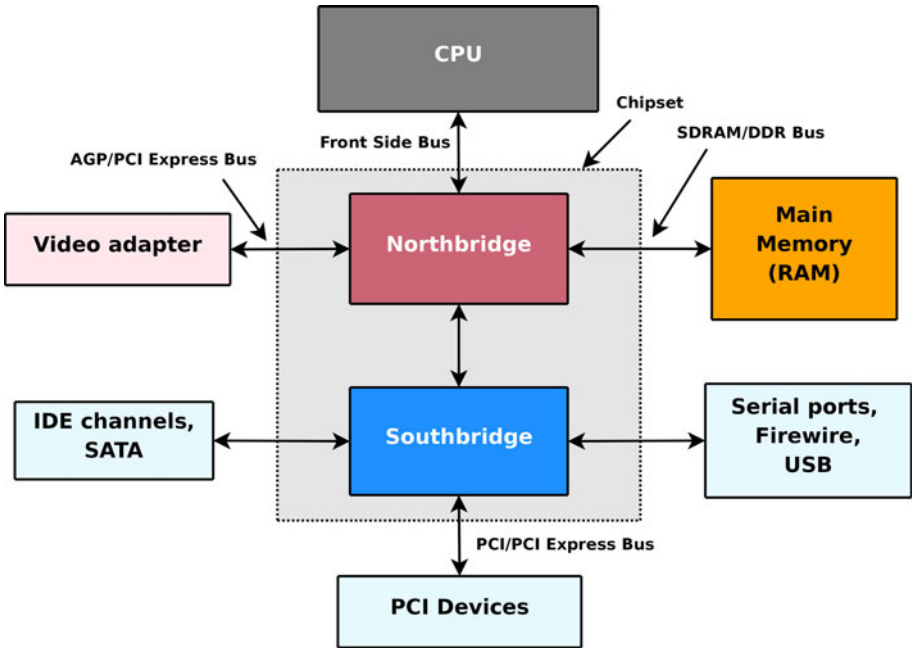


Fig. 1 (simplified) x86 architecture

We do not cover in this paper how the system needs to boot so that our hypervisor takes control over an initially safe kernel. However this kind of action can be done through Static Root of Trust Measurement (SRTM—that checks the BIOS then the master boot record then the kernel) or Dynamic Root of Trust Measurement (DRTM) allowed by Intel Trusted Execution Technology (TXT) [3] or AMD Secure Virtual Machine (SVM).

1.2 Contents

The remaining of this paper is organized as follows. First, we recall in Sect. 2 the technical background required to make this paper self-contained. Then, we establish in Sect. 3 a characterization of malicious actions that can cause a loss of integrity of a running operating system kernel. Section 4 discusses existing security measures that can be deployed in order to partially cover the different classes of malicious kernel-targeted actions. Section 5 is dedicated to the presentation of our approach, called Hytux, that implements security measures in a lightweight hardware-assisted hypervisor in order to protect the Linux kernel from malicious actions. Finally, Sect. 6 provides a summary and discusses future work.

2 Technical background

This technical background focus on the IA-32 architecture[2] [4,5] that is widespread. Although each architecture has its own characteristics, they share some common features: memory management (less typical for embedded system), processor's privilege levels, communication between the different hardware parts and the software (often through interrupts), etc.

2.1 IA-32 architecture

An IA-32 computer is generally based on two main components, a chipset and a processor (or CPU, for Computer Processing Unit). All software components (BIOS, operating system, applications) run on the processor. Meanwhile, the chipset is in charge of device handling. It is generally composed of a Northbridge connected to the main memory (through a component called the MCH—Memory Controller Hub) and the video adapter, and of a Southbridge connected through various buses to the other computer devices (cf. Fig. 1).

On IA32, memory management is operated through a segmentation unit (mandatory) and a paging unit (optional) (cf. Fig. 2). Contrary to the segmentation unit, the paging one is very common to all kind of architectures. As Linux is a multiplatform kernel, the segmentation unit is only used in its bare mode (i.e. the flat mode).[3] This enables to easily cut oneself off from it, to eventually use the paging mechanism only (cf. Fig. 3). Nonetheless, let us briefly explain how the segmentation unit is used. The kernel has to establish segments by writing their description in memory inside a table of segment descriptors, called the GDT (Global Descriptor Table).

[2] We do not cover the IA-32e mode in this section as it would have complicated the memory management explanation.

[3] A single memory segment is set up and associated with the linear addresses from 0 to $2^{32} - 1$ in 32 bits mode or $2^{48} - 1$ in 64 bits mode.

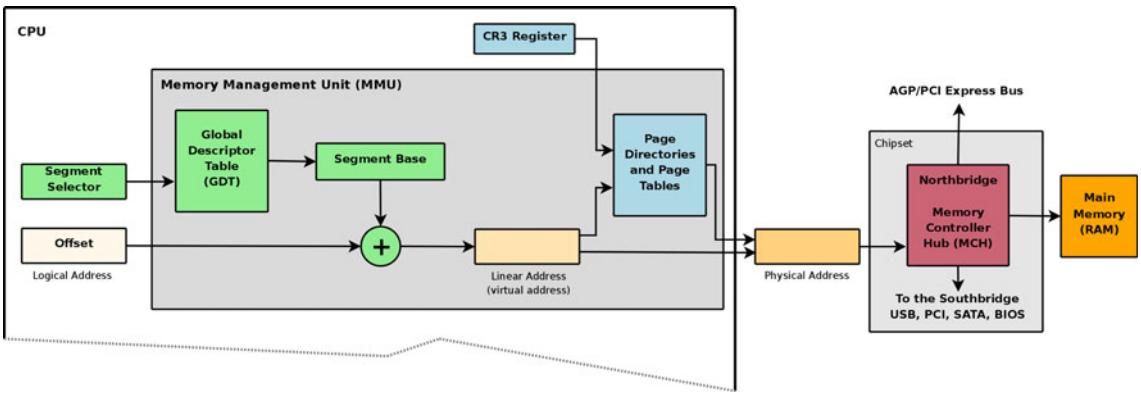
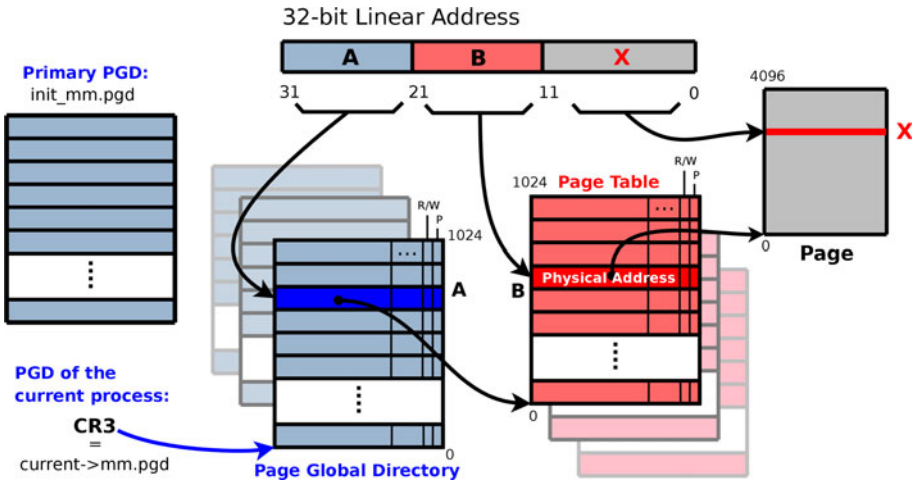


Fig. 2 MMU, segmentation and paging units

Fig. 3 Paging mechanism



Then it loads the table address in the `gdtr` register in order for the CPU to know where the GDT is. The CPU needs a code segment (CS) from which it fetches the instructions to execute, a data segment (DS) and a stack segment (SS).

The IA32 architecture is designed with a 4-ring structure, and each of them represents a specific execution mode. A privilege level is associated to each mode. The most privileged ring is ring 0—the kernel execution mode—while the least privileged mode is ring 3 which is dedicated to user space applications.

The communication between kernel and user space—i.e switching from ring 0 to ring 3 and conversely—can be established by different events. Among them, interrupts are the most frequent. They are divided into exceptions (i.e., interrupts from the processor whenever a division by zero or a page fault occurs, etc.), hardware interrupts (i.e. those which are triggered by devices, such as pressing a key for example) and finally software interrupts (i.e., interrupts that are triggered by the software, e.g., when a user space application invokes a system call).

On IA32 architecture, those interrupts are numbered from 0 to 255. Each of them is associated to a handler if it has actually been set by the kernel. That handler is a function that is executed when the interruption is raised. All these functions are accessible from a specific table in memory: the Interrupt Descriptor Table (IDT). The kernel fills this table and then loads its address into the processor via the `lidt` instruction.

A hardware interrupt or a processor exception stops user space or kernel-space execution and launches the corresponding kernel function. Hardware interruptions occur asynchronously whereas processor's exceptions trigger synchronously. The kernel handles the interruption or exception and then hands over to the user space. However, before that, the kernel can decide to carry out more urgent tasks. Particularly, in the Linux case, the scheduler verifies whether there exists a higher priority process that needs to be executed.

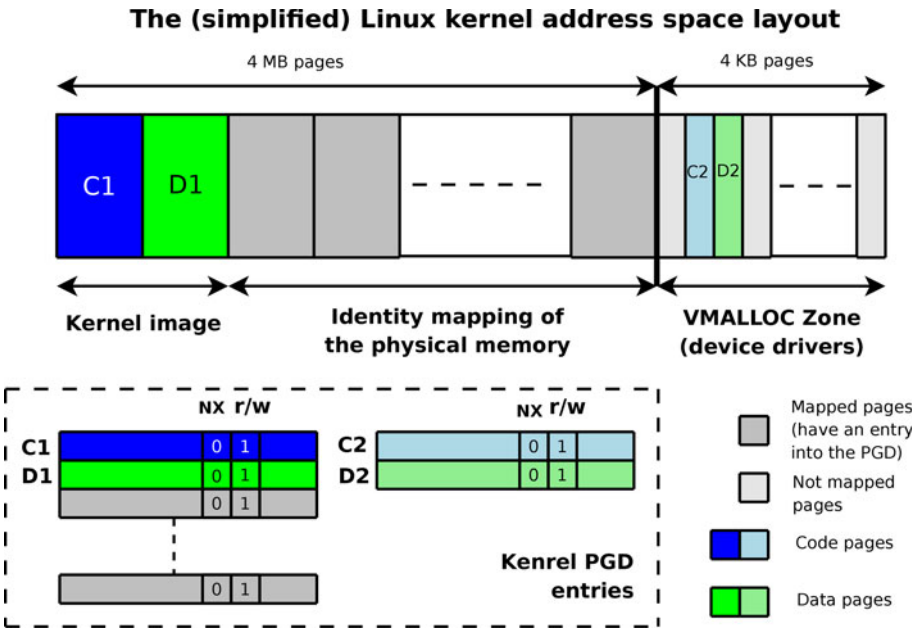


Fig. 4 Kernel address space layout

2.2 Linux kernel address space layout

Figure 4 represents a simplified view of the kernel memory-space layout. Let us take the opportunity of this section to introduce the page attributes that allow the paging unit to enforce memory access rights on a page basis. Those attributes that qualify the different pages on memory are written—in the 4 KB paging mode—on the 12 lower bits of each page entry (as they are not used to reference a 4 KB page). Similarly, attributes for group of pages are present in the 12 lower bits of each page directory entry. These page directory entries can also be used as 4 MB page entries, if their *Page size* attribute is set to one.

Let us now mention the attributes that especially have an importance in this article. First, the Read/Write (`R/W`) attribute allows a read or a write access from the CPU to the affected page, if it is set to 1. Otherwise, the page is enforced to be read-only by the MMU. The second attribute that has an importance in our context is the No eXecution (`NX`) attribute which, if set, enforces that the page cannot be accessed for instruction execution. Let us emphasize that when the CPU tries to access a page in a mode that is forbidden an exception, more precisely a page fault, is triggered.

2.3 Hardware support for virtualization—the case of Intel VT

Virtual-machine extensions of Intel processors define processor-level support for virtual machines on IA32 processor. They allow to support two classes of software: first, the Virtual Machine Monitor (VMM, a.k.a. the hypervisor) that acts as a host and has full control of the processor(s) and other platform hardware; then, the Guest Software which is run inside a Virtual Machine (VM). Each of these VM operates independently of the other ones and uses the same interface to processor(s), memory, storage, graphics, and I/O provided by a physical platform.

Processor support for virtualization is provided by a form of process operation called VMX operation. There are two kinds of VMX operation: VMX root operation that is provided for the VMM execution, and VMX non-root operation that is provided for guest software execution. Processor behavior in VMX root-operation is quite the same as it is outside VMX operation with the main difference that a set of new instructions is available. Processor behavior in VMX non-root operation is restricted and modified to facilitate virtualization. Instead of their ordinary operation, some instructions and events cause transition to the VMM, also called VM-exits. Because these VM-exits replace ordinary behavior, the functionality of software in VMX non-root operation is limited. This limitation allows the VMM to retain control of processor resources. Because VMX operation places restrictions even on software running with current privilege level 0 (a.k.a. ring 0 mode), guest software can run at the privilege level for which it was originally designed. This capability may simplify the development of a VMM.

The life cycle of a VMM can be summarized as follows. First, software enters VMX operation by executing the VMXON instruction. Then, using VM-entries, a VMM can launch guests into virtual machines (to carry out a VM-entry, the VMM executes the instruction VMLAUNCH and VMRESUME). It regains control using VM-exits.

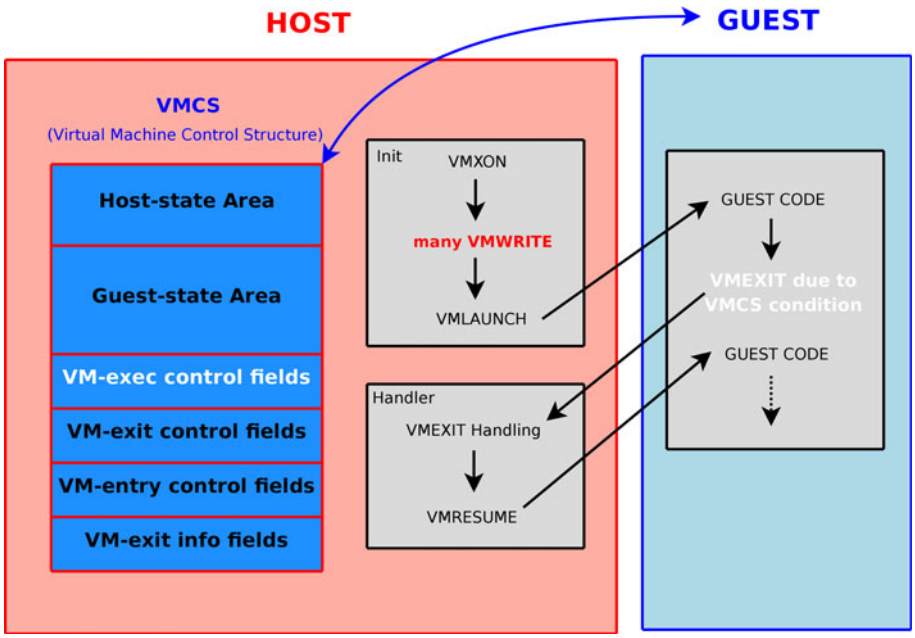


Fig. 5 Brief overview of Intel VT-x

Those latter transfer control to an entry point specified by the VMM. The VMM can take action according to the cause of the VM-exit and can then return to the virtual machine using a VM-entry. Optionally, the VMM may decide to shut itself down and leave VMX operation (by executing the VMXOFF instruction).

VMX non-root operation and VMX transitions are controlled by a data structure called a Virtual-Machine Control Structure (VMCS). Access to the VMCS is managed through a component of the processor state called the VMCS pointer (which contains the address of the VMCS). This pointer is read and written using the instructions VMPTRST and VMPTRLD. The VMM configures a VMCS using the VMREAD, VMWRITE, and VMCLEAR instructions. It is worth noting these instructions trigger VM-exits if they are executed from VMX non-root operation. The Fig. 5 summarizes the way to use those instructions.

3 Malicious kernel-targeted actions

Only the malicious actions that imply a loss of integrity of a running operating system kernel are considered. This loss of integrity is related to an abnormal modification of either (1) the kernel memory, or (2) the hardware components that the kernel depends on for its execution (the CPU and the MCH), or finally (3) the hardware components it communicates with (i.e., the devices).

In our work we only consider *logical* malicious actions,[4] and for the sake of brevity we call them malicious actions. We also make the following hypotheses:

Assumption 1 The hardware structure[5] on which the kernel depends to execute itself is considered unalterable, except by the provided functions if available (e.g., microcode update facilities of Intel processors [4]).

Assumption 2 The hardware components on which the kernel depends to execute itself do not contain exploitable bugs, backdoors or undocumented functions [6] with regard to security.

From the first hypothesis, we can consider that the part of the hardware structure that can be altered by provided facilities is included in the hardware state. Thus, regarding the hardware components on which the kernel depends for its execution, we consider that only the state of these hardware components can be altered.

So it follows that the loss of integrity of a running kernel stems from the alteration (i.e., an abnormal modification) of either (1) the kernel memory, or (2) the state of at least one hardware components on which the kernel depends to execute itself (e.g. the registers and internal memory of the processor), or finally (3) the hardware components that it communicates with but does not directly depends on to execute itself (that is especially the devices that are connected through the southbridge).

To be more succinct in the remainder of the article we name: the state of the hardware components that the kernel depends on to execute itself, the *execution environment memory*; and the hardware components that it communicates with but does not directly depends on to execute itself, the *devices*.

[4] To oppose to physical malicious actions.

[5] Note that a system, in this case a hardware system, is made of a *structure* that allows it to generate its behaviour, and to hold its *state*.

We can thus classify at a first level the malicious actions that affects kernel integrity, with regard to the kind of the modification they make:

- the malicious actions that alter the *kernel memory* makes up the **Class 1**;
- the malicious actions that alter the *execution environment memory* makes up the **Class 2**;
- the malicious actions that alter the *devices* makes up the **Class 3**.

In order to proceed with a more detailed classification of these malicious actions, we first analyse the access vectors to the kernel memory, then to the *execution environment memory* and finally to the *devices*.

3.1 Access vectors to kernel memory

The first way to access to the kernel memory is through the CPU. This access necessarily implies: first, the Memory Management Unit (MMU) in CPU, then the Memory Controller Hub (MCH) in the northbridge. Thus, an abnormal modification of the kernel memory can stem from:

- A system feature that directly provides the means to modify any regions of kernel space memory. It can be either a software feature (such as the kernel module loader [7], the `/dev/kmem` and `/dev/mem` virtual devices in the Linux case [8,9]) or a hardware feature (such as the CPU System Management Mode [10,11]).
- A system feature that does not provide it but through the exploitation of a flaw inside it (buffer overflows, format strings, usage of incorrect data—null kernel-pointer dereference [12]—or outdated data—cf. the vulnerability that affected Linux kernels patched against the security protection *PaX* [13, Section 2], etc.).

The second way to access the kernel memory is from a device connected to a DMA-capable (Direct Memory Access) I/O bus. So it involves the MCH. These access vectors can be divided in two categories depending on whether the access is initiated by the device or ordered by the CPU:[6]

- In the case the access is initiated by the device, it concerns the devices that are connected on a bus capable of bus mastering (like the PCI or PCI Express bus on IA-32 and Intel 64 architectures). These devices can then take control of the bus and perform a data transfer to the memory without the processor involvement. Thus, for instance, the Firewire bus can be used to read or inject data in physical memory without the operating system consent [14–16].
- In the case the access is ordered by the CPU, the abnormal modification of the kernel memory comes from some malicious software actions that is executed through the operating system.

On recent computers, it is possible to control these accesses through the northbridge by a hardware component called the Input/Output Memory Management Unit (IOMMU) [17] which acts as a router and a filter of data flows to the main memory that come from system devices, and allows the kernel to control DMA access from these devices.

3.2 Access vectors to the execution environment memory

The execution environment memory is composed first of the registers and the internal memory of the CPU, and secondly by the registers of the MCH.

The registers of the CPU are only accessible from the CPU, thus from the software that is executed on it. Let us note that for software running with the nominal mode of x86 CPU[7] in ring 0 privilege, all the registers are accessible except specific SMM registers. In less privileged rings like the ring 3, the software is restricted and cannot access all the registers. In SMM mode, all the registers are accessible plus some private CPU states indirectly (e.g., the SMBASE). Let us remark that some internal memory or registers of the CPU are not accessible at all (e.g., the hidden part of the segment selectors).

The registers of the MCH are only accessible through the CPU[8] and thus by the software that runs upon it. These registers are accessible through the *Memory-Mapped I/O* (MMIO) mechanism which is implemented by the MCH. The MCH maps registers or internal memory of capable devices into the physical address space, which are thus accessible like the main memory (and can be read and written by the assembler instruction `mov` [18]).

3.3 Access vectors to the devices

Only the CPU can access the devices of the computer.[9] It does it in order to configure those devices and access their functions. Three main ways are provided by IA-32 and Intel 64 architectures and depends on the device that is accessed:

[6] In the case a device command another one to perform DMA, we consider the latter as the initiator.

[7] The *protected mode* for IA-32 architecture, and the *IA-32e* mode for Intel 64 architecture.

[8] Some hardware platform can support PCI peer-to-peer transactions that traverse multiple PCI host bridges. In our work we do not consider these platforms.

[9] cf. Footnote 8.

- the *Memory-Mapped I/O* (MMIO) mechanism: which performs the mapping of the registers into the physical address space (as explain previously);
- the *Programmed I/O* (PIO) mechanism: which performs the mapping of the registers into a separate 16-bit address space, and can be accessed by the assembler instruction `in` and `out` [18];
- the PCI mechanism [19]: which is used to access PCI configuration registers (included in each PCI device). These registers are located in a third address space. They can be accessed by specifying in the PIO register of address `0xcf8` the address of the register that we want to access. Then, the chipset automatically updates the PIO register of address `0xcfc` with the value of the PCI register expected, which can be then read and written thanks to PIO access.

Access to MMIO or PIO is restricted to the ring 0, that is the kernel mode. But it can be granted by the operating system to privileged user space applications (for Unix-based OS it usually means for application that runs with root privileges) through the system calls `iopl` (for full access on PIO) and `ioperm` (for access on specific PIO).

3.4 Malicious kernel-targeted action classes

We now discuss the kind of malicious actions that alter the kernel behaviour. The analysis that we performed has lead to a more detailed classification on these malicious actions.[10]

3.4.1 Class 1—alteration of the kernel memory

– *Class 1.1—invalid modification of kernel-mode execution path:*
This class is characterized by malicious actions that need to inject some code in order to achieve their work. This class has some prerequisites that depend on the kind of the action.

- *Class 1.1.1—addition of a reachable malicious kernel code region:*
This class is characterized by the malicious actions that inject a code region in the kernel memory space. Examples of such malicious actions benefit from kernel features such as a kernel module loader [20].
- *Class 1.1.2—overwriting an existing kernel code region with malicious code:*
This class is characterized by the malicious actions that need a code region to be writable. Either they permanently overwrite existing code with no more possible execution of this one; or they hijack the existing code and keep executing it but with some new malicious instruction added (such malicious actions were pioneered by Silvio Cesare [21]) thanks to padding in *code* pages.
- *Class 1.1.3—injection of reachable malicious code into a kernel data region:*
This class is characterized by the malicious actions that need a data region to be executable. For instance, malicious actions that use buffer overflow techniques [22] belong to this class. This class also encompasses the malicious actions that inject code into *data* page padding in order to carry out their work.
- *Class 1.1.4—injection of a reachable malicious code into a non-kernel region (typically user space region):*
This class is characterized by the malicious actions that only need that the kernel does not prevent invalid pointers to be dereferenced from kernel mode. It means that the malicious action exploits a flaw in the kernel that enables the execution of random non-kernel (e.g., user space, hypervisor space) code in ring-0. This stems from kernel bugs that can be exploited in order to write a valid user space address into a kernel pointer, that allows at least an injection of unexpected data from user space[11] to kernel space and in the worst case an execution of user space code. An example of such a malicious action is depicted by the local root exploit that was allowed by the vulnerability of Linux's*vmsplice* system call [23,24] (cf. [12] for an explanation on how an exploit based on a null kernel-pointer dereference works).

– *Class 1.2—invalid modification of kernel-mode variables:*
This class is characterized by malicious actions that do not inject code into the kernel, but provoke an abnormal modification of the kernel behaviour by modifying its variables.

- *Classe 1.2.1—alteration of execution state variables:*
The actions of this class alter the kernel behaviour by modifying some of the variables which its execution depends on.
Examples of such variables are: the control flow data (especially the program counter) that reside in the stack, the data used in a branching condition of some code, the attributes of page tables (`Present`, `Read/Write`, `No eXecution` flags, etc.).
Malicious actions that alter the control flow data in the stack can be used in order to execute existing kernel code in a wrong order [25,26]. For instance, the malicious action could execute a function (or just only

[10] It is worth noting that an attack that targets a kernel is composed of multiple malicious actions.

[11] The user space limit in Linux is represented by the constant `TASK_SIZE`.

some code) with forged parameters by modifying the stack frame. It could replace the program counter that has been saved in the stack with the address of an existing code in kernel memory in order to divert the execution flow, hence to execute an abnormal code with regard to the execution flow.[12]
Other malicious actions overwrite some page attributes in order to circumvent execution prevention on a particular page.
Furthermore, integer overflows and especially reference count overflows are malicious actions part of this class [27].
Likewise, all malicious actions that disable security protection by overwriting only kernel data (without any other code execution) are part of this class.

- *Classe 1.2.2—alteration of auxiliary variables:*
  The actions of this class alter the kernel behaviour by modifying some of the memory variables that do not affect the execution flow, and that we call the auxiliary variables.
  Such actions can be used to blank out error messages, alert messages, etc. (by nullifying for instance some strings used by the primitive `printk()` in a particular section of kernel code).
  Other actions can just modify auxiliary variables that will be sent to another computer through the network, from which they will be used as execution state variables.

3.4.2 Class 2—alteration of the execution environment memory

– *Class 2.1—alteration of CPU registers or CPU internal memory:*
  This class is characterized by the malicious actions that abnormally alter:

  - some critical CPU registers such as segment selectors (`cs`, `ds`, `ss`, etc.), `idtr` register, `gdtr` register, Memory Type Range Registers (MTRR), Model-Specific Registers (MSR), and so on;
  - parts of CPU internal memory such as the microcode region (if available) used to change the processor behavior [4].

  Some attackers, in order to install kernel rootkits [2], copy the IDT, then modify this copy to finally load its address into the `idtr` register of the processor (thus replacing the previous one) [28]. This last action is malicious and is part of this class.
  Another example of such malicious actions is shown by Loïc Duflot [29] in his modification of the SMI handlers (that is the routines executed in SMM by the CPU in response to a System Management Interrupt). Its proof-of-concept implies the modification of the internal CPU register `SMBASE`, and some critical `MTRR` registers of the CPU.
– *Class 2.2—alteration of MCH registers:*
  This class is characterized by the malicious actions that alter some registers of the MCH in order to alter the behaviour of the kernel. Such malicious actions are also illustrated by Loïc Duflot's proof-of-concepts in [29,30]. In [29], it benefits from the modification of the `SMRAMC` register of the MCH[13] and in [30] it especially involves the `AGPM` register (that is written in order to enable graphics aperture accesses) of the MCH.

3.4.3 Class 3—alteration of the devices

This class is characterized by the malicious actions that alter values on some registers of a device,[14] or in its internal memory (if available), or even that alter the structure of a device if this one is adaptable (like devices that use FPGA).

To our knowledge, there is no example of such malicious actions that has been published. We can only imagine possible scenarios where a device, say a network adapter, built with FPGA, could be reprogrammed in order to become hostile to the kernel, and for instance exploits an hypothetical vulnerability of its network stack, through the injection of malicious network packets.

4 How to protect the kernel against malicious actions

In this section, we discuss how to provide some protection against malicious actions on a running kernel. This discussion led us to the development of a new approach based on hardware-assisted virtualization that we detail in Sect. 5.

The discussion that follows is structured according to the classification of malicious actions that we set up in the previous section.

4.1 About security mechanisms

The security measures used to protect an information system are generally classified in three main groups: prevention,

[12] This approach is generalisable in order to execute in sequence many parts of the legitimate existing code (by modifying the saved program counters in the successive stack frames). We can name this approach, a maliciously ordered execution flow.

[13] Further information on that topic is available in [31], which discusses SMM rootkits.

[14] Let us recall that what we call *devices* are the hardware components which the kernel communicates with but does not directly depends on to execute itself.

detection and recovery. It has been proved that malware detection is an undecidable problem [32, Chap. 3]. Thus, as recovery mechanisms need detection measures, we favour in our approach prevention measures when possible. In the remaining of the section, we only focus our attention on prevention measures that protect the kernel space against malicious actions.

4.2 Control of the access vectors

We identified two kinds of access vectors to the kernel memory for malicious actions in Sect. 3.1, and one kind of access vectors to the execution environment memory in Sect. 3.2 and to system devices in Sect. 3.3. We discuss existing security measures at this level. Note that a malicious action uses only one access vector but can then enable other access vectors, for other malicious actions.

4.2.1 Control of the access vectors to the kernel memory

- *Control of the CPU-based Access Vectors:*
  As explained in Sect. 3.1, kernel features that directly provide write access to any region of the kernel space (such as the kernel module loader, the `/dev/kmem` or `/dev/mem` devices in the Linux case) are broadly used by lots of malware to inject themselves into the kernel memory space [2]. These features must obviously be controlled. For instance, the `/dev/kmem` and `/dev/mem` devices can be disabled (as done by grsecurity [33] for instance) or can be filtered to only allow the access to memory-mapped I/O (as done by current Linux kernels if correctly configured). Also, to detect malicious kernel modules, a solution is to set up an automatic verification of modules through cryptographic signatures [34]. However, by this way we do not prevent exploitation of bugs that can be present inside signed modules. Also, we must ensure that the way to add modules is unique and cannot be tampered with.
  The other access vector used by malware in order to alter kernel memory is the exploitation of flaws in kernel features that are not supposed to provide the ability to modify the kernel space. Obviously, contrary to the previous access vector, this one cannot be controlled by the same techniques. Besides, finding this kind of access vector inside the kernel is easier if more modules—that can be potentially bogus—are added to it. Actually, the vast majority of kernel flaws stems from device drivers (cf. Footnote 1). A security solution, called *PaX* [35], developed for Linux contains mechanisms (such as *randkstack* that implement kernel stack randomization) to provide some generic ways to protect the kernel against malicious actions. However, those mechanisms are currently implemented in the same level of privilege that the kernel and thus only try to prevent malicious data from entering the kernel space. They could not be effective if malicious code is already present inside the kernel.
- *Control of the DMA-based Access Vectors:*
  In order to circumvent this problem, it is possible to disable the DMA channels from the kernel, but it is then really CPU-time consuming to transfer data through I/O devices, and it requires that device drivers are modified in order to poll for data instead of setting DMA transfer (which is unacceptable for some devices). To a lesser extent, for Linux kernels, disabling raw I/O and the `/dev/port` device (as done by grsecurity [33] for instance) forbids DMA transfers to be established from user space.
  Finally, the most efficient approach applies to computer systems that include an Input/Output Memory Management Unit (on Intel the technology is VT-d, and on AMD it is part of HyperTransport architecture). With that unit, it is possible to protect main memory against malicious devices [17]. An IOMMU is a memory management unit (MMU) that connects a DMA-capable I/O bus to the main memory. Like a traditional MMU, the IOMMU takes care of mapping I/O addresses to physical addresses. The translation tables are located in main memory and are under the control of the CPU, i.e., the kernel, instead of the device. That said, the translation tables for the IOMMU are now a critical part that need to be protected against malicious kernel actions. Again, the protection mechanisms need to have a higher privilege level than the kernel.

4.2.2 Control of the access vectors to the execution environment memory and the devices

Currently, operating systems implement the control by the kernel of user space applications (ring 3) to access execution environment memory and devices. However there is no control of ring 0 access nor SMM access.[15] Thus, these controls may be evaded if the kernel suffers from a security flaw that allows ring 0 or SMM code execution under the control of the attacker.

4.3 Analysis of existing approaches to prevent kernel corruption

4.3.1 How to protect against Class 1 actions

Here, we focus on existing approaches to protect the kernel memory, i.e., the existing approaches that try to cover the malicious actions of the Class 1 (refer to Sect. 3.4).

[15] It is worthwhile to note that this kind of control cannot be effectively performed in ring 0, as it need to be achieved at a more privileged level.

Let us first note that techniques like the *Address Space Layout Randomization* (such as the one proposed by *PaX* [35]) are not effective to protect kernel space against malicious actions. Not only the ASLR has to be carried out on a 64 bits architecture [36] in order to have an effective protection but it solely applies to user space. Indeed, some vital kernel structures may precisely be located in user space regardless the ASLR. For instance, the GDT can be pinpointed in memory thanks to the execution of the instruction `sgdt` which is legal in user mode.

- *How to Protect Against Class 1.1 Actions:*
  Concerning Class 1.1, let us focus on the protection of the kernel against malicious actions of each subclass.
  To protect against Class 1.1.1, it is possible to develop solutions restricting the use of kernel features able to modify any region of kernel space memory (as described in Sect. 4.2). To protect against Class 1.1.2, code regions can be enforced to only be executable and not writable. Similarly, to protect against Class 1.1.3, data region can be enforced to only be readable and writable but not executable. That all can be done through page table entry attributes (cf. Sect. 2). However, there may be some issues with the execution prevention of the kernel stack. Indeed, code is sometimes legitimately injected inside the stack as a way to implement certain features. The OpenWall project faced this kind of problem in order to implement non-executable *user* stack for Linux. So, implementing a non-executable *kernel* stack could have led to the same kind of problems. Fortunately, in the Linux case, these issues do only target the *user* stack. Indeed, first, nested functions are not used inside the kernel and thus there is no need for *gcc* to use an executable stack (that is needed for *function trampolines*). Then, the part of the Linux kernel that relies on executable stack—the signal handling subsystem—setup code only in the *user* stack. Finally, functional languages and programs that use runtime code generation, rely on executable stack, but they are executed in user space and thus do not rely on executable *kernel* stack.
  However a malicious kernel action could break out this protection by first changing the page attributes of a data memory region that contains malicious code and then executing this region. Thus, modification of the page attributes must be prevented in order to forbid transition from data to code region. We could prevent the page tables from being modified, by setting to non-writable the pages that contains them. But it would not be possible again for the kernel to add new kernel memory mappings—for modules injection—as the pages that contain the page tables would not be writable anymore. The only solution is then to craft new page tables and to load the `cr3` register with the physical address that references them. But it can also be done by a malware that lives inside the kernel. Thus, we cannot rely on kernel protection that lives at the same level than the kernel. In our approach, presented in Sect. 5, we explain how to face such issues. By using hardware virtualization it is possible to enforce the notion of kernel data and code region with respect to execution rights.
  Finally, to protect against Class 1.1.4, generic solutions to deal with buffer overflow exploitation (such as PointGuard [37]) can be contemplated, since they protect against malicious modification of pointers. Thus, they protect against the diversion of execution to a specific address in memory. Another practical approach is to prevent user space pointers from being dereferenced in kernel mode. This scheme is followed by the security solution PaX [35] with their mechanism *UDEREF* [38].
- *How to Protect Against Class 1.2 Actions:*
  To protect against Class 1.2, the approaches adopted for Class 1.1 is not satisfactory because there is no code injection, only kernel variables are modified.
  In order to prevent malicious actions of Class 1.2.1 (that provoke the alteration of execution state variables) from running, it is crucial to protect control-flow data (e.g., to protect the control-flow information in the stack frame, to prevent kernel pointers from being maliciously overwritten, etc.), but this is not sufficient.
  Execution state variables are numerous, some examples have been given in Sect. 3.4. There is no generic solution to protect the kernel against the abnormal modification of these variables. But approaches for some specific variables exist. We give some of them in what follows.
  Concerning control-flow data, we can consider at a first stage the mechanisms that protect against execution flow diversion through stack overflow, like StackGuard [39] or Propolice/SSP (Stack-Smashing Protection) by using canaries. But they do not protect against buffer overflows that overwrite function pointers [40] (like heap overflow [41]). Thus, at a second stage we could follow a generic approach to protect against all buffer overflows exploitation, such as PointGuard [37] that encrypts pointers when stored in memory.
  This last solution is really intrusive, and relies on the confidentiality of the encryption key. At this stage, we propose a complementary approach that broadly prevents some kernel actions from going mad. In other words, we try to prevent the kernel from maliciously behaving.
  In order to protect the kernel against more insidious malicious actions like reference count overflows [27], the security solution *PaX* [35] provides a generic protection with their mechanism *REFCOUNT*.
  In order to prevent malicious actions of Class 1.2.2 (that provoke the alteration of auxiliary variables), there is, to our knowledge, no existing approach.

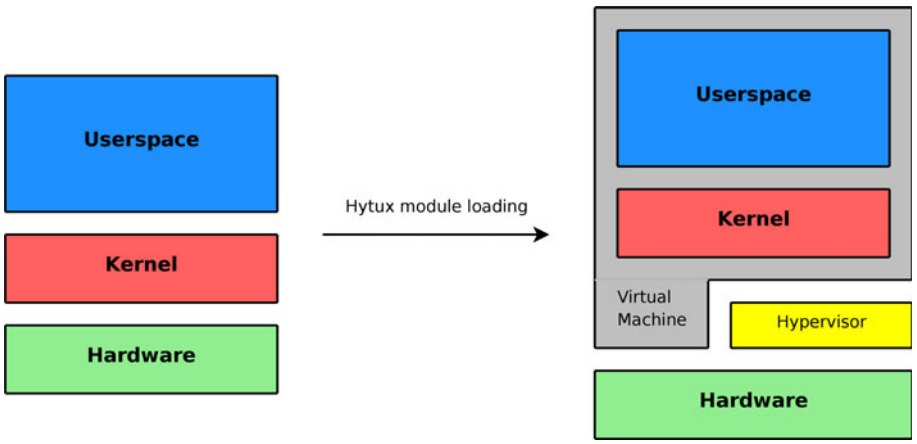


Fig. 6 Hytux—a lightweight hypervisor

The next section presents our approach, based on the preservation of constrained object through a hardware-assisted virtualization solution, which provides a solution to partially cover this class.

4.3.2 How to protect against Class 2 and Class 3 actions

To our knowledge malicious actions of these classes are only partially covered for user space applications as they act in ring 3, and thus can be controlled by the kernel, which can grant or remove privileges to access critical devices or the execution environment memory. However, these approaches suffer from the way they act. They only control the ring 3 access to these resources. Thus they can be tricked by other malicious actions that first exploit some security flaws in the kernel in order to execute some ring 0 code which has full access on the devices and the execution environment memory.

In our work, we try to step up to a solution to this problem, and propose an original approach based on hardware virtualization in Sect. 5.

5 Hardware virtualization enables kernel malware prevention

The traditional security measures we have just discussed face some unresolved issues with regard to malicious actions that occur in kernel space. In our approach, we try to encompass those problems by limiting the damages kernel actions can do to the system. In order to provide this security measure, we implement a lightweight hypervisor that controls some of the actions the kernel can do. This approach is practicable thanks to hardware virtualization technology that enables running the hypervisor in a higher harware privilege level than the kernel. Again, we need to act at a higher privilege level than the kernel if we want to beat malicious actions that occur inside the kernel. Also, as the hypervisor is lightweight, the verification of its correctness is easier. In the next section we discuss our approach. The broad concept is to try to ascertain that some constraints of the system are preserved.

This approach as described in the remaining of this section is self-satisfactory for the classes 1.1.2, 1.1.3 and 1.1.4. For Classes 1.1.1 our approach is complementary to the previously discussed solutions. Finally, concerning the classes 1.2, 2 and 3, our approach provides a unique ability to restrict the ring 0 mode (i.e., the kernel mode) and thus can partially overcome malicious actions of this class.

5.1 Hytux overview

We have developed a partial proof-of-concept for a Linux x86 target that runs on a 64 bits system that supports Intel VT-x [5] and optionally Intel VT-d [42] (cf. Appendix A). Our proof-of-concept is called Hytux and is a lightweight hypervisor that relies on these virtualization technologies (cf. Fig. 6). It borrows this concept from the *bluepill* project [1]. It installs itself as a Virtual Machine Monitor (also called an hypervisor) on a running Linux system[16] and put this one on-the-fly inside a Virtual Machine that is then monitored and controlled (through the configuration of a unique VMCS).

In what follows we explain the different activities that are performed (or envisioned to be performed) by our hypervisor (Fig. 6).

5.2 Protection of kernel-constrained object against alteration through CPU-based access vectors

The reasoning behind this activity is to preserve the entities that are considered to be constrained by the kernel. We define the concept of *Kernel-Constrained Object* in what follows.

[16] Note that Hytux is a Linux Kernel Module.

Definition 1 A *Kernel-Constrained Object (KCO)* is an entity of the system upon which the kernel runs and that *legitimately* should be in a fixed state or in a state that is predictable, during the system execution.

What we emphasize in this definition is that an entity is considered to be a KCO if it is specified to be constrained, no matter if the implementation is bogus or a design flaw exists.

Also what is worth noting is that if we want to preserve a KCO, its constraints need to be verifiable, i.e., they first need to be observable.

5.2.1 KCO preservation explained through an example

Thus, in this activity we try to prevent KCO from being altered by any means. Note that the first state of the KCO that our hypervisor (Hytux) sees is assumed to be safe. From that point Hytux tries to prevent a KCO from being altered. To fully understand this concept, let us take the example of the processor register `idtr` that is a KCO from the Linux kernel point of view. Indeed, it is set at the initialization time to the address of the IDT and is not supposed to be modified afterwards. However, the processor instruction `lidt` available in ring-0 mode—i.e., in kernel mode—allows a new address to be loaded inside this register. Therefore if the kernel contains a bug that can be exploited or a feature (that we call in this context a design flaw) to execute this instruction with an arbitrary parameter, the KCO `idtr` could be altered. Nonetheless the `idtr` register is a KCO. That is why we need in that case to preserve the fixed constraint that governs `idtr`. In order to achieve this goal our approach is to emulate the instruction `lidt` inside our hardware-assisted hypervisor. Thus, when the kernel executes it for the first time the normal behaviour is emulated by Hytux, then it switches permanently to an emulation that does nothing. In this way this KCO is preserved.[17] The `lidt` instruction emulation is easily achieved through Intel VT-x. Indeed, a VM-exit is enforced by setting to 1 the `Descriptor-table exiting` field of the VMCS. We proceed the same way for the `gdtr` register,[18] that is also tagged as a KCO. For the control registers `cr0` and `cr4`, we act quite the same, but only for their bits that can be considered to be KCO.[19] Finally, the case of the `cr3` control register is singular, it is a part of a more complicated KCO that encompasses code and data memory region constraints. This KCO is further discussed in the next paragraph.

5.2.2 The kernel memory space layout as multiple KCO

In order to protect against Class 1;1 malicious actions, Sect. 4 showed that kernel page attributes with regard to page usage can be automatically set. More precisely, for a page that contains code, the `R/W` flag is not set; for a page that contains data that can be written, the `NX` flag and the `R/W` flag are set; and finally for read-only data pages the `NX` flag is set but the `R/W` flag is not. As presented in Sect. 2, the first part of the kernel space is full of 4 MB mapped pages and their attributes are not supposed to be modified. Thus, page attributes must be set in order to enforce executable-only pages, read/write-only pages and read-only pages. Similarly, for the VMALLOC area that is composed of 4 KB pages, we could reflect these constraints with page attributes. However, in this case it is a little bit tricky as this memory space is mainly used to load Linux Kernel Module (LKM). Thus, no page is mapped at all except the ones that contain the already loaded modules. That is why the kernel primitive `vmalloc`—used to allocate memory for LKM—must be modified. In our approach, this kernel primitive must take a flag parameter that informs itself about the type of allocation, that is: code, data or read-only data. With this mechanism in place, `vmalloc` can then set page attributes accordingly to the constraints needed by the different segments of the module (code, data and read-only data), at the time this one is loaded and thus `vmalloc` called. This scheme leads to the situation that is shown in Fig. 7.

However, a malicious *kernel* action could modify the page attributes of a kernel page it wants to use for another purpose (typically a data page transformed in a code page). To face this problem the `R/W` page attribute on the pages that contains all the *kernel* page tables must be unset as the Fig. 8 shows.

But this solution is not satisfactory as the kernel cannot further writes new kernel page table entries when it needs to, i.e., when it loads a module, because a fault page would be triggered and this trap could not be handled. This is obviously not the expected behaviour. To bypass this problem our approach benefits from hardware virtualization and triggers VM-exit when the *kernel* page tables are accessed. To achieve that goal, the hypervisor sets the bit 14 in the *Exception Bitmap* of the VMCS in order to trigger VM-exit on page

[17] In fact for the case of registers `idtr` or `gdtr`, the addresses that are stored inside are linear addresses. Thus the two values in these registers need to be checked against kernel page table entries in order to verify that the corresponding physical addresses are never changed. Besides, it needs to be checked that these physical addresses are uniquely mapped in the linear address space. Thus, when page tables are modified, it needs to be verified that no new mappings with these physical addresses are written. We do not further develop on this topic, as the next section illustrates it with an explanation on how to preserve the constraints of the kernel memory space layout.

[18] Refer to Footnote 17.

[19] Intel VT-x provides guest/host masks for these control registers, which simplify the process.

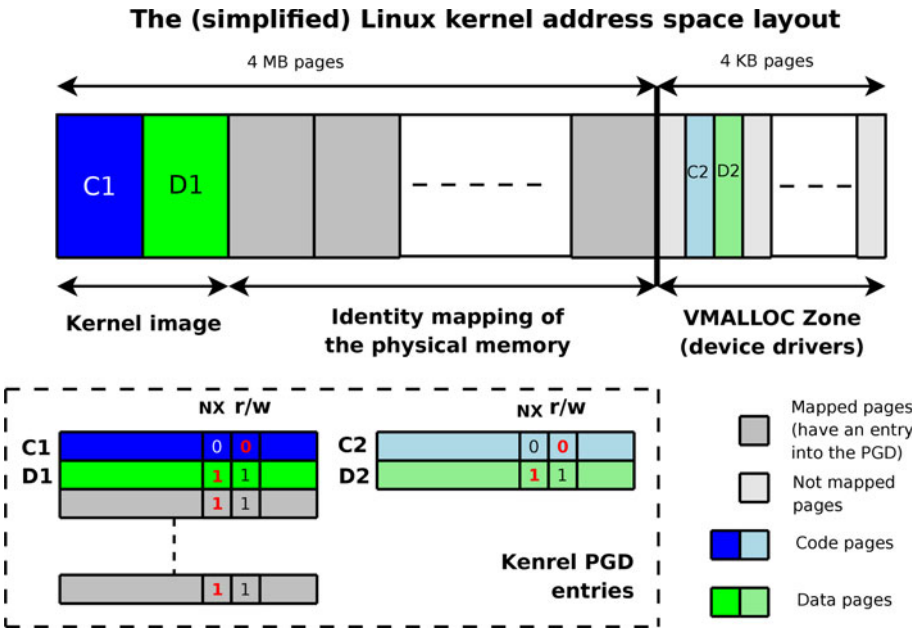


Fig. 7 Kernel address space layout (first modification)

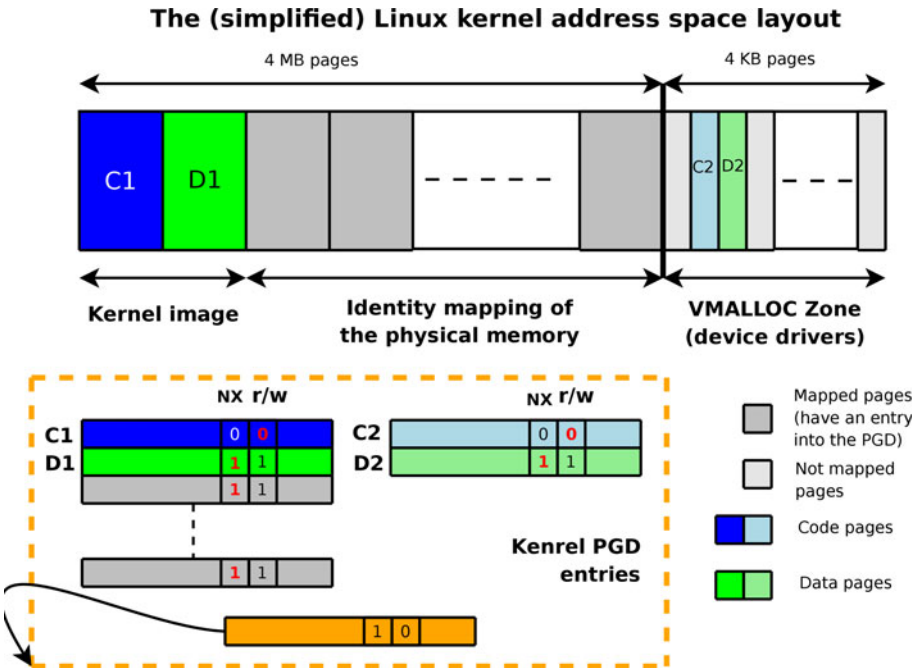


Fig. 8 Kernel address space layout (second modification)

faults. Then when a page fault occurs the CPU switches to the hypervisor. Let us mention at this point that our hypervisor has its own kernel page tables—automatically loaded during a VM-exit—that allows it to write in all memory. Besides, in order to preserve the KCO, the hypervisor needs to keep a copy of the initial kernel space layout with regard to executable-only, read/write-only and read-only pages, (i.e., it keeps a copy of the kernel page tables) in order to validate or not, the future modifications of page table entries. However, page table entries in kernel space are not changed after the system initialisation except for the `VMALLOC` area.[20] Thus the hypervisor only needs to be kept informed on the `VMALLOC` area layout. That implies a modification of the `vmalloc` function in order to inform the hypervisor from the allocation of new pages (through the `VMCALL` instruction that merely

[20] We voluntary forget to talk about the `KMAP` area because the approach deployed to handle this case is similar to the `VMALLOC` one.

triggers a VM-exit). First, it allows the hypervisor to update its KCO (the constrained memory layout) and then, it allows it to effectively write the page table entries with the attributes that depend on the needed constraints.

Let us now explain what happens when the hypervisor takes control of the CPU as a result of the page fault. At this time the hypervisor checks if the fault occurs due to an access to the kernel page tables (by reading the faulting address in the `exit qualification` field of the VMCS). If the faulting address is not in the range of the kernel page tables, then the hypervisor hands over to the kernel (through a VM-entry).[21] Otherwise, if the faulting address is inside the range of the kernel page tables, then the hypervisor replays the instruction that causes the page fault in order to effectively write the page table entry. Then it verifies that the page constraints are preserved with regard to the kernel memory layout it knows.[22] If the instruction results in the invalidation of the constraints on an already existing page table entry (in the kernel page tables), the hypervisor restores the constraints. If the instruction results in the writing of a new page table entry (in the kernel page tables), the hypervisor merely erases this new entry. This last case is justified by the fact the kernel only adds new page table entries in the kernel space through the `vmalloc` function (cf. Footnote 20) and this primitive is modified in order to inform the hypervisor when it wants to add an entry.

We now have to handle a last problem. Consider that a malicious action crafts its own kernel page tables based on the existing ones but with malicious constraints (e.g., a data page with execution rights). Then, it injects them in a kernel data region, and eventually triggers the loading of the `cr3` register with the address of the top of these malicious page tables. This scenario circumvents our protection. This is why all `cr3` loads must be controlled. This is again easily done through hardware-virtualization. In our approach, the `CR3-load exiting` field of the VMCS is set, in order to trigger a VM-exit on each `cr3` load. At this time the hypervisor checks the last entries of the top-level page table (known as the Page Directory in the IA32 mode and as the PML4 table in the IA32e) from the address that is going to be loaded on `cr3`. These entries constitute the kernel address space. Thus, they must be equal to the ones it knows. If it is not the case the hypervisor emulates the instruction that triggers a `cr3` load by doing nothing, then it hands over to the kernel (through a VM-entry).

Finally, it is worth noting on this KCO that there are some kernel regions that need to be placed inside read-only pages. This is the case, at least, for the region that contains all the kernel page tables, the GDT and the IDT. Also, in this section, we have not covered the case of the collection of page tables that describe the user address-space for each process. In our context, we try to prevent the kernel space from being corrupted. Thus, our hypervisor should verify—in a similar way that has been explained for kernel page tables—that no page table entry, that is written for describing user space layout, contains a physical address of a *kernel* page.

5.2.3 Generic handling of simple kernel-constrained data

Let us note that the security measure we have just presented to preserve the kernel page tables can easily be used for any simple Kernel-Constrained data in memory. The generic approach consists in allocating the specific kernel-constrained data in an empty specific page (for instance in 4 KB pages in the VMALLOC area) and to unset its `R/W` page attribute. Then, the hypervisor preserves the constraint in the same way that has previously been described. With that mechanism, kernel or user code cannot break covered data constraints.

To conclude on this hypervisor activity, it is worth noting that the KCO that we have focused on does not constitute an exhaustive list. We only aim at pointing some KCO of the Linux kernel and how to protect themselves against alteration. We hold to highlight the fact that all KCO could not be easily captured. However, just preserving some well-chosen KCO can protect the kernel against most existing malware at the kernel level in a global way (such as the ones that rely on overwriting either the GDT, or the IDT, or the system call table, or registers like `idtr`, `gdtr`, etc.).

5.3 Prevention of hypervisor memory corruption

5.3.1 Through the control of cpu-based access vectors

In order to prevent the corruption of the hypervisor memory space, this one must virtualize the paging unit. That is, it must retain control over the processor's address-translation mechanisms. In our case, it means that the register `cr3` must only be accessed by the hypervisor and that it needs to emulate the modification of the guest page tables in order to check that the physical addresses that cover its memory space are never used inside them.[23]

[21] Note that in this case the hypervisor needs to perform extra work. It must write information about the page fault—that just triggered—into the VMCS in order for the VM-entry to deliver this event within the guest context.

[22] Note that doing the verification without replaying the instruction would be more complicated and so, more time-consuming as we would have to first determine what is the instruction and then check its arguments.

[23] It is worth noting that the instruction `invlpg` that invalidates an entry in the Translation Lookaside Buffer (TLB) does not need to be emulated, as our hypervisor does only have one guest that coincides with the host. Thus, it does not need to maintain shadow page tables.

Also, the hypervisor must filter some I/O ports[24] (at least the PCI address ports—`0xCF8-0xCFB`, and the PCI data ports—`0xCFC-0xCFF`) in order to protect it against CPU System Management Mode hacks [10,11].

5.3.2 Through the control of DMA-based access vectors

A primary approach is to control and filter I/O port accesses (cf. Footnote 24) that originate from a kernel device driver (or user space) in order to prevent the setting of a DMA transfer from the related device to the hypervisor memory space. In that case, we need to trigger a VM-exit when an access to the specific I/O ports is done, and then to take measures with regard to the physical address that is set to be written by the device. Nonetheless, this approach really seems hard to implement as the I/O ports involved in the establishment of DMA transfers depend on the kind of the bus from which it originates and on the device itself [43]. Also, it prevents insiders from corrupting hypervisor memory space, but it does not protect this space against malicious BusMaster-DMA devices that would take control of a bus such as the Firewire bus [14], without the CPU involvement. To protect against this kind of issue, a system that contains an IOMMU is needed.

5.4 Prevention of kernel memory corruption from hardware features

Section 5.3 discusses solutions in order to protect the hypervisor memory-space against corruption. The envisioned solutions (except for the processor's address-translation mechanisms) can also prevent the kernel memory-space from being corrupted through malicious access to hardware features.

6 Conclusion and future work

In this paper, we have presented security mechanisms that protect the system against some classes of malicious kernel actions. However, these mechanisms are limited. To make them impossible to evade, they must run in a more privileged mode than the kernel itself and thus must use dedicated hardware. That is why we propose to implement them in a light-weight hypervisor called *Hytux*. Such a hypervisor performs different verifications in order to prevent the corruption of some crucial constrained-object of the guest kernel running on top of the hypervisor. We propose a first classification of the possible attacks and for some of them the corresponding virtualization-based solutions. We have also presented a first proof of concept for a IA32 Linux kernel on a 64 bits system that supports the Intel Virtualization Technology. The Hytux demonstrator is currently under development, and we intend to publish it as open source when it is achieved.[25] Although we cannot, for the moment, precisely evaluate the system slowdown that would be induced by Hytux, we can still roughly estimate it through simple considerations. Basically, our hypervisor does not perform a lot of work, it just checks some constraints and then directly hands over to the kernel. Moreover, the impact on the system performance also depends on how the hardware extensions for virtualization perform (i.e., how prompt VM-exit, VM-entry and event injections are). At this level, we can look at existing hypervisors that use hardware virtualization (such as KVM—Kernel Based Virtual Machine [44]). These solutions do not cause major system slowdown and thus similar results are expected with our approach.

Additionally, we work on a hypervisor-based solution that protects the kernel from the malicious actions of the Class 1.1.4. Furthermore, in order to validate our approach based on Kernel-Constrained Objects, we currently work on a model that proposes a formal framework in order to represent interactions between the hardware platform and the different software layers (in our case, the hypervisor, the kernel and the user space layers). We hope this formalization will help us to verify if our approach is efficient in preserving the integrity of the kernel space. We also try to make the model useful for representing Kernel-Constrained Objects as soon as the stage of kernel specification.

Appendix A: Hytux code sample

A "hardware hypervisor" needs to handle specific events from the guest operating system as we have seen in Sect. 2.3. In what follows, we show the way we do it in our demonstrator as well as the way we put the current running Linux kernel into a virtual machine. Note that this sample of code is only given to illustrate the design we adopted, and for that matter we do not try to explain it in details.

[24] Note that an access to any I/O ports can trigger a VM-exit if the VMCS is correctly configured.

[25] The lightweight hypervisor is implemented, and the security mechanisms are currently partially implemented.

```
/* It is the core hypervisor function. It fills the VMCS,
 * puts the current running Linux kernel into the corresponding VM,
 * executes it, and handles the VM-exits. */

int init_and_run_vm(struct vmx_conf *vmx_conf)
{
    hytux_vm.fail = 0;
    hytux_vm.launched = 0;
    hytux_vm.exit_count = 0;

    local_irq_disable();

/* We write all the fields of the VMCS. They represents the state of the VM, plus
 * additional information about event restriction/interception. */

    vmcs_write_hoststate_area(&hytux_vm, vmx_conf);
    vmcs_write_vmexit_ctrl_fields(vmx_conf);
    vmcs_write_vmentry_ctrl_fields(vmx_conf);
    vmcs_write_vmexec_ctrl_fields(vmx_conf);
    vmcs_write_gueststate_area(vmx_conf);

/* We put the current running Linux kernel into the just configured VM
 * (we assume that VMCLEAR has been executed on that VMCS)
 * Then the hypervisor hands over the processor to the VM (ASM_VMX_VMLAUNCH). */

    asm volatile(
        /* vmwrite of GUEST_RSP */
        "mov %[GUEST_RSP], %%rdx \n\t"
        ASM_VMX_VMWRITE_RSP_RDX "\n\t"
        /* vmwrite of GUEST_RFLAGS */
        "pushq %%rax \n\t"
        "pushfq \n\t"
        "popq %%rax \n\t"
        "mov %[GUEST_RFLAGS], %%rdx \n\t"
        ASM_VMX_VMWRITE_RAX_RDX "\n\t"
        "popq %%rax \n\t"
        "movb $1, %c[guest_mode](%[vm]) \n\t"
        ASM_VMX_VMLAUNCH "\n\t"
        ".Lvmlaunch_fail: "
        "setbe %c[fail](%[vm]) \n\t"
        "movb $0, %c[guest_mode](%[vm]) \n\t"
        ".Lvmx_guest_entry: "
        : :
          [vm]"c"(&hytux_vm),
          [fail]"i"(offsetof(struct vmx_vm, fail)),
          [GUEST_RSP]"i"((unsigned long)GUEST_RSP),
          [GUEST_RFLAGS]"i"((unsigned long)GUEST_RFLAGS),
          [guest_mode]"i"(offsetof(struct vmx_vm, in_guest_mode))
        : "cc", "rax", "rdx", "memory");

/* If VMLAUNCH has not failed we are in guest mode for the first time
 * (the VM has been set to enter here), so we return to the init module
 * function. */
```

```
    if (hytux_vm.in_guest_mode) {

        hytux_vm.launched = 1;
        local_irq_enable();

        return 0;
    }

/* VMLAUNCH failed during the first step of the guest launching
 * (intel chap22), so we inform the user. */

    vmx_dump_guest_register();

    /* We do not use a simple "else" because gcc will make
     * optimization that screw things up, i.e., it will end the
     * function before .Lvm_exit_handler (thus this label will be
     * undefined at link time). */

    if (hytux_vm.fail == 1) {

        printk(KERN_ERR "Hytux: VMLAUNCH failed\n");

        hytux_vm.exit_info.fail_entry_reason = vmcs_read32(VM_INSTRUCTION_ERROR);
        printk(KERN_ERR "Hytux: INSTRUCTION_ERROR = %d\n",
                hytux_vm.exit_info.fail_entry_reason);

        local_irq_enable();
        return -1;

    } else if (hytux_vm.fail == 0) {

        printk(KERN_ERR "Hytux: VMLAUNCH failed but no indication of failure
               in RFLAGS\n");
        local_irq_enable();
        return -1;
    }

/* This is the entry point for VM-exits. We first store some registers
 * that are not saved in the VMCS at VM-exit. Then we handle these VM-exits
 * through the function vmx_check_error_fields() (which implements the
 * verification and preservation of constraints). Finally we reload the VM
 * previously stored registers and resume VM execution
                        (through ASM_VMX_VMRESUME). */

    asm volatile(".Lvm_exit_handler: ");

    /* (23.5.3) When a VM-exit occurs, rflags is cleared except
     * bit 1 (so rfalgs.IF = 0, i.e., local interrupts are
     * disabled). */

    store_vm_regs(&hytux_vm);
```

```
    /* Now, gcc cannot rely on any previous registers' value as
     * they are all clobbered in store_vm_regs(). This is what we
     * want as we land here because of a VM-exit (all registers'
     * value come from the guest context). */
    hytux_vm.exit_count++;

    /* Here, we handle the VM-exit. */

    hytux_vm.ret = vmx_check_error_fields(&hytux_vm);

    if (hytux_vm.ret < 0) {
        local_irq_enable();
        panic("Hytux Dead! (VM-exit not handled)");
    }

    load_vm_regs(&hytux_vm);

    asm volatile(".Lvmx_resume: " ASM_VMX_VMRESUME "\n\t");

    /* VMRESUME failed during the first step of the guest
     * launching (chap22), so we inform the user. */

    vmx_dump_guest_register();

    local_irq_enable();
    panic("Hytux Dead (VMRESUME failed)!");
}
```

The functions `store_vm_regs()` and `load_vm_regs()` are really part of the previous function (they are inlined), and are shown in what follows (for the sake of clarity, only their 64-bit version is shown).

```
static inline void store_vm_regs(struct vmx_vm *vm)
{
/* We do not store rsp, cr3, rflags, as they are VMCS fields. */

        asm volatile(
                    "pushq %%rcx \n\t"
                    :::"rcx");

    asm volatile(
            /* Save guest registers */
            "mov %%rax, %c[rax](%0) \n\t"
            "mov %%rbx, %c[rbx](%0) \n\t"
            "popq %c[rcx](%0) \n\t"
            "mov %%rdx, %c[rdx](%0) \n\t"
            "mov %%rsi, %c[rsi](%0) \n\t"
            "mov %%rdi, %c[rdi](%0) \n\t"
            "mov %%rbp, %c[rbp](%0) \n\t"
            "mov %%r8,  %c[r8](%0) \n\t"
            "mov %%r9,  %c[r9](%0) \n\t"
            "mov %%r10, %c[r10](%0) \n\t"
```

```
            "mov %%r11, %c[r11](%0) \n\t"
            "mov %%r12, %c[r12](%0) \n\t"
            "mov %%r13, %c[r13](%0) \n\t"
            "mov %%r14, %c[r14](%0) \n\t"
            "mov %%r15, %c[r15](%0) \n\t"
            "mov %%cr2, %%rax   \n\t"
            "mov %%rax, %c[cr2](%0) \n\t"
            : : "c"(vm),
              [rax]"i"(offsetof(struct vmx_vm, arch.regs[VM_REGS_RAX])),
              [rbx]"i"(offsetof(struct vmx_vm, arch.regs[VM_REGS_RBX])),
              [rcx]"i"(offsetof(struct vmx_vm, arch.regs[VM_REGS_RCX])),
              [rdx]"i"(offsetof(struct vmx_vm, arch.regs[VM_REGS_RDX])),
              [rsi]"i"(offsetof(struct vmx_vm, arch.regs[VM_REGS_RSI])),
              [rdi]"i"(offsetof(struct vmx_vm, arch.regs[VM_REGS_RDI])),
              [rbp]"i"(offsetof(struct vmx_vm, arch.regs[VM_REGS_RBP])),
              [r8]"i"(offsetof(struct vmx_vm, arch.regs[VM_REGS_R8])),
              [r9]"i"(offsetof(struct vmx_vm, arch.regs[VM_REGS_R9])),
              [r10]"i"(offsetof(struct vmx_vm, arch.regs[VM_REGS_R10])),
              [r11]"i"(offsetof(struct vmx_vm, arch.regs[VM_REGS_R11])),
              [r12]"i"(offsetof(struct vmx_vm, arch.regs[VM_REGS_R12])),
              [r13]"i"(offsetof(struct vmx_vm, arch.regs[VM_REGS_R13])),
              [r14]"i"(offsetof(struct vmx_vm, arch.regs[VM_REGS_R14])),
              [r15]"i"(offsetof(struct vmx_vm, arch.regs[VM_REGS_R15])),
              [cr2]"i"(offsetof(struct vmx_vm, arch.cr2))
            : "cc", "memory"
              , "rax", "rbx", "rdx", "rdi", "rsi"
              , "r8", "r9", "r10", "r11", "r12", "r13", "r14", "r15"

/* 'rbp' must be added to the clobbered list if the kernel is compiled
 * without CONFIG_FRAME_POINTER, as gcc could use 'rbp' for anything and
 * screw things up (and that's exactly what it does in this situation). */

#ifndef CONFIG_FRAME_POINTER
              , "rbp"
#endif
            );

}


static inline void load_vm_regs(struct vmx_vm *vm)
{

/* We do not load rsp, cr3, rflags, as they are VMCS fields */

    asm volatile(
            /* Load guest registers. */
            "mov %c[cr2](%0), %%rax \n\t"
            "mov %%rax, %%cr2 \n\t"
            "mov %c[rax](%0), %%rax \n\t"
            "mov %c[rbx](%0), %%rbx \n\t"
            "mov %c[rdx](%0), %%rdx \n\t"
            "mov %c[rsi](%0), %%rsi \n\t"
```

```
        "mov %c[rdi](%0), %%rdi \n\t"
        "mov %c[rbp](%0), %%rbp \n\t"
        "mov %c[r8](%0),  %%r8  \n\t"
        "mov %c[r9](%0),  %%r9  \n\t"
        "mov %c[r10](%0), %%r10 \n\t"
        "mov %c[r11](%0), %%r11 \n\t"
        "mov %c[r12](%0), %%r12 \n\t"
        "mov %c[r13](%0), %%r13 \n\t"
        "mov %c[r14](%0), %%r14 \n\t"
        "mov %c[r15](%0), %%r15 \n\t"
        "mov %c[rcx](%0), %%rcx \n\t"
        : : "c"(vm),
          [rax]"i"(offsetof(struct vmx_vm, arch.regs[VM_REGS_RAX])),
          [rbx]"i"(offsetof(struct vmx_vm, arch.regs[VM_REGS_RBX])),
          [rcx]"i"(offsetof(struct vmx_vm, arch.regs[VM_REGS_RCX])),
          [rdx]"i"(offsetof(struct vmx_vm, arch.regs[VM_REGS_RDX])),
          [rsi]"i"(offsetof(struct vmx_vm, arch.regs[VM_REGS_RSI])),
          [rdi]"i"(offsetof(struct vmx_vm, arch.regs[VM_REGS_RDI])),
          [rbp]"i"(offsetof(struct vmx_vm, arch.regs[VM_REGS_RBP])),
          [r8]"i"(offsetof(struct vmx_vm, arch.regs[VM_REGS_R8])),
          [r9]"i"(offsetof(struct vmx_vm, arch.regs[VM_REGS_R9])),
          [r10]"i"(offsetof(struct vmx_vm, arch.regs[VM_REGS_R10])),
          [r11]"i"(offsetof(struct vmx_vm, arch.regs[VM_REGS_R11])),
          [r12]"i"(offsetof(struct vmx_vm, arch.regs[VM_REGS_R12])),
          [r13]"i"(offsetof(struct vmx_vm, arch.regs[VM_REGS_R13])),
          [r14]"i"(offsetof(struct vmx_vm, arch.regs[VM_REGS_R14])),
          [r15]"i"(offsetof(struct vmx_vm, arch.regs[VM_REGS_R15])),
          [cr2]"i"(offsetof(struct vmx_vm, arch.cr2))
        : "cc", "memory"
          , "rax", "rbx", "rdx", "rdi", "rsi"
          , "r8", "r9", "r10", "r11", "r12", "r13", "r14", "r15"
        );

}
```

References

1. Rutkowska, J.: Subverting vista kernel for fun and profit. In: Black Hat in Las Vegas (2006)
2. Lacombe, É., Raynal, F., Nicomette, V.: Rootkit modeling and experiments under Linux. J. Comput. Virol. **4**(21), 137–157 (2008) http://www.ingentaconnect.com/content/klu/11416/2008/00000004/00000002/00000069
3. Intel: Intel trusted execution technology—measured launched environment developer's guide (2008)
4. Intel: Intel 64 and IA-32 Architectures software developer's manual, vol. 3A: System programming guide, Part 1 (2008)
5. Intel: Intel 64 and IA-32 Architectures software developer's manual, vol. 3B: System programming guide, Part 2 (2008)
6. Duflot, L.: CPU Bugs, CPU backdoors and consequences on security. In: ESORICS 2008 (2008)
7. Truff: Infecting loadable kernel modules. Phrack **61** (2003)
8. sd, devik: Linux on-the-fly kernel patching without LKM. Phrack **58** (2001)
9. c0de: Reverse symbol lookup in Linux kernel. Phrack **61** (2003)
10. BSDaemon, coideloko, D0nAnd0n: System management mode Hacks. Phrack **65** (2008)
11. Duflot, L., Etiemble, D., Grumelard, O.: Using CPU system management mode to circumvent operating system security functions. In: CanSecWest/core06 (2006)
12. sqrkkyu, twzi: Attacking the core: kernel exploiting notes. Phrack **64** (2007)
13. Lacombe, É.: Le fonctionnement de PaX : Protection against eXecution. GNU/Linux Magazine France **79** (2006) http://www.unixgarden.com/index.php/securite/le-fonctionnement-de-pax-protection-against-execution
14. Piegdon, D.R.: Hacking in physically addressable memory: a proof of concept. In: Easterhegg (2008)
15. Dornseif, M., et al.: FireWire: all your memory are belong to us. In: CanSecWest/core05 (2005)
16. Boileau, A.: Hit by a Bus: physical access attacks with firewire. In: Ruxcon (2006)
17. Rutkowska, J.: Beyond the CPU: defeating hardware based RAM acquisition tools (Part I: AMD case). In: Black Hat DC (2007)
18. Intel: IA-32 Intel architecture software developer's manual, vol. 2b: Instruction Set Reference, n-z (2008)

19. PCI-SIG: PCI Local Bus Specification. Technical Report revision 2.2, PCI Special Interest Group (1998)
20. pragmatic, THC: (nearly) Complete linux loadable kernel modules. The definitive guide for hackers, virus coders and system administrators (1999)
21. Cesare, S.: Kernel function hijacking (1999) http://vx.netlux.org/lib/vsc08.html
22. Hoglund, G., McGraw, G.: Exploiting software: how to break code. Pearson education. Addison-Wesley, Reading (2004)
23. Corbet, J.: vmsplice(): the making of a local root exploit (2008) http://lwn.net/Articles/268783/
24. Corbet, J.: The rest of the vmsplice() exploit story (2008) http://lwn.net/Articles/271688/
25. Nergal: The advanced return-into-lib(c) exploits: PaX case study. Phrack **58** (2001)
26. Designer, S.: Getting around non-executable stack (1997) http://seclists.org/bugtraq/1997/Aug/0063.html
27. Pol, J.: [PINE-CERT-20040201] reference count overflow in shmat() (2004) http://seclists.org/bugtraq/2004/Feb/0140.html
28. kad: Handling interrupt descriptor table for fun and profit. Phrack **59** (2002)
29. Duflot, L., Levillain, O., Morin, B., Grumelard, O.: Getting into the SMRAM: SMM reloaded. In: CanSecWest/core09 (2009)
30. Duflot, L., Etiemble, D., Grumelard, O.: Utiliser les fonctionnalités des cartes mères ou des processeurs pour contourner les mécanismes de sécurité des systèmes d'exploitation. In: SSTIC (2006)
31. Embleton, S., Sparks, S., Zou, C.: SMM Rootkits: a new breed of independent malware. In: SecureComm (2008)
32. Filiol, É.: Computer viruses: from theory to applications. IRIS international series. Springer, France (2005)
33. Spengler, B., et al.: Grsecurity features (2009) http://www.grsecurity.net/features.php
34. Corporation, M.: Digital signatures for kernel modules on systems running Windows Vista. Technical report, Microsoft Corporation (2006)
35. Spengler, B., et al.: PaX documentation (2003) http://pax.grsecurity.net/docs
36. Shacham, H., Page, M., Pfaff, B., Goh, E.J., Modadugu, N., Boneh, D.: On the effectiveness of address-space randomization. In: CCS '04: Proceedings of the 11th ACM conference on computer and communications security, pp. 298–307. ACM, New York (2004)
37. Cowan, C., Beattie, S., Johansen, J., Wagle, P.: PointGuard: protecting pointers from buffer overflow vulnerabilities. In: 12th USENIX Security Symposium (2003)
38. Spengler, B.: PaX's UDEREF: technical description and benchmarks (2007) http://www.grsecurity.net/~spender/uderef.txt
39. Cowan, C., Pu, C., Maier, D., Walpole, J., Bakke, P., Beattie, S., Grier, A., Wagle, P., Zhang, Q., Hinton, H.: StackGuard: automatic adaptive detection and prevention of buffer-overflow attacks. In: Proceedings of the 7th USENIX security symposium (1998)
40. Bulba, Kil3r: Bypassing stackguard and stackshield. Phrack **56** (2000)
41. anonymous: Once upon a free()... Phrack **57** (2001)
42. Intel: Intel virtualization technology for directed I/O: architecture specification (2007)
43. Duflot, L., Absil, L.: Programmed I/O accesses: a threat to virtual Machine Monitors? In: PacSec 2007 (2007)
44. Kivity, A., et al.: KVM: the linux virtual machine monitor. In: Linux Symposium (2007)