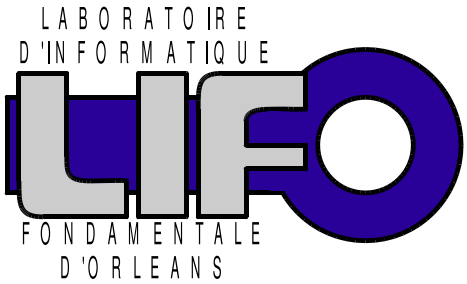
Security and Virtualization: a Survey

Patrice Clemente, Jonathan Rouzaud-Cornabas



4 rue Léonard de Vinci
BP 6759
F-45067 Orléans Cedex 2
FRANCE
http://www.univ-orleans.fr/lifo

Rapport de Recherche

Security and Virtualization: a Survey

Patrice Clemente
LIFO – ENSI de Bourges,
Jonathan Rouzaud-Cornabas
LIFO – Université d'Orléans

Rapport n^o **RR-2010-06**

1 Introduction

In this report, we investigate the security aspects and challenges about computer virtualization. In a few words, virtualization is what allows the execution of multiple operating systems on a single machine at the same time. A virtualization component can be viewed as a layer or a container making some kind of emulation, allowing to execute programs or operating systems on the virtualized layer, for example executing Microsoft Windows and Linux on one single machine at the same time. On one hand, as virtualization can provide a kind of isolation between users/applications/operating systems, it can address some security containment problems. But on the other hand, there exist today many security flaws and attacks focused on such systems, as the virtualization layer controls and monitors all virtualized applications/operating systems.

The outline of this report is the following. We first review precisely all the various forms the virtualization layer and related components can take. And what technical aspects it involves. Then we survey more precisely all the vulnerabilities and exploits that currently exist or may appear in the near future. And then we study what are the proposed security protection by actual virtualization technologies. We conclude the report with what is remaining to be done and what are the forthcoming security challenges.

1.1 Virtualization History

Computer virtualization is not a new research field. First works about virtualization appeared conjointly with the foundations of modern computing systems. It was introduced to optimize the usage of expensive computing resources and isolates users from each others, using the time-sharing technology appeared earlier in IBM System/360 mainframes (1965), with CTSS ([CMC62]).

Virtualization firstly appeared as hardware processor instructions, in the IBM VM/370 [IBM72] and at the same time in the DEC PDP 8 computer series. DEC PDP-10 computer series were the first computers PDP with un-bugged virtualization instructions. Several programs could co-exist on the same computer.

Actually, virtualization, since those time was mainly made possible using switching instructions. Those instructions, always used today, allow to save (push) the processor context before executing another program and restore its context in order to pursue the execution of the first process. Thus, each program could use a virtual entire system, even if other programs was using it too, at the same global time (but not with real parallelism: one instruction at a time) [Gal69].

Except in IBM VM/370 and successor mainframes ([Var97] for a complete VM history), virtualization have been quite forgotten by the most part of computer

industry and manufacturers. But, recently, an economic explosion (with VMWare in head) has made virtualization quite interesting for attackers, as it has widespread the IT market.

1.2 What is Virtualization?

Today, one could define "Virtualization" as the act of presenting a software or harware entity (i.e. a program, an operating system, a device) as being real where actually it is not. Virtulization is a synonym of emulation.

1.2.1 Basic concepts

In the following, the virtualization layer is referred as hypervisor or the Virtual Machine Monitor(s) (VMM). The hypervisor is a software providing an abstraction of the underlying layer, whether it is software or hardware.

The virtualized machine(s)/operating system(s)/application are referred as VM. They are executed on the virtual hardware provided by the hypervisor.

1.2.2 Virtualization principle

Generally, the virtualization process, from a computer architecture point of view, can be defined as the following:

> **Virtualization principle.** *The virtualization process is the act of abstracting the underlying layer (i.e. the hardware layer). It inserts a layer between existing layers (hardware and operating systems/softwares) to resolve problems such as the support of legacy functionality, standard interfaces or isolated users/resources and servers consolidation.*

1.2.3 Virtualization requirements

The founding work of G. Popek and R. P. Goldberg ([PG73]) about virtualization concepts introduced a set of three sufficient conditions for a computer architecture to efficiently support system virtualization. These requirements, based on simple assumptions, still remain accurate to determine whether a computer architecture supports efficient virtualization or not. They also provide guidelines for the design of virtualized computer architectures.

1. **Efficiency**: VM must not suffer noticeable performances degradation i.e. a large subset of instructions must run directly on real processor without the need to ask permission to the virtualization layer (VMM).

2. **Resource control**: The hypervisor must allocate the resources when asked by virtualized programs and cleaning them up when they are not used anymore. A VM must not be able to interfere with other VM's resources. Thus, the VMM must totally control the virtualized resources.

3. **Equivalence**: the VMM must provide a virtual hardware abstraction layer that allows program to be executed exactly the same way as it should be on a real hardware.

Popek and Goldberg explain what characteristics the Instruction Set Architecture (ISA) of the hosting physical machine should provide in order to run VMMs which possess the above properties. Popek and Goldberg introduce a model of "third generation architectures" that can be extended to classical modern machines. This model includes a processor that operates in either system or user mode, and has access to linear, uniformly addressable memory. Some instructions are only available in system mode. The memory addresses are relative to a relocation register.

Popek and R. P. Goldberg ([PG73]) also introduced two classes of instructions based on how they affect the system state:

1. **Privileged**: Instructions that only trap (into kernel mode) if the processor is in user mode.

2. **Sensitive**: Instructions that can attempt to change the allocation/configuration of the whole system's resources.

Their main result leads to the following formal theorem, with some security implications that is discussed later:

Theorem 1 *For any virtualizable third generation architecture, a VMM may be constructed if the set of sensitive instructions is a subset of the set of privileged instructions.*

Intuitively, this theorem expresses that to build a hypervisor it is needed and sufficient that all instructions that could affect the correct functioning of the hypervisor (i.e. sensitive instructions) always trap and pass control to the hypervisor itself. This guarantees that the resource control cannot be done outside of the hypervisor, and prevent it to be corrupted. On the other hand, non-privileged instructions must be executed natively (i.e., efficiently).

1.2.4 Virtualization advantages and disadvantages

Advantages. Virtualization is an architectural design of computing systems that brings some architectural benefits.

1. **Costs reducing.** Virtualization allows the regroupment of applications/users/operating systems on a reduced number of physical computers. A modern term for such a design is consolidation, of servers for example. With virtualization, modern industries nowadays want to limit the number of servers and their related costs (energy, infrastructure, technical staff) with grouping services that were previously dispersed. Following the virtualization paradigm, the hosting model had switched from 1 service per server to n **services per server**.

2. **«Write once, run everywhere».** In addition, it is possible to make only one single OS/hardware specific implementation and then, using a virtualization layer, it is possible to spread that implementation over all machines, whatever the real OS/hardware of the hosting machine. This is the so-called **write once, run everywhere** principle, which another way of saving money, by saving software development time.

3. **Efficiency.** This property has already been seen above: it guarantees an efficient use of hardware resources for VM.

4. **Isolation.** Using virtualization technologies for security was introduced by Pr. Suart Madnick and J.J. Donovan (MIT) when working on Project MAC in 1973 ([MD73]). Nowadays, the hypervisor is often used as a reference monitor with the purpose to isolate the workload within a VM among other ones, whereas OS only provide a weak isolation between processes. This isolation capability allows the load balancing over the hosting machines.

5. **Reliability.** Another advantage is that one virtualized failing service does not imply the fall of all other virtualized ones. This is a consequence of isolation.

Disadvantages. On the other hand, these architectural aspects negatively impact on some other points.

1. **Overhead.** When using virtualization, there should exist an overhead due to the virtualization layer, which needs computing time to do its job. In addition, in virtualization technologies that do not fully virtualize the bare hardware, there is some overhead due to the translation of instructions into real processors ones.

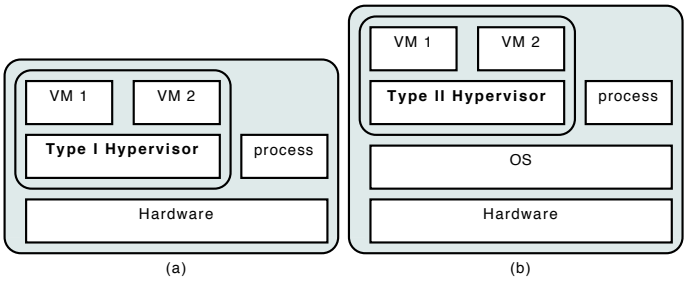


Figure 1: Type I (a) and type II (b) hypervisors

2. **Not a real efficiency.** In some virtualization technologies, services/applications/OS need sometimes to be partially rewritten/patched to work, in order to be executed in virtual environments. That can lead to unportable or difficult to upgrade versions of applications.

1.3 Types of Hypervisors

There are two main types of hypervisors (or VMM) (cf. figure 1):

- **Type I hypervisors** (or *native*, or *bare-metal* hypervisors) are softwares running or real host's hardware. They act as an operating system for VM and directly control the accesses to the real hardware (cf. figure 1.(a)).
- **Type II hypervisors** (or hosted hypervisors) are software running *above* a conventional operating system and are simply processes running on the real host's OS (cf. figure 1.(b)). Another processes can co-exist at the same level. Guest OS and VM are thus at a layer above (a third one) than the hypervisor.

1.4 Actual Virtualization architectures

As the virtualization field has widely spread in the IT market, there exist today many architecture that implement the concept of virtualization. In a few words, here are the most common architectures, before going into further details on each one :

- **Full virtualization** provides a virtualization of the full hardware.
- **Paravirtualization** provides a virtualization of the almost full hardware, but with some hardware access not virtualized in order to make them more efficient.

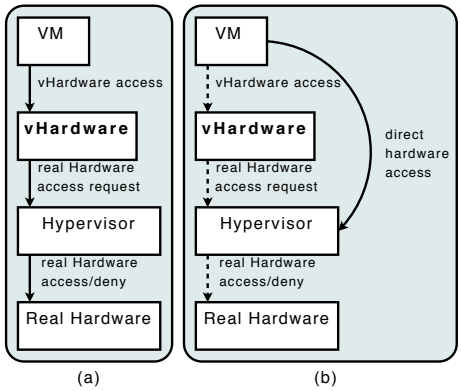


Figure 2: Full virtualization (a) *vs* Para-virtualization (b)

- **Partial virtualization** provides only a virtualization of only a sub part of the system to be virtualized.
- **Operating System level virtualization** is mainly a server technology used to provide multiple isolated user sessions. They all lye on the same operating system. Each guest is a sophisticated chroot like.

1.4.1 Full virtualization

Full virtualization is a technology providing a total virtualization of the full hardware. Each virtualized entity can thus run as it should be on a real hardware. That includes operating systems. This is different than other virtualization technics, on which only a part of the hardware is virtualized, implying that some softwares/OS to be virtualized need to be partially rewritten. The figure 2.(a) describes the full virtualization principle. Each VM accesses to the virtual hardware, which asks the hypervisor to access to the real one.

The are two main families of full virtualization as described below:

- **Software-assisted virtualization**: is the historical first full virtualization approach introduce with with IBM CP/CMS in 1969: all is simulated. This means that every operating system and applications can be run without modifications. It is very expensive in terms of resources because every operations made by the guest operating system need to be simulated. Each of these operation also needs to be checked to be sure it will not interfere with others guest operating systems and the hypervisor. VMWare introduced a method that simulates full virtualization but is not quite it: binary translation. It

works by modifying on-the-fly the x86 instructions. It permits nearly the same effect than full virtualization with a reduce overhead and makes it possible for x86 architecture. There also exists a particular kind of software assisted virtualization: application virtualization. It provides a container for only a single application to be virtualized.

- **Hardware-assisted virtualization**: This method allows to run an unmodified operating system into a virtual machine. Unlike software assisted virtualization, the overhead is reduced because it extends the hardware (mainly the processor's ones) with new instructions (known as Intel-VT or AMD-V instructions[1]. Those new instructions bring new functionalities allowing to directly access to the hardware without the need of many software traps. It was introduced by IBM in 1972 [IBM72] but only recently in x86 architectures ([AA06]) because they didn't meet the classical virtualization definition of Popek and Goldberg ([PG73]).

1.4.2 Paravirtualization

Paravirtualization was introduced to simplify the usage of virtualization and bought close to non-virtualized performances. The virtual machine (VM) needs to be modified and especialy its kernel to introduce the notion of hypercall. Each call to the system (system call, or *syscall*) in order to access the hardware are replaced in the source code of the application/OS to be virtualized by call to hypervisor, i.e., *hypercalls*. Hypercalls allow the guest operating system to directly send syscall to the hypervisor without the need of complicated hardware simulation, as described on the figure 2.(b). For some hardware access, the VM accesses to the virtual hardware but for some para-virtualized hardware (e.g. the disks) it skip the simulation layer (the v-hardware layer). By doing so, it allows greater performances. It was firstly introduced in 1972 by IBM [IBM72] and currently, it is wide used by Parallels Workstation and Xen [MC00].

On x86 architectures, when using a non-hardware virtualization-ready CPU, the hypervisor run at the *ring0* level[2]). While running at Ring0, the hypervisor will have a full control over the VM it hosts while being protected from them.

[1]PDP-10 [Gal69] and Ultra SPARC workstations previously also introduced some harware assisted virtualization facilities.

[2]Conventionally, in computer science, various privilege levels of execution, called protection rings, are considered. This is a mechanism of data protection and operation control. *Ring0* is the most privileged and interacts more directly with the hardware while *Ring3* is the less privileged one and is used for user applications and for non-virtualized Operating Systems. *Ring1* and *Ring2* are often used for device drivers. Ring1 is also used for paravirtualized virtual Machine kernels. Device drivers need to ask Ring0 applications (i.e. the OS kernel) for hardware access. With virtualization, new protection rings appeared, such as *Ring-1* for hardware hypervisors and even *Ring-2* for the

1.4.3 Partial virtualization

With partial virtualization, the VM simulate only a part of the hardware of the host. The virtualized entity thus needs some modifications, if using a part of the non simulated hardware. Running an operating system under this virtual machine is, in most of the cases, not possible. The most known implementations of this kind of virtualization is the address space virtualization, where each process has a full address space without needing the total amount of memory space on the hardware. Historically, it was a first step in the virtualization field and was introduced in CTSS and IBM M44/44X [MS70]. It helps to bring the capabilities of multi users or/and processes on the same OS.

1.4.4 Operating system-level virtualization

The kernel acts as a hypervisor and is shared between the virtual machines known as containers in this case. Each container acts as a fully functional real server. It permits to share a hardware computer between multiple users without the risk of interference between each others. This technology sometimes also includes resources managements features based on quota to share each type of ressources between the container. Another point worth noticing is that this kind of virtualization brings nearly no overhead. Most of the well known operating systems are able to support this kind of virtualization like OpenVZ for Linux, Jail for BSD, Zone for Solaris, Virtuozzo for Windows, Rosetta for Mac OS X and many others.

The drawback of this virtualization solution is the impossibility to run heterogeneous operating systems on the same hardware.

1.5 Harware resources virtualization

One can separate the hypervisors' policy of sharing resources into two main categories: "Pure isolation hypervisors" and "Sharing hypervisors" ([Kar05]).

With pure isolation hypervisors (see figure 3), each guest have its own dedicated I/O hardware and device drivers, they are not shared between guests. Pure isolation hypervisors are the simplest and most secure approach because devices are located in the guest.

With sharing hypervisor, the I/O hardware and device drivers are shared. They are two main methods to do that.

System Management Mode (SMM) which is loaded by the BIOS into a protected memory area (i.e. the SMRAM: System Management RAM). SMM is an operating mode in which all normal execution (including the operating system) are suspended, and a special separate software (usually a firmware or a hardware-assisted debugger) is executed in ultra high-privileged mode.

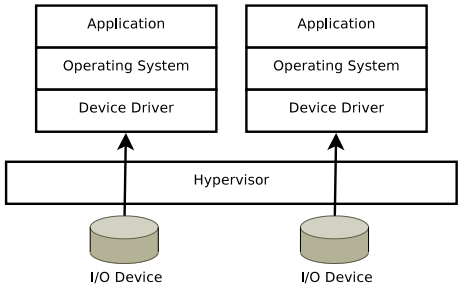


Figure 3: Pure hypervisor isolation

With the first method (see figure 4), the device driver are put into the hypervisor and it only accesses to the I/O hardware.

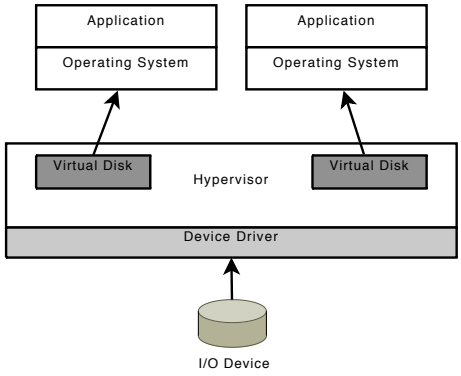


Figure 4: First Sharing method

With the second method (see figure 5), a special privileged guest is created to manage all the sharing devices for other guests through the VMM. All devices drivers only run into this special privileged guest.

1.6 General security considerations about virtualization

Virtualization seems to increase the security of hosting multiple users/applications/OS on the same hardware by creating containers that help the non interference between each others. But virtualization is not a security component. Worth, it can increase the number of potential weaknesses in the system by bringing a complex code. The hypervisors and others virtualization related facilities can add new attack

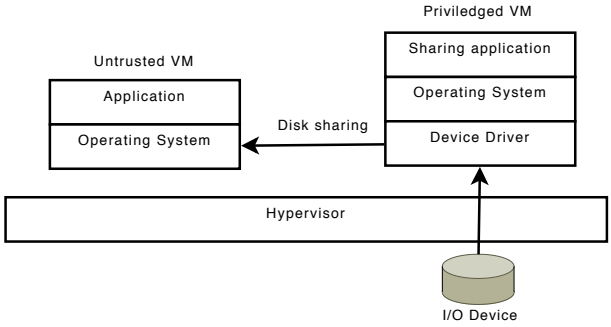


Figure 5: Second Sharing method

vectors. In consequence, it is very important to not consider and use virtualization as a security technology. Moreover, it is also essential to secure any virtualization technology/archicture [GR05, SJV+05a].

1.6.1 General threats of virtualization

We consider the following main threat classes for virtulization:

(a) To corrupt the VM;

(b) To escape the VM and take control of the computer's ressources;

(c) To directly or indirectly alter other VM running on the same hardware;

(d) To migrate a non-virtualized OS into a VM at run-time;

(e) To corrupt the hypervisor.

The successful usage of those threats is not new. Back in 1970s, IBM penetrates their VM/370 system using I/O facilities' flaws and realize the threats (b) and (c) []. Later, in 2006, two research groups use virtualization hardware technologies to realize the threat (d) [KC06, Fer06]. The threat (a) has been achieved many times.

1.6.2 VM machines remain machines...

A VM can be a full operating system. In consequence, it has the same attack vectors available than in any other operating system running on real hardware. Using the same security facilities that on classical operating systems (IDS, firewalls, antivirus, ...) is not always a good choice. First, some tools may not work properly

depending on the virtualization technology used. Second, it will need an instance of each security facility on each VM, that will bring a major overhead. Thus VM are not so secure, this is why points (a) and (b) above are possible.

1.6.3 VM can attack other VM?

VM are not so secure, nor they are isolated from each others. Virtualizing distributed physical machines and group them inside a single one may add new security potentials as the resulting VM are not physically isolated any more. This is why point (c) above is possible.

1.6.4 Hypervisor corruption

Running at the most privileged level, hypervisors are key elements in the *trusted computing base* (TCB[3]) to enforce explicit or implicit system security policies. But they also can be the most critical security concerns.

The corruption of the hypervisor, especially in cases of type I hypervisors, running directly on the real hardware, can lead to the most harmful exploits. In those cases, the attacker must manage to take control of the hypervisor from inside his VM or exploit a bug in the hypervisor and/or and simulated drivers, or both.

For type II hypervisors, it may easier to take control over the hypervisor as it is simply a process running on the host OS. Then, it may be used to bring a nearly invisible hypervisor with malicious purposes to replace the existing one or directly modify it. This would allow the attacker to spy and control any VM. But the host may be more protected than with type I hypervisors.

In the next section, we study the attacks related to the general threats discussed above. We see what are the actual attacks and which risks can lead to other new ones.

2 Attacks

Currently, attacks on virtualization are rare but with the growing usage of this technology, they emerge fast. Indeed, with CPU including hardware virtualization acceleration, almost all operating systems may be virtualized.

[3]The TCB is the minimal part of the OS's code allowing the system to run perfectly and **safely**. The TCB thus needs to be trusted because its corruption can lead to many security issues. No one should run a system he does not trust.

In the literature, the security classification that is used for attacks, is based on their effects on the system, in terms of security properties. They are divided into three categories:

- Attacks on **Integrity** : attacks that write or modify unauthorized data;
- Attacks on **Confidentiality** : attacks that permit to read unauthorized data;
- Attacks on **Availability** : attacks that permit to disrupt the normal running of computation.

In this section, we present the attacks vectors that permit crafted attacks on virtualization [Fer06]. These attacks can be classified into three main categories:

- from a VM's operating system to the hypervisor;
- from the hypervisor to a VM;
- from one VM to another.

First, we introduce "from VM to the hypervisor" attacks, then "from the hypervisor to VM" attacks and finally "from VM to VM" attacks.

2.1 From VM to the Hypervisor

It is possible for an attack to corrupt or interfere from a VM to the hypervisor. This class of attacks is the worst case scenario for virtualization technology because it allows an attacker to read and/or modify unauthorized data and programs. The effect of such attacks is not limited to the virtual machine but also to the other ones and the hypervisor itself.

For example, it can allow an attacker at the hypervisor level to modify the access control of a virtual machine in order to give it higher privileges. Also, it can allow an attacker to silently spy another virtual machine without any possibility of detecting it from inside the spied VM.

2.1.1 Direct Memory Access

Some drivers have the ability to access the underlying hardware (i.e. the real hardware). It can be done using Direct Memory Access DMA) [Mur08]. This attack allows to read and write the entire physical memory.

As shown on figure 6, on the virtualized architecture, there are two ways to access memory:

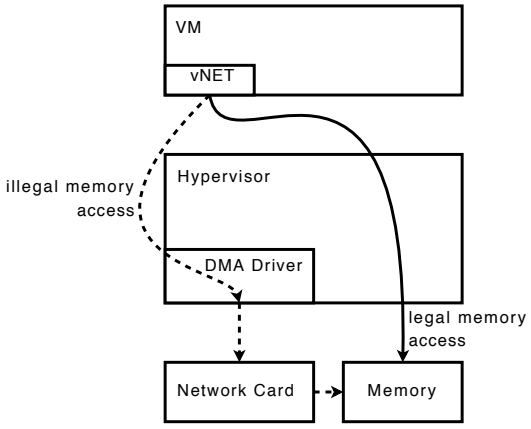


Figure 6: An attack from a VM to memory through DMA

- normal way: any access to the memory is controlled by the hypervisor to guarantee that a VM can only access to its own memory range.
- malicious way: the attacker can reach the entire physical memory exploiting a bugged DMA driver that acts as a wrapper between the virtual network interface (vNet) and the physical network card. The physical network card can indirectly access to the memory through DMA without any control of the hypervisor.

Using this attack, it is possible to corrupt both the integrity and confidentiality of the hypervisor and all the running VM on the physical hardware. Also, it can be used to induce deny of service (availability) that disrupt the hypervisor and all VM e.g. by deleting the memory kernel space of a VM.

[Woj08] describes an implementation of the DMA attack using a rogue network driver. Network drivers have the ability through loopback mode to copy data between two locations in RAM. By modifying the network driver's code, it is possible to read/write any part of the physical memory. As shown on the figure 7, the rogue network driver is loaded at the kernel level in the VM. It has access to the legal network driver structure. In consequence, the rogue network driver has the ability to send DMA transfers between two locations in RAM. Thus, it can attack the integrity and confidentiality of all the VM and the hypervisor. This modification makes possible to run any codes at any level in all the VM and even at the hypervisor's one.

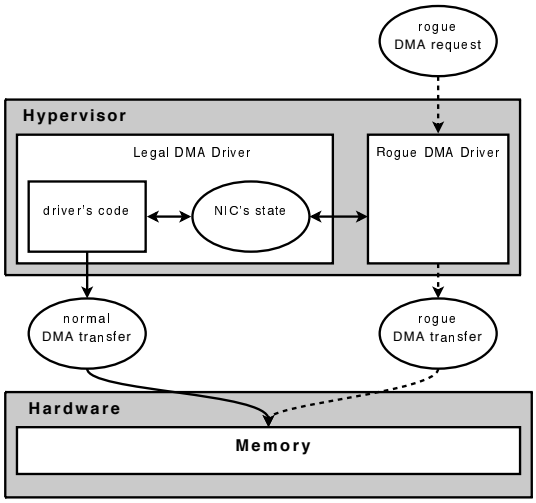


Figure 7: Implementing a DMA attack

2.1.2 Other vectors

The previous type of attacks through DMA can not work "as is" on hardware-assisted virtualization due to hardware protection i.e. IOMMU[4]. But, new attack vectors [Mur08] have been found in the specialized hardware instructions dedicated to virtualization. For example, an instruction on Intel processors permits memory remapping. Through this one, it is possible to migrate a part of the memory to an area where the attacker can read and write on. In consequence, it is possible to use the same attack previously introduced based on DMA driver just by changing the vector allowing memory modification and reading. An example of this attack has been implemented based on a bug in DQ35JO motherboard BIOS. The attack exploits the Intel chipset's memory remapping feature and allows to circumvent some CPU or chipset memory protection mechanisms. That includes the possibility to gain full access to a specially protected region of system memory, called SMRAM (System-Management RAM) memory.

With the growing complexity of hypervisors, it is harder to validate the code and this can bring new attack vectors. For example, if a buffer overflow in the hypervisor can be exploited, it can allow to execute code with hypervisor privileges.

[4]An IOMMU (Input/Output Memory Management Unit) is a memory management unit (MMU). The interesting thing for virtualization security is that some IOMMU provide memory protection from misbehaving devices, preventing a device to read or write to memory that it should not access (i.e. that has not mapped for it).

2.2 From the hypervisor to VM Attacks

The purpose of this class of attacks is to disrupt, spy, steal or modify VM's data with malicious code running at the hypervisor level. This class of attacks is based on injecting code in the hypervisor. Code injection can be done:

- using techniques presented in the previous section;
- by a malware running at the same level than the hypervisor.

This class of attacks are mainly known from the infamous Blue Pill attack [Rut08]. We divide this section into three subsections:

- hardware accelerated rootkits (HAR);
- high privileged virtual machines;
- hardware attacks.

2.2.1 HAR or shadow virtualization

This kind of attacks [ESZ08] can virtualize the running operating system or the running hypervisor at run-time without being detected by it. It thus permits to have a shadow hypervisor (i.e. not visible from the user viewpoint) that can spy and control the whole machine. This type of rogue hypervisors are known as hardware accelerated rootkits (HAR).

However, this kind of attacks can be done in multiple ways:

1. Build a new rogue hypervisor and make it supersede the existing OS/hypervisor. The following steps describe how a HAR can take control of a machine (figure 8):

   (1) the HAR starts to run on the OS by using any well known attack vector;

   (2) the HAR moves its memory range at the beginning of kernel space and remaps the kernel's one;

   (3) the HAR migrates the operating system in a virtual machine without rebooting it. Any operation of the newly virtualized OS/hypervisor is now controlled by the HAR.

2. Statically modify a legal hypervisor in order to make it do malicious things (e.g. modify an open source hypervisor (i.e. Xen) to spy keystrokes). This second form of the attack may be harder to detect.

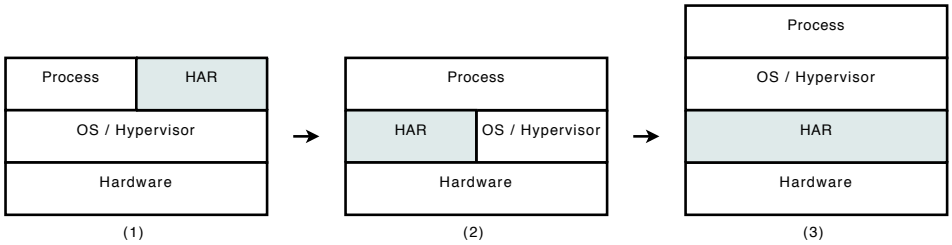


Figure 8: HAR taking control of an operating system

3. Dynamically modify a legal hypervisor at run-time. This can be done with a DMA access (or other vectors allowing memory modification) to replace a hypervisor's module by a malicious one. This is the hardest to detect form of the attack. At the operating system's viewpoint, nothing in the memory or in the CPU registry has changed during and after the migration (i.e. the virtualization of the previous OS/hypervisor). This is however the most difficult attack to achieve.

It is theoretically possible to detect all those attacks [RKK07] using latency measures. Indeed, with the addition of a virtualization layer, the running time of any system operation suddenly takes more time to achieve, which is possible to observe. But the latency due to malicious code is often negligible compared to the latency due to the legal hypervisor. Moreover all the virtualization issues can be attributed to the legal hypervisor that has been modified, which makes those attacks hardly detectable from the VM viewpoint.

This attack has been demonstrated several times. Operating systems virtualization at run-time has already been done [Rut08] as well as modifying a legal hypervisor or installing a rogue hypervisor at run-time. When this kind of attacks succeed, the HAR gains full access to all the hardware and VM, i.e. the entire machine.

[KC06] implemented shadow virtualization for GNU/Linux and Microsoft Windows XP operating systems. Their approach, 'SubVirt' as described in the figure 9, has the purpose to virtualize an operating system without being noticeable for the end-user. Moreover, their approach allows to run an underlying rogue operating system as a hypervisor that hosts malicious services. The rogue operating system has been implemented on top of Microsoft Windows XP with Virtual PC and on top of Linux using VMWare. Installing such HAR takes only 24 seconds and 226Mb of disk space. After the installation of the shadow virtualization, the operating system takes only 19 seconds more to boot. Using their proposal, [KC06] implemented:

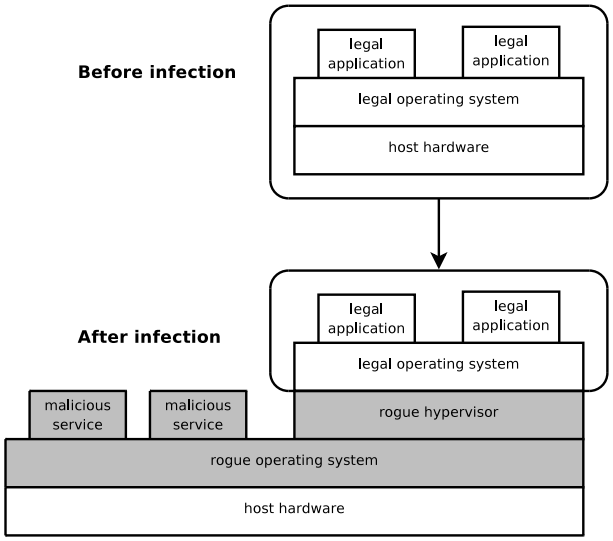


Figure 9: SubVirt taking control of an operating system

- hidden rogue services: spam relays, botnet zombies or phishing web servers.
- services that observe data and events coming from the legal OS. Those services can stealthy log keystrokes and network packets.
- malicious services that trap the execution of the legal OS (or applications in it) at arbitrary instructions. One of those one traps every call to the "write" method of the SSL library socket that permits to read the clear-text data before it is encrypted.
- services that deliberately modify the execution of the target system. The services proposed permit to modify network communication, delete e-mail messages or change the execution of an application in the legal OS.

Another method that leads to shadow virtualization aims at modifying a legal hypervisor in run-time. Unlike previous ones where the hypervisor's code is statically modified, in this case, it is modified on-the-fly. This can be done with a DMA access (or other vectors allowing memory modification). It replaces a hypervisor's module by a malicious one. This attack is hardly detectable from the VM viewpoint because:

- the latency due to malicious code is negligible compared to the latency due to the legal hypervisor;

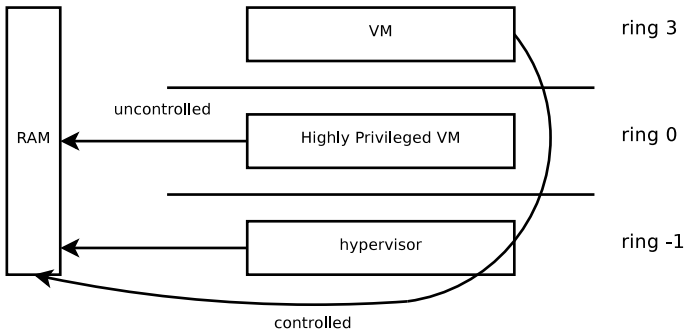


Figure 10: Memory access from a highly privileged VM and from a normal one

- all the virtualization issues can be attributed to the legal hypervisor that has been modified.

2.2.2 High privileged Virtual Machine

The purpose of this type of attacks is to disrupt a VM using a high privileged VM. Some hypervisors (like Xen) have the possibility to run VM at different levels of privileges in terms of allowed interactions with VM and hardware. This feature has been introduced to allow some VM to do advanced tasks for others VM like handling input and output of hardware devices. It permits to reduce the amount of trusted code. Under Xen, they are two levels of privileges:

- The *normal privileged* VM which is called *domU*. It accesses the hardware through the high privileged VM.
- The *high privileged* VM which is called *dom0*. It has the possibility to directly access the hardware. In consequence, it has the ability to read and/or write on the memory of others VM.

This architecture allows a special VM to act for example as an anti-virus appliance. In practice, the anti-virus running in the high privileged VM accesses other VM to analyze them (i.e. inspect their memory and files). As shown in the figure 10, a normal VM is running at the user level (ring 3), a high privileged one at the system level (ring 0) and the hypervisor at a higher privileged level (ring -1).

When the normal VM tries to interact with the RAM, it has to past through the hypervisor that will allow or deny it. When a high privileged VM tries to interact with the RAM, it has direct access to it and through it is not controlled by the hypervisor.

It is possible to use this scheme to create a rogue VM with high privileges, that could monitor/spy other VM. Using this vector, it is possible to attack the integrity, confidentiality and availability of any others VM and takes the control of them.

Moreover, the attack is not noticeable for a normal VM as the changes take place outside of it. Indeed, the rogue VM is just another VM and does not modify the current hypervisor. At least, the hypervisor can detect such attacks by looking for a high privileged VM that inspects other VM. But, even this can be bypassed if the rogue VM is registered as a legal security component.

2.2.3 Attack on Virtualized Hardware

The main purpose of virtualization technology is to run multiple operating systems on a single physical computer. As multiple OS use the same hardware, it makes possible to cause information flows between VM with a sharing hypervisor. All shared hardware devices can be a vector of attacks:

- vHDD : virtual hard disk drive;
- vCPU : virtual CPU;
- vRAM : both RAM and video memory;
- vNet : virtual network card.

Each attacks allowed to corrupt the integrity, confidentiality and/or availability of a VM from the hypervisor.

vHDD (Virtual Disk Storage)

A vHDD can be stored in :

- a file on a disk;
- a partition of a disk;
- a physical disk;
- a storage area network disk;

For the first three types of vHDD, it is possible for a hypervisor to modify, read or delete the vHDD. In terms of security properties, this means that the hypervisor can attack the integrity, confidentiality and availability of a vHDD. Indeed, a vHDD is a physical resource for the hypervisor. The hypervisor can access the vHDD as any others hardware resources. It has full control on the vHDD as it runs at the highest level of privileges. Using other vectors like memory and network corruption, it is even possible for a hypervisor to tamper SAN disks (see Attacks on

Virtual Network). All those attacks are due to the fact that the hypervisor has full control on the hardware and VM. In consequence, the only way to prevent those attacks is to trust the hypervisor or monitor it using a trusted tier.

An example of that attack is the modification of a VM's kernel. In this case, only the integrity and availability are important, the confidentiality of a kernel is not an issue. Of course, it is possible to encrypt file to protect integrity and confidentiality but the hypervisor can access the encrypted key and decrypt all the data (unless the key is stored in vTPM [BCG+06]). Therefore, the hypervisor can modify the kernel directly on the disk. An integrity check detects such modifications but it is possible to implement an attack known as time-of-check-to-time-of-use [BD96] quicker to execute and modify thnings in order to not being detected, than the time needed to detect it. Moreover, static verification forbids the dynamic loading of module [MMH08] unless the hypervisor is modified to verify the integrity of each loaded modules.

vRAM (Virtual Memory)

Both the confidentiality and integrity of physical RAM can be attack from the hypervisor viewpoint because it has full access i.e. read and write to the whole memory. Moreover, it is also possible for a hypervisor (and a high privileged VM) to access memory through DMA access. Finally, hypervisor can use System Management Mode to corrupt or read the memory of video cards.

An example of attack on memory is the modification of VM's kernel on-the-fly. Contrary to the previous attack (on disk), this one modify the kernel when it is loaded on the memory. The integrity check on the disk will thus be successful even if the kernel's binary memory segment is modified.

vCPU (Virtual Processor)

A vCPU is a virtual processor that contains, like a normal CPU, registries and flags. Those ones can be modify or read through special hardware instructions by a privileged process like a hypervisor. For example, an attacker can read the vCPU registries during an integrity verification and modify them to return a successful answer whatever the file contains. Potentially, it can allow to execute code in any VM.

vNet (Virtual Network)

Within most of hypervisor implementations, the network cards provided to virtual machines are software bridges which are connected to a physical network interface. A rogue hypervisor could plug a sniffer on it and so, could attack the confidentiality of the virtual network. Also, the software bridge can be modified by the hypervisor to corrupt data contained in the network packets and through attack the integrity of the vNet of any VM. For example, when a SAN disk is used, the vNet can be modified to substitute the kernel, when it is loaded, to a rogue one by modifying the content of network packets on-the-fly.

All these attacks are due to the fact that the hypervisor is at a higher privileged level than the VM on the physical hardware. Unless the hypervisor is controlled by another process (a TPM or a hypervisor of hypervisors) that should act as a trusted tier, those attacks remain possible.

2.3 From VM to other VM attacks

The purpose of this type of attack is to disrupt a VM from another VM. It targets the main security properties of virtualization: containment. Two vectors can be found in the literature, attacks:

- on resources i.e. modifying the hardware resources of other VM;
- using backdoors.

2.3.1 Attacks on resources

The class of VM to VM attacks through tampering resources can be done in many ways. Almost all shared hardware devices between two VM can be used to. This class of attacks uses the same method than DMA attacks (see section 2.1.1). Based on DMA, a VM can access the memory of any others VM and read or write in it.

2.3.2 Backdoors

This class of attacks allows a VM to disrupt another VM. A backdoor is a special trap in a computer program that permits to enter a system without being noticeable. By default, a backdoor is deactivated but when some special events are sent, it opens a hidden door in the system. In consequence, it requires that the hypervisor has been previously modified in order to react to some special events. For example, the hypervisor shifts to a special mode that allows a VM to spy on other one (or increases VM privileges) when a specially crafted packet is received on the network interface. This packet raises an interruption that launches the backdoor. Such method has been previously implemented and can be used to remotely attack the integrity, confidentiality and availability of VM. For example, a backdoor is implemented as an hypervisor module. This module is not activated by default. It is activated when receiving a special network packet on the loopback. When activated, the backdoor modifies the "authorized_keys" file [5] of the root account. Moreover, the backdoor modifies the SSH configuration file to allow root login.

[5] The "authorized_keys" file contains all the keys that are allowed to be used to connect to the account

Using the backdoor, it become possible to add its ssh key to any root account on any VM running on the hypervisor that implements the backdoor.

2.4 Conclusion

In this section, we have presented all the different attacks specialized for virtualization technology: attacks from VM to hypervisors, between VM themselves or from hypervisors to VM. All of these attacks are able to break the three main security properties: integrity, confidentiality and availability. Having a proper implementation of hypervisor and security components dedicated to virtualization security, is essential. This is the focus of the next section: what are the actual security measures proposed by virtualization researchers/vendors to address the security issues presented above.

3 Hypervisor Security

In the previous section, we have presented the different vectors to subvert virtualization technologies. In this section, we present the different security mechanisms to increase the security of the hypervisor and the other security components that are deployed into the hypervisor or act as modules for it.

First, we will present the protection of the hypervisor itself. We will then survey the different methods to guarantee the integrity of the VM. We will then discuss the isolation of resources between VM and finally, we will present the access control mechanisms between VM.

3.1 Protecting the Hypervisor Layer

3.1.1 Type II Hypervisor

When the hypervisor is a process of an operating system (e.g. VMWare Workstation), the security is related to the whole OS. The protection mechanisms available are the same as on a classical operating system like anti-virus, anti-malware, etc.

The drawback is that it is possible to use any vectors present in the OS to subvert the hypervisor. For example, any rootkit will have at least the same (or more) privileges than the hypervisor and can subvert it by inspecting its memory or even modify the hypervisor virtual drivers.

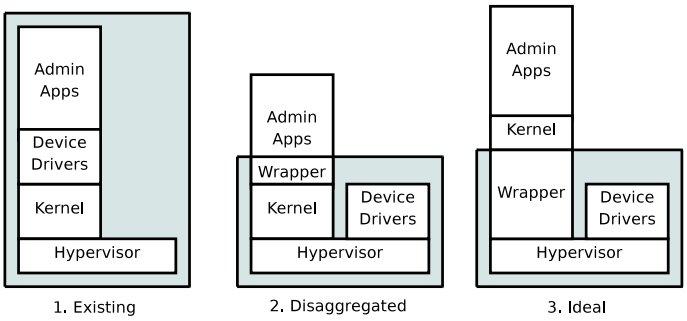


Figure 11: The three models of virtualization TCB disaggregation

3.1.2 Type I Hypervisor

When the hypervisor is directly running on the hardware (e.g. Xen), the principal challenge is that it needs to be small in terms of lines of codes, in order to be verified and validated as safe and trusted. In this case, the hypervisor does not includes any exploits. But, the growing number of lines in hypervisor code brings the unability to verify the whole code. In consequence, others solutions for hypervisor security have been introduced.

Classical Security Components

The first approach is to use security components previously used at operating system level to secure the hypervisor. For example, anti-virus or anti-malware can be used to secure the hypervisor.

The drawback is that a hypervisor is not an OS, in consequence, the same methods can not be used. Moreover, the classical security components like antivirus have shown their limits in the latest years. Including antivirus or other malware protections into the hypervisor to control it, can be done but is not very efficient. Also, it includes more codes at the hypervisor level and as explained before, this leads to code verification issues.

Reducing the TCB through Disagregation

As explained before, the less code the hypervisor is composed of, the easier it is to verify. For example, in a classical operating system, it is required to trust at least the kernel and often the administrative applications. The goal of disaggregation is to reduce the size of the trusted computing based (TCB). These components are then put into less privileged containers. For example, with Xen, the TCB [MMH08] contains the hypervisor itself and the privileged VM that manages IO and administrative tasks (i.e. dom0). The privileged VM is a complete Linux OS and needs to be trusted.

component	C	ASM
Hypervisor	98	3
Dom0 kernel	1500	9.6
Dom0 drivers	≤ 2400	≤ 2.6
Total number of lines	≤ 4000	≤ 15.2

Table 1: Size of a Xen TCB (in thousands of code lines).

It is possible to reduce the TCB to what is essential: IO managers and administrative tools. Three disaggregation schemes exist (cf. figure 11):

1. The first scheme (on the left) shows the existing Xen architecture where the TCB (gray rectangle) includes a whole operating system (i.e. Linux).

2. The second one (centered) describes a better situation [MMH08], where only the Linux kernel and needed device drivers to be trusted and not a complete Linux stack.

3. The third scheme (on the right) presents the best case scenario for Xen disaggregation with a minimal TCB: only the device drivers need to be trusted, in addition to the hypervisor and the wrapper. The wrapper acts as an API between hypervisor and its management tools. The wrapper permits to verify inputs and ouputs given to the management tools. The wrapper also allows to avoid to put the management tools in the TCB.

[MMH08] have implemented the second scheme presented in the figure 11. They started with an existing Xen TCB which code size (cf. table 1) was 4000,000 lines of C code and 15,200 lines of ASM code.

They manage to strongly reduce the size o the TCB, as described in the table 2, where rows beginning with $+$ indicate where code was added to the TCB; rows beginning with $-$ indicate where it has been removed. 920,000 lines of C code were removed and only 9,200 lines were added in order to support the disaggregation scheme (-23%). They also reduced the amount of ASM code (500 lines) of the TCB (-96%). In addition, they also reduced the size of Python code (160,000 lines), not in the TCB but used in some services. As this middle disaggregation scheme is not the best case scenario, there still remains a large amount of code in the TCB. Especially, the number of code lines linked to the kernel and VM management drivers. With ideal disaggregation, those two components could be heavily reduced to few thousands lines. That should trend to build hypervisors as micro-kernel architectures [HUL06].

+/−	component	C	ASM	Python
+	DomB	9.2	0.5	−
−	libc	690	15	−
−	Python	220	−	140
−	libxc	9.9	−	−
−	xend	2.4	−	17
+	Added	9.2	0.5	−
−	Removed	920	15	160
	Total number of lines	$\simeq 3080$	$\simeq 0.7$	-160

Table 2: Migration to a disaggregated Xen TCB: code size (in thousands lines).

The drawback is that reducing the TCB requires multiple changes in the hypervisor code. Thus, knowledges about the hypervisor code, hypervisor architecture and related coding language are essential to be able to reach the task of reducing the TCB.

Moreover, a totally disaggregated TCB, as any other micro-kernel architecture, increases the number of transitions between privileged and unprivileged domains. This kind of transitions has is very costly in terms of processor interruptions. Also, it brings many inter-process communications to permit data exchange between each component of the hypervisor. This large number of communications increases the overall system latency.

Integrity Verification

Another approach to guarantee the protection of the hypervisor is to verify its integrity. For example, a footprint of the hypervisor is done when it is installed. At each boot, before loading the hypervisor, a verification of the hypervisor is computed against the footprint to verify that a modification has not been done on it. Two different methods have been used to do the integrity check process. The first method (software) integrates it inside the BIOS. This process can be done by special BIOS like Sun OpenBoot [Inc97] and LinuxBIOS [MHW00]. The second one (hardware) integrates specialized hardware instructions as Intel TXT (Trusted Execution Technology) [UNR+05] to verify the integrity of the hypervisor [WR09]. This process uses another hardware component, the TPM to store the footprint[6].

The drawback is that the hardware approach is more robust to attacks but requires additional hardware equipments. The software approach can be corrupted by a BIOS malware. Invisible Things Labs [WR09] demonstrated attacks on TXT using the SMM. Intel is currently preparing STM (SMM Trasnfert Monitor) in or-

[6]There also exists an equivalent technology at AMD, which is called Presidio.

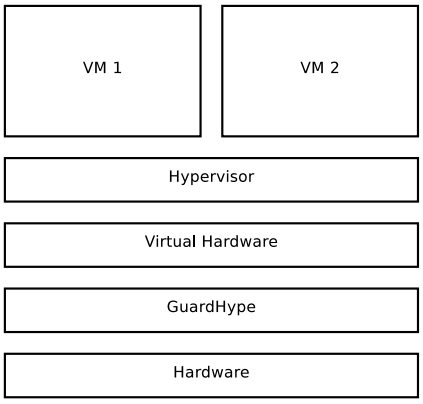


Figure 12: GuardHype

der to solve the issue, but it is probable that actual Intel processors all suffers from this flaw.

Micro-Kernel

Another approach of hypervisor protection is to use byzantine fault tolerant systems [ACDA09] like micro-kernel to create a hypervisor that healths itself from attacks. Hypervisor designed as micro-kernels [HUL06] already exist. Extending them to support byzantine fault can be done but remains a theoretical concept for the moment.

The drawback is that, as disaggregation, the main micro-kernel's drawback is a sensible increase of transitions between unprivileged and privileged contexts in the processor, and many communications between components, leading to more latency.

Protecting Hypervisor Layer

Another approach to protect the hypervisor is to protect the hypervisor layer itself. It is useful to protect the hypervisor layer because it is where the hardware accelerated rootkits are plugged (see section 2.2.1). GuardHype [CZL08] introduced such protection. GuardHype is a hypervisor for hypervisors. It allows only well known and verified hypervisors to run. Like hardware accelerated rootkits and as shown on figure 12, it creates a complete virtualized hardware layer and allows a legal hypervisor to run on it.

The drawback of this approach is that it is not protected against bugs in legitimate hypervisors. For example, if a legitimate hypervisor like Xen includes a buffer overflow that allows a VM to execute code as a hypervisor, it will not stop it. HAR works by duplicating the state of the current OS into a VM. This behavior is

detected by GuardHype because it does not allow an unknown hypervisor to load. The approach supposes that the boot process is trusted, it can be done using TPM[7].

We have presented how can actual hypervisors be protected. The technology evolves fastly and some new protections may appear, such as STM for the protection of the Intel TXT and AMD Presidio. In the next subsection, we will present different approaches allowing the hypervisor to verify the integrity of virtual machines.

3.2 Integrity of VM

There exist different ways of verifying the integrity of VM from the hypervisor. The verification process can be done using software or hardware components and ables the hypervisor to load only sane VM into memory.

3.2.1 Software Integrity Verification for VM

Software integrity verification can be done using well-known algorithms like SHA. Few hypervisors [QT06] include verification of critical VM components like kernel and critical applications when a VM is started. It allows to detect any illegal modifications of them. Moreover, it can be used at run-time, by inspecting the memory containing the component. It can do the same by hooking system calls performing modifications on filesystems and verify that none is corrupting the integrity of the VM or its components.

The drawback is that if the hypervisor is corrupted, the footprints can be changed and the whole integrity process bypassed.

3.2.2 Hardware Integrity Verification for VM (vTPM)

This approach uses at least a TPM component, which can be enhanced by other hardware instructions like Intel TXT (cf. section 3.1). Some hypervisors e.g. Xen/sHype [SVJ+05] are able to virtualize the TPM [BCG+06] into multiple instances where each one is assigned to a VM. This permits to use TPM when multiple operating systems are running concurrently on the same physical hardware, whereas a normal TPM can only supports one OS. The TPM can be used to store the

[7]Trusted Platform Module (TPM) is an emerging security building block offering system-wide hardware roots of trust that the system softwares can not compromise.

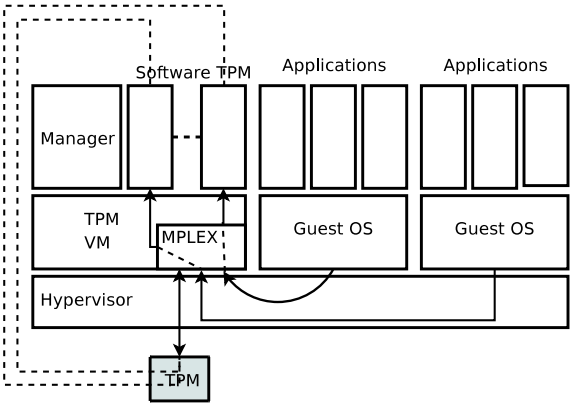


Figure 13: sHype: a Virtualized TPM

footprint of the OS's kernel or other software of the VM to verify their integrity. As shown in figure 13, the VTPM uses multiple components:

- a hardware TPM;
- a TPM virtual machine;
- a multiplexer (MPLEX);
- a vTPM manager;
- multiple software TPM.

The TPM VM interacts with the TPM through the hypervisor. The use of a multiplexer ables the capability to have multiple software TPM controlled. Each software TPM is linked to a a virtual machine through the multiplexer. Each VM has a fully fonctionnal TPM that is separated from other software TPM. The isolation between vTPM is made at the TPM VM level.

The drawback is that, contrarily to the software implementation, the hardware approach can be (almost[8]) fully trusted but it required additional hardware devices.

3.3 Resources Isolation between VM

Until the last few years, the isolation of resources between VM was done using only softwares. As explained in the section 2, many attacks vectors target the resources

[8]Except the attacks to TPM, e.g. TXT [WR09].

isolations. But, new hardware instructions and devices permit a better isolation at the resources level. Those instructions are primarily known as IOMMU[9] and allow an isolation at the hardware level.

3.3.1 Without hardware protections

Disk

A VM can only read its disk partitions but a hypervisor can read all the disk partitions. A rogue administrator at the hypervisor level can read, write and modify data on disk partition of the VM. To be protected against such attacks, the VM needs to use full disk encryption. But it is not enough, the keys must be stored using a sealing functionality of a safe place like a TPM [BCG+06].

The drawback is that encryption allows to ensure the integrity and confidentiality of the disk partitions related to VM but is not protected against a destructive attacks i.e. deleting the disk partitions. In addition, as keys are store in memory at all time, they may be read, especially using an attack against SMM, which can access the whole system memory [WR09].

Network

All the network traffic that passes through the virtual network can be read and modified by the hypervisor. To protect network against rogue administrators, it is possible to use encryption scheme such as TLS or IPSEC [MTS+06]. It guarantees the confidentiality of data. But by modifying the software bridge i.e. the virtual network card, it is possible for the hypervisor to attack the integrity of a packet. To protect confidentiality and integrity of the network, a virtual private networking (VPN) software [LVBP05] in the VM is sufficient.

The drawback is that using in-deep memory inspection, it is possible to read all the keys certificates and VPN related information. Based on those data, a rogue administrator can connect to the VPN and spy it.

3.3.2 With hardware protections

Hardware protections (IOMMU) are known as VT-d on Intel processors [Hir07]. If a hypervisor is designed to support IOMMU [BY06], it can address the problem of most of the hardware backdoors. IOMMU permits to move DMA devices drivers into a separate unprivileged driver domain. By doing so, each PCI device can be

[9] IOMMU (Input/Output Memory Management Unit) are security oriented instructions aiming at prevent drivers to access to memory range where they should not (see subsection 3.3.2 for hardware protection using IOMMU).

limited to DMA only to the memory region occupied by its own driver and not the whole memory.

The network card's micro-controller can still be used to compromise a network card driver but not the whole memory. Assuming that only encrypted communications are passing through the network, there is not much an attack can gain by compromising the network card driver, besides a deny of service. Similarly for the disk drivers, if a full disk encryption is used, an attacker will not be able to retrieve any information.

The drawback is that IOMMU are limited to DMA devices and are not applied to CPU and memory controllers that must be trusted also.

3.4 Access Control between VM

Traditional hypervisors do not include any mechanisms to control the access of resources from a VM to another one. With the growing number of VM and the multiple interactions between them in virtualized networks, it becomes essential to have a mean to regulate them. In a first time, complex access policies manually written have been proposed. They lied on softwares like firewalls.

Nowadays, an access control mechanism is included in some hypervisors [SVJ+05] : mandatory access control (MAC). MAC enforcement, applied to virtualization, controls inter-VM communications, whether they are on a single physical machine or across machines. Such MAC enforcements are both stronger than traditional VM isolation, as even network communications are controlled. MAC is also more flexible than traditional VM isolation, as local VM interactions are now enabled. Such MAC inter-VM mechanisms promise comprehensive control of system information flows [SJV+05b]. But they can be subverted by covert channels[10].

sHype [SJV+05b] adds authorization hooks to Xen's (overt[11]) communication mechanisms to authorize inter-VM communications on the same physical computer. It uses a Type Enforcement [BSS+96] (TE) model to describe the inter-VM communications authorized in a system. TE is expressive enough to enforce security policies such as isolation. As describe in figure 14, sHype adds two components:

- an Access Control Manager (ACM) that allows to controls the access on resources shared between VM;

[10] In information theory, a covert channel is a parasitic communications channel that draws bandwidth from another channel in order to transmit information without the authorization or knowledge of the latter channel's designer, owner, or operator.

[11] Overt channel is any communications path for the authorized data transmission within a computer system or network.

Status	Name	Label (type)
Subject	$RedVM_1$	red_t
Subject	$RedVM_2$	red_t
Subject	$BlueVM_1$	blue_t
Subject	$BlueVM_2$	blue_t
Subject	$GreenVM_1$	green_t
Object	$RedNetC_2$	rednet_t
Object	$BlueNetC_1$	bluenet_t
Object	$RedHDD_1$	reddisk_t
Object	$BlueHDD_1$	bluedisk_t
Object	$GreenHDD_1$	greendisk_t

Table 3: Example of sHype MAC labelling.

```
# xm addlabel red_t dom RedVM1.xm
# xm addlabel reddisk_t res file:///dev/sda
```

Listing 1: "Labeling command for VM and resources"

- a Policy Manager that stores the security policy describing allowed interactions between VM.

3.4.1 Access control on a single physical machine

As an example of what sHype is actually able to guarantee, let consider five VM: $RedVM_1$, $RedVM_2$, $BlueVM_1$, $BlueVM_2$, and $GreenVM_1$ and the following resources: two network cards $RedNetC_1$ and $BlueNetC_1$, and three hard drives $RedHDD_1$, $BlueHDD_1$, $GreenHDD_1$.

The first operation is to give a label to each VM and each resource. The table 3 describes that labelling. The $RedVM_1$ and $RedVM_2$ VM have the label red_t. $BlueVM_1$ and $BlueVM_2$ are labelled with the blue_t and $GreenVM_1$ with green_t. The network resource $RedNetC_1$ has the label rednet_t and $BlueNetC_1$ has the label bluenet_t. $RedHDD_1$, $BlueHDD_1$ and $GreenHDD_1$ have respectively the labels reddisk_t, bluedisk_t, greendisk_t.

The labels are set on a VM or a resource using "xm" command interface (i.e. the command allowing the administration and management of VM and resources under sHype). For example, the first command in the listing 1 permits to add the label red_t to the $RedVM_1$ virtual machine and the second command permits to add the label reddisk_t to the $RedHDD_1$.

Subject (type)	Object (type)	Access(Action)
red_t	reddisk_t	* (r,w,x,a,d,m,...)
blue_t	bluedisk_t	*
green_t	greendisk_t	*
red_t	rednet_t	*
blue_t	bluenet_t	*
green_t	rednet_t	*
green_t	bluenet_t	*

Table 4: Example of sHype MAC policy.

Using those labels, sHype has the ability to apply any policy that classical MAC mechanism can enforce.

Let consider a concrete sample policy, summarized in table 4. That policy restricts the access to reddisk_t to only red_t virtual machines and apply the same principle between, respectively blue_t and green_t VM to bluedisk_t and greendisk_t disk resources. Moreover, $RedVM_1$ and $RedVM_2$ can only use the $RedNetC_1$ network cards. $BlueVM_1$ and $BlueVM_2$ can only use the $BlueNetC_1$ network card. $GreenVM_1$ can use both network card.

sHype uses XML files to describe policies. The sHype XML configuration file describing the policy given above (in table 4) is presented in the listing 2.

3.4.2 Access control on multiple physical machines

sHype is not currently able to control the access of VM running on different physical machines, but it soon will. Moreover, the access control XML DTD already allows to express restrictions related to physical machines. For example, let consider another type of distributed access control policy that permits to avoid to run two concurrent VM on the same physical computer. This type of restriction is called Chinese Wall. A possible example, applied to the previous policy should be to prevent running green_t VM on the same physical computer than red_t ones.

To apply this policy, sHype will permit to add the set of rules and permissions described in the listing 3 at the end of the previous rules (listing 2).

The drawback is that currently, sHype does not support access control between VM located on different physical machines. Moreover, sHype does not protect

```
<?xml version="1.0" encoding="UTF-8"?>
<MAC_Rules id=1>
<Source_Type>red_t</Source_Type>
<Destination_Type>reddisk_t<Destination_Type>
<Action>*</Action>
</MAC_Rules>
<MAC_Rules id=2>
<Source_Type>blue_t</Source_Type>
<Destination_Type>bluedisk_t<Destination_Type>
<Action>*</Action>
</MAC_Rules>
<MAC_Rules id=3>
<Source_Type>green_t</Source_Type>
<Destination_Type>greendisk_t<Destination_Type>
<Action>*</Action>
</MAC_Rules>
<MAC_Rules id=4>
<Source_Type>red_t</Source_Type>
<Destination_Type>rednet_t<Destination_Type>
<Action>*</Action>
</MAC_Rules>
<MAC_Rules id=5>
<Source_Type>blue_t</Source_Type>
<Destination_Type>bluenet_t<Destination_Type>
<Action>*</Action>
</MAC_Rules>
<MAC_Rules id=6>
<Source_Type>green_t</Source_Type>
<Destination_Type>rednet_t<Destination_Type>
<Action>*</Action>
</MAC_Rules>
<MAC_Rules id=7>
<Source_Type>green_t</Source_Type>
<Destination_Type>bluenet_t<Destination_Type>
<Action>*</Action>
</MAC_Rules>
```

Listing 2: "MAC Policy for VM and resources"

against covert channels. Another drawback is linked to Mandatory Access Control. MAC requires to write an access control policy. The writing process of the policy requires an overall knowledge of the virtualized architecture and the underlying virtualization components. And as with any MAC mechanism, writing fine-grained security policy takes much time.

4 Virtual Machine Security

VM security can be divided into two approaches:

- executing an existing security system e.g. malware detection systems into

```
...
<ChineseWall id=1>
<ConflictSet>
<Conflict>
<Type>red_t</Type>
<Type>green_t</Type>
</Conflict>
</ConflictSet>
```

Listing 3: "sHype MAC Policy for Chinese Wall enforcing"

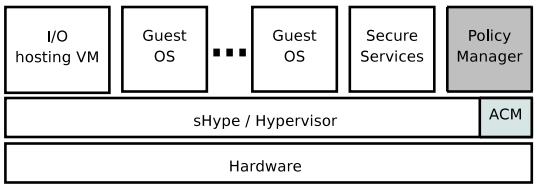


Figure 14: sHype Architecture

the VM i.e. running it as an application in the virtualized OS.

- executing a security system outside of the VM.

4.1 Classic Security components

Using the existing security components of classical un-virtualized OS into virtualized ones has the advantages to not need any modification in the code of virtualization architectures and can work at the virtualized system level.

The drawback is that it can not use the virtualization isolation because the security components (e.g. anti-virus and anti-malware) run directly on top of the operating system. Thus, the security components are inside the VM and are exposed as any components in the OS. In consequence, the security components can be attacked and disabled like any security mechanisms (e.g. anti-virus) on a normal un-virtualized OS.

4.2 Auditing and Monitoring

This approach resolves the issue of attacks on security systems (described in subsection 4.1) because it provides security components isolated from the supervised operating system. Moreover, the attacker is not even able to see the security components.

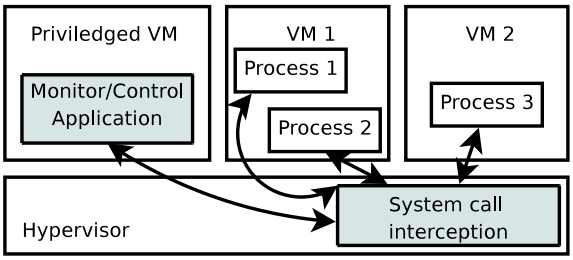


Figure 15: Syscall interception from outside a VM

The drawback is that "outside" security components can only see low-level information. They can only work with memory inspections and those information are hard to use for security purpose. But, this limitation can be overcome by extracting high-level information from these data as explained in [GR03]. Another advantage of this approach is that it does not have to run once per VM but once for all VM and by doing so reduces the resources consumption.

To harvest information about the state of an operating system and its softwares, current security applications often use system calls monitoring or other logs information. It gives high level information (from kernel to applications logs). This approach brings good results but has a major flaw: it needs to trust the kernel and the security applications. If one of those is compromised, the whole auditing and monitoring process can be corrupted. In practice, it means that security components can not be trusted as soon as a critical step of an attack has been successfully done.

By putting out the monitoring and auditing systems from the OS, the issue is solved. The attacker can no more see the security applications. As shown on the figure 15, using memory inspections, the monitoring and auditing systems, when put into a special privileged VM, are able to intercept and control the executions invoked in the VM and control their execution [OOY08] (like any other security systems within a VM). The interception and control is done at the hypervisor level where each execution step of the VM is checked. While verifying the system call, the hypervisor puts the intercepted process in hold until the hypervisor allows or denies the system call.

Moreover, the security applications can also run at the hypervisor level. Running at the hypervisor level is always dangerous because a flaw in the security application can lead to the corruption of the hypervisor. Running it in a special privileged VM that can be trusted is a better solution (cf. figure 15). It is close to micro-kernel architecture [HUL06] with privileged VM acting as services. It allows

```
1 traceChild: yes
2 open
3   fileEq(1,"/etc/passwd")
4   or fileEq(1,"/etc/shadow")
5   or fileEq(1,"/etc/crontab")
6 exec
7   fileEq(1,"*")
```

Listing 4: "Policy of the monitoring component"

```
1 traceChild: yes
2 exec
3   default: allow
4   fileEq(1,"/home/*/scan_ssh")
5      deny(-1)
```

Listing 5: "Policy of the control component"

better separation of tasks and so a higher isolation in case of corruption.

For example, as proposed in [OOY08], it is possible to use a policy to enable auditing of system call of the VM by the hypervisor. An auditing policy that monitors the "open" syscall when accessing sensible data (`passwd`, `shadow`, `crontab`) and the "`exec`" syscall are described in the listing 4.

The traceChild keyword is used to monitor every syscall launched by the child created by an "`exec`" syscall.

[OOY08] proposed to extend the auditing properties to be able to control syscall from outside the VM. To implement it, they extend the language allowing to monitor syscalls. For example, if one want to deny the launching of "`/home/*/scan_ssh`" application at the syscall level, it is possible to write the statement shown on the listing 5.

Those applications allowing monitoring and control increase the execution time of each syscalls. Nonetheless, the overhead is small enough to allow this architecture to be used in real world applications. As shown on table 5, the overhead is at an average of 20% with only monitoring and 50% when controlling. But this overhead decreases when the same sequence of syscall is made multiple times. For example, in the table 5, the overhead almost disappears when using monitoring facility and the number of requests (against an Apache server running in the controlled VM with a Xen hypervisor) increases.

This approach of auditing an operating system from the outside can lead to new approaches in security virtualization. Starting from the fact that a VM OS can be corrupted and that an OS is almost impossible to be completely verified, running critical applications on it, can lead to disrupt them. [CGL+08] showed that con-

Number of requests	1	2	32	256	1024
Xen	1.02	0.94	0.87	0.84	0.85
Xen+Monitor	1.23	0.94	0.88	0.84	0.85
Xen+Control	1.57	1.54	1.50	1.49	1.50

Table 5: Request execution time (in seconds) against an Apache server.

trolling system calls from the outside of the VM can be used to protect applications from disrupt OS by making the critical application directly interacts with the hypervisor. In this case, the hypervisor creates a double isolation barrier (cf. figure 16). The first one is located between the VM and the hypervisor through the usage of virtual hardware. The second barrier is located between the critical applications and the virtualized operating system itself. The second barrier is based on cryptographically protected system calls between the application and the hypervisor to avoid integrity or confidentiality corruption at the virtualized kernel level. Using this solution, the virtualized operating system can not control the critical application. Moreover, the virtualized OS can not inspect the memory of the critical application because it is protected through cryptograhical algorithms. In consequence, even if the virtualized operating system is corrupted, it does not disturb the running of the protected critical software. This can not be done without virtualization and system calls control from outside of the VM.

The drawback is that the API allowing Auditing and Monitoring process can be used to create specialized malware. Moreover, those API can contain flaws (e.g. buffer overflow) that permit to corrupt the hypervisor. And, thus, it should permit to execute code at the hypervisor level.

4.3 Intrusion Detection System

As explained in the beginning of this section, security applications running directly on the OS have a major flaw: if the OS kernel is corrupted, the security applications will be too. Using virtualization, security applications can be put outside of the VM. To leverage this new architecture capability, special intrusion detection system (IDS) have been developed [GR03, KC05]. First, normal IDS have been put outside the VM to monitor it using memory inspections or other means as seen in section 4.2. Recently, new interfaces have been developped to plug IDS outside of a VM. These interfaces are implemented inside the hypervisor. This brings the capability to have an IDS that is not resilient in the OS (as network IDS for example). Moreover, the IDS has the same level of information of what is happening on a OS as a host IDS. Moreover, it brings the ability to control interactions between the

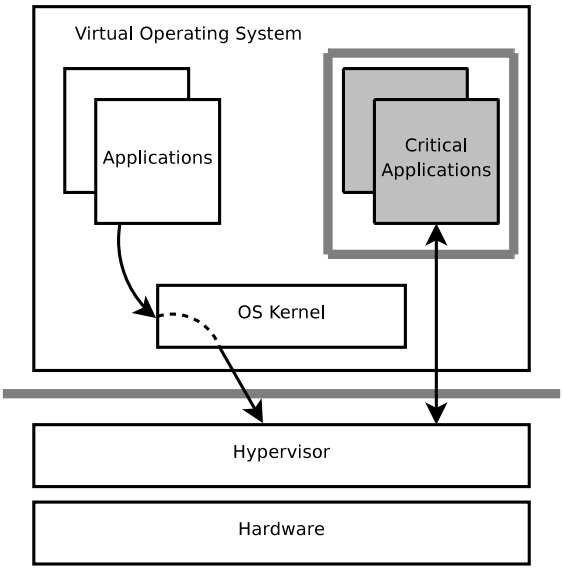


Figure 16: Hypervisor with two isolation barriers (in grey)

OS and the underlying hardware that can not be done even by HIDS. Those special IDS combine the advantages of HIDS and NIDS with the ability to manage hardware events. By doing so, they are able to have a very precise view of what happens in the VM. Moreover, they are able to monitor network events and acting as network IDS too by plugging on the software network bridges. Not only IDS can be plugged to those security interfaces on hypervisors. Antivirus and other malware detections systems can be used to leverage the new capacities of virtualization technologies.

For example, a virtual IDS is able to detect and block a network packet containing a known malicious payload. Contrary to a network IDS that is only able to block the network packet after it has been sent and during its travel on the network, the virtual IDS can block the syscall. Indeed, by controlling system events, a virtual IDS is able to parse the arguments sent to a syscall. A virtual IDS is able to read the arguments of the "write" syscall on the "socket" and particulary the data field. Then, it can verify if it contains a known malicious payload and if it is the case, deny the syscall. In this case, the payload will not travel on the network. Moreover, using this method, the virtual IDS is able to detect payload even in SSL sockets by inspecting the data sent through the socket before it has been encrypted. Contrary to a host IDS that is running on each operating system, the virtual IDS is able to run one time for all the VM running on the same physical computer using special API in the hypervisor (like VMI for VMWare). And a host IDS can not be trusted

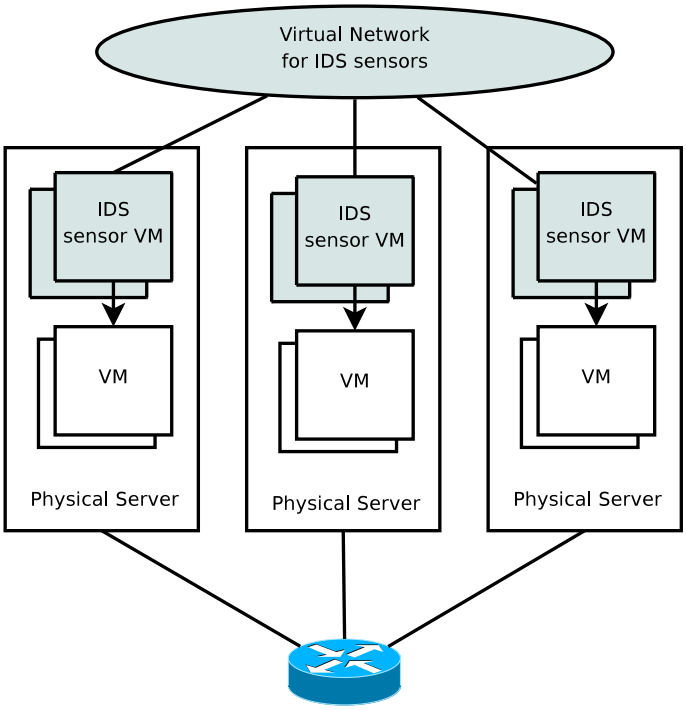


Figure 17: Virtual Distributed Intrusion Detection System

after the OS kernel corruption (i.e. because, in case of kernel corruption, the host IDS can be stealthy modified) whereas a virtual IDS is isolated from the VM and its kernel and can be trusted even after VM total corruption.

The drawback is that, like the auditing and monitoring process, the API used to connect IDS or other security components can be used to leverage new specialized malwares. Those malwares can be able to inspect memory of the entire computer i.e. VM and hypervisor. Also, the API can include flaws (e.g. buffer overflow) allowing to execute code at the hypervisor level.

4.4 Global Security Framework

With virtualization, the complexity of networks and virtual networks grows fast with a large number of VM and hypervisors. Virtualized Distributed IDS [KC05] (vDIDS) are becoming essential to closely monitor those large virtualization facilities.

The figure 17 shows that in vDIDS, each hypervisor runs multiple IDS sensors

in dedicated VM, and all are linked by a private virtual network. As shown on figure 17, they are not limited to one kind of sensors but use many of them to have a global view of the virtualized information system. Moreover, some of these sensors are specialized for the detection of VM interactions. Others are specialized for the detection of illegal shared physical resources between VM. For example, they can use classical NIDS. They can also use IDS that are specially crafted for VM technology like IDS outside VM. Finally, virtualization brings easier and cheaper architectures to deploy. Moreover, by using specialized and isolated network for communication between IDS sensors with virtual networking, the monitoring network can be isolated from the classical network (i.e. the monitored one) without any expense.

The drawback is that the framework is expensive to deploy. The global security framework requires to run a VM (at least) on each hypervisor and to setup the virtual private network. Moreover, the framework also requires the setup of each underlying security components (e.g. HIDS, NIDS).

4.5 Distributed Access Control

Using advanced security components like mandatory access control on system calls like SELinux [LS01] and MAC on inter-VM interactions [SJV+05b], a global mandatory access control has been proposed [MTS+06]. It ables to control at a fine grained level the interactions between applications in different virtualized OS. In practice, it is possible to control the interactions between two applications that are located in two different VM if they use overt channels to communicate. This could not been done without virtualization technologies. Distributed access control grows with high-interaction VM in grid computing and other distributed architectures like Cloud.

The major drawback are those of Mandatory Access Control, already mentioned in section 3.4.2.

4.6 Forensic

Common system loggers depend on the integrity of the underlying OS they monitor. For example, syslog events can not be trusted after the corruption of the kernel on the monitored operating system. Moreover, most of the time, the events do not permit to replay the attacks and are not complex enough to contain deep information. With the missing of such information, the replay mechanism misses critical events that include non-deterministic events[12] that are created during attacks. With

[12]Non-deterministic events are the one where loads, time, random generator or other non-deterministic mechanisms are used.

non-deterministic events, it becomes impossible to replay as identical a previous program execution on VM.

The virtualization technology allows to replay the system's execution before, during and after the attack using checkpointing and resuming capability. This permits to go back in time and observe the exact behavior of an OS from the start to the end of an attack [DKC+02]. It can be done by logging all the input/output of a VM at the virtualized hardware layer level. Under classical OS, it is almost impossible to achieve the same goals because low-level hardware logging is complex to do. Moreover, those events can be corrupted as soon as the attacker can interact with the system.

[DKC+02] proposed the ReVirt system ("Replay Virtual machines") that allows to log at the hypervisor level every operation made in a VM. It permits to replay an almost perfect runtime. It is particularly useful to study attacks because it permits to replay it over and over with always the same runtime. Running the same exploit on the same system can cause a different runtime because of multiple parameters (e.g. time or the load of the computer). Their system has been applied on several virtual machines:

- POV-Ray: a VM running raytracer application;
- kernel-build: a VM that recompiled its kernel continuously;
- NFS kernel-build: another VM recompiling its kernel continuously;
- SPECweb99: a VM running the SPECweb99 benchmark;
- daily use: a VM used for classical network usage (i.e. web browsing).

As shown in the table 6, the overhead of running VM with logging facility is small: always less than 8% compared VM without logging facility. Moreover, the overhead in term of log size is also quite small. Finally, the replay is at least as fast as running a normal VM and at best is 97% faster.

The drawback is that the forensic framework allowing logging and replaying of virtual machine's runtime is not trivial to setup. It required to install the logging facility for each VM. Moreover, the log files are large (more than 1GB per day) and need to be stored.

4.7 Summary

In this section, we have presented all the security mechanisms and components that are currently or will soon be provided by virtualization technologies/vendors or studied by researchers in order to prevent attacks of section 2 and forthcoming

Workload	Runtime with logging	Log growth rate	Replay runtime
POV-Ray	1.00	0.04 GB/day	1.01
kernel-build	1.08	0.08 GB/day	1.02
NFS kernel-build	1.07	1.2 GB/day	1.03
SPECweb99	1.04	1.4 GB/day	0.88
daily use	≈ 1	0.2 GB/day	0.03

Table 6: Logging and replaying overheads and logs size for ReVirt.

ones to really happen on virtualized architectures. We presented what are existing means to protect hypervisors according to whether they are type I or type II ones. For type II hypervisors, the main security risks rely on the fact that they simply are classical processes, like any other one, running on the host OS. It is quite simple to take the control over it. Undoubtedly, the best way to secure type II hypervisors is to switch to type I ones.

For type I hypervisors, the security problems remain numerous. In addition to reduce the size of code needed to be trusted (i.e. the TCB), and the (not so good) possibility of running classic security components, there exist some new approaches that aim at securing the hypervisor layer itself. GuarHype protects hypervisors, acting itself as a meta-hypervisor.

Some other approaches focus on virtual machines protection. Some aim at guaranteeing the integrity of virtual machines (using software, hardware (TPM, or mixed (vTPM) technics). Some other prefer to observe and control what is happening in the VM: they audit or monitor the system.

Some approaches focus on security of shared resources, again using soft or hardware technics. The most advanced approaches try to benefit from the strong security potential that present Access Control mechanisms, such as Mandatory access control between VM (sHype) and even between physical hosts.

More general approaches are also appearing and propose global security frameworks in order to supervise the virtual network, and to ensure a distibuted access control and even to be able to provide forensics and offline attacks analysis.

5 Conclusion

No one can ignore the explosion in the computer virtualization market. Virtualization appears anywhere: on desktop computers, servers, clusters, grids, mobile phones, and in cloud computing. As virtualization groups numerous physical computers into virtualized ones on only few servers, reaching high privileges on such

architectures is very attractive for attackers. As the technology evolves fast, many flaw appear, and security updates are (for the moment) not so often than on some classical unvirtualized systems.

Along this report, we have presented the different known attacks on virtualization technologies: attacks from VM to hypervisors, between VM themselves or from hypervisors to VM. All of these attacks are able to break the three main security properties: integrity, confidentiality and availability. That is why having a proper implementation of hypervisors and security components dedicated to virtualization security, is essential. We presented the actual security measures provided by actual technologies, such as protection of VM and hypervisors, and also of virtual networks. But many things remain to be done, as many of the presented measures are still in the stage of prototypes, sometimes even only concepts. Actually, the reality in security is merely isolation facilities. But virtualization and isolation are not security components. In addition, they must be secured.

Moreover, it is more than probable that first viruses for virtualization layers arrive soon.

Virtualization vendors need thus to increase the quality of hypervisor while reducing the size of their TCB. They should quickly evaluate them (or be able to provide strong security level assertions) in order to state minimum security guaranties.

Futures security trends, as the size of the "virtual(ized) world" exponentially increases, should provide strong security properties enforcement, as hypervisors have the task of isolate OS and control accesses and interactions between VM. Of course, security components, whether outside or inside VM, such as MAC mechanisms, IDS, anti-malwares, anti-viruses, distributed IDS will be usefull to manage globally all the virtual machines. In addition to those softwares components, processors vendors still work hard to propose strong protection mechanisms at the processor level. But even there, we have seen that many ways exist to bypass the protection. However, it is possible that processors vendors soon propose on-chip hypervisors, verified by TPM modules.

But new potential usages of virtualization may bring new flaws, that maybe noone has even imagined.

For example, the gold age of "Cloud computing" may lead to virtual architectures with thousands or even millions of VM. Everyone has to understand that the entire computing world seems to be going to be totally virtualized (at least the servers and their services). If no solutions allowing strong security and easy to use administration tools are soon provided, security in such architectures may quickly tend to zero, and the computing world may collapse.

References

[AA06] K. Adams and O. Agesen. A comparison of software and hardware techniques for x86 virtualization. In *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, pages 2–13. ACM New York, NY, USA, 2006.

[ACDA09] E. Wong A. Clement, M. Marchetti, M. Dahlin, and L. Alvisi. Making byzantine fault tolerant systems tolerate byzantine faults. In *Proceedings of the 6th USENIX Symposium of Network Systems Design and Impementation (NSDI '09)*, Berkeley, CA, USA, 2009. USENIX Association.

[BCG+06] Stefan Berger, Ramón Cáceres, Kenneth A. Goldman, Ronald Perez, Reiner Sailer, and Leendert van Doorn. vtpm: virtualizing the trusted platform module. In *USENIX-SS'06: Proceedings of the 15th conference on USENIX Security Symposium*, Berkeley, CA, USA, 2006. USENIX Association.

[BD96] Matt Bishop and Michael Dilger. Checking for race conditions in file accesses. *Computing Systems*, 9(2):131–152, Spring 1996.

[BSS+96] Lee Badger, Daniel F. Sterne, David L. Sherman, Kenneth M. Walker, and Sheila A. Haghighat. A domain and type enforcement unix prototype. In *Proceedings of the Fifth USENIX UNIX Security Symposium*, pages 127–140, 1996.

[BY06] Muli Ben-Yehuda. Using iommus for virtualization. In *Linux Symposium 2006*, July 2006.

[CGL+08] Xiaoxin Chen, Tal Garfinkel, E. Christopher Lewis, Pratap Subrahmanyam, Carl A. Waldspurger, Dan Boneh, Jeffrey Dwoskin, and Dan R.K. Ports. Overshadow: a virtualization-based approach to retrofitting protection in commodity operating systems. In *ASPLOS XIII: Proceedings of the 13th international conference on Architectural support for programming languages and operating systems*, pages 2–13, New York, NY, USA, 2008. ACM.

[CMC62] F. J. Corbató, Merwin-Daggett M., and Daley R. C. An experimental time-sharing system. In *Proc. Spring Joint Computer Conference (AFIPS)*, page 335–44, 1962.

[CZL08] Martim Carbone, Diego Zamboni, and Wenke Lee. Taming virtualization. *IEEE Security and Privacy*, 6(1):65–67, 2008.

[DKC+02] George W. Dunlap, Samuel T. King, Sukru Cinar, Murtaza A. Basrai, and Peter M. Chen. Revirt: enabling intrusion analysis through virtual-machine logging and replay. *SIGOPS Operating System Review*, 36(SI):211–224, 2002.

[ESZ08] Shawn Embleton, Sherri Sparks, and Cliff Zou. Smm rootkits: a new breed of os independent malware. In *SecureComm '08: Proceedings of the 4th international conference on Security and privacy in communication netowrks*, pages 1–12, New York, NY, USA, 2008. ACM.

[Fer06] P. Ferrie. Attacks on virtual machine emulators. Symantec Advanced Threat Research, Dec 2006.

[Gal69] S. W. Galley. Pdp-10 virtual machines. In *Proceedings ACM SIGARCH-SIGOPS Workshop on Virtual Computer Systems*, pages 30–34. ACM, 1969.

[GR03] Tal Garfinkel and Mendel Rosenblum. A virtual machine introspection based architecture for intrusion detection. In *Proc. Network and Distributed Systems Security Symposium*, pages 191–206, 2003.

[GR05] T. Garfinkel and M. Rosenblum. When virtual is harder than real: Security challenges in virtual machine based computing environments. In *10th Workshop on Hot Topics in Operating Systems*, 2005.

[Hir07] R. Hiremane. Intel virtualization technology for directed i/o (intel vt-d). *Technology@ Intel Magazine*, 4(10), 2007.

[HUL06] Gernot Heiser, Volkmar Uhlig, and Joshua LeVasseur. Are virtual-machine monitors microkernels done right? *SIGOPS Oper. Syst. Rev.*, 40(1):95–99, 2006.

[IBM72] IBM. Ibm corporation, ibm virtual machine facility/370 introduction, gc20-1800. 1972.

[Inc97] Sun Microsystems Inc. *OpenBoot Command Reference*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1997.

[Kar05] P.A. Karger. Multi-level security requirements for hypervisors. In *Computer Security Applications Conference, 21st Annual*, page 9, 2005.

[KC05] Kenichi Kourai and Shigeru Chiba. Hyperspector: virtual distributed monitoring environments for secure intrusion detection. In *VEE '05: Proceedings of the 1st ACM/USENIX international conference on Virtual execution environments*, pages 197–207, New York, NY, USA, 2005. ACM.

[KC06] ST King and PM Chen. SubVirt: Implementing malware with virtual machines. In *2006 IEEE Symposium on Security and Privacy*, page 14, 2006.

[LS01] Peter Loscocco and Stephen Smalley. Integrating flexible support for security policies into the linux operating system. In *Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference*, pages 29–42, Berkeley, CA, USA, 2001. USENIX Association.

[LVBP05] J. Laganier and P. Vicat-Blanc Primet. Hipernet: A decentralized security infrastructure for large scale grid environments. In *GRID '05: Proceedings of the 6th IEEE/ACM International Workshop on Grid Computing*, pages 140–147, Washington, DC, USA, 2005. IEEE Computer Society.

[MC00] D.J. Magenheimer and T.W. Christian. vBlades: Optimized paravirtualization for the Itanium processor family. 2000.

[MD73] S.E. Madnick and J.J. Donovan. Application and analysis of the virtual machine approach to information system security and isolation. In *Proceedings of the workshop on virtual computer systems table of contents*, pages 210–224. ACM New York, NY, USA, 1973.

[MHW00] Ron Minnich, James Hendricks, and Dale Webster. The linux bios. In *ALS'00: Proceedings of the 4th annual Linux Showcase & Conference*, pages 21–21, Berkeley, CA, USA, 2000. USENIX Association.

[MMH08] Derek Gordon Murray, Grzegorz Milos, and Steven Hand. Improving xen security through disaggregation. In *VEE '08: Proceedings of the fourth ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, pages 151–160, New York, NY, USA, 2008. ACM.

[MS70] RA Meyer and LH Seawright. A virtual machine time-sharing system. *IBM Journal of Research and Development*, 9(3):199, 1970.

[MTS+06] McCune Jonathan M., Jaeger Trent, Berger Stefan, Caceres Ramon, and Sailer Reiner. Shamon: A system for distributed mandatory access control. In *ACSAC '06: Proceedings of the 22nd Annual Computer Security Applications Conference*, pages 23–32, Washington, DC, USA, 2006. IEEE Computer Society.

[Mur08] Junichi Murakami. A hypervisor ips based on hardware assisted virtualization technology. In *BlackHat USA*, 2008.

[OOY08] Koichi Onoue, Yoshihiro Oyama, and Akinori Yonezawa. Control of system calls from outside of virtual machines. In *SAC '08: Proceedings of the 2008 ACM symposium on Applied computing*, pages 2116–1221, New York, NY, USA, 2008. ACM.

[PG73] G.J. Popek and R.P. Goldberg. Formal requirements for virtualizable third generation architectures. *Trans. on Computers C*, 22:644–656, 1973.

[QT06] Nguyen Anh Quynh and Yoshiyasu Takefuji. A real-time integrity monitor for xen virtual machine. In *ICNS '06: Proceedings of the International conference on Networking and Services*, page 90, Washington, DC, USA, 2006. IEEE Computer Society.

[RKK07] Thomas Raffetseder, Christopher Krügel, and Engin Kirda. Detecting system emulators. In Juan A. Garay, Arjen K. Lenstra, Masahiro Mambo, and René Peralta, editors, *ISC*, volume 4779 of *Lecture Notes in Computer Science*, pages 1–18. Springer, 2007.

[Rut08] J. Rutkowska. Introducing blue pill. In *BlackHat USA*, 2008.

[SJV+05a] R. Sailer, T. Jaeger, E. Valdez, R. Caceres, R. Perez, S. Berger, JL Griffin, L. Van Doorn, I.B.M.T.J.W.R. Center, and NY Hawthorne. Building a MAC-based security architecture for the Xen open-source hypervisor. In *Computer Security Applications Conference, 21st Annual*, page 10, 2005.

[SJV+05b] Reiner Sailer, Trent Jaeger, Enriquillo Valdez, Ramon Caceres, Ronald Perez, Stefan Berger, John Linwood Griffin, and Leendert van Doorn. Building a mac-based security architecture for the xen open-source hypervisor. In *ACSAC '05: Proceedings of the 21st Annual Computer Security Applications Conference*, pages 276–285, Washington, DC, USA, 2005. IEEE Computer Society.

[SVJ+05] R. Sailer, E. Valdez, T. Jaeger, R. Perez, L. Van Doorn, J.L. Griffin, and S. Berger. sHype: Secure hypervisor approach to trusted virtualized systems. *IBM Research Report RC23511*, 2005.

[UNR+05] R. Uhlig, G. Neiger, D. Rodgers, AL Santoni, FCM Martins, AV Anderson, SM Bennett, A. Kagi, FH Leung, and L. Smith. Intel virtualization technology. *Computer*, 38(5):48–56, 2005.

[Var97] M. Varian. Vm and the vm community: Past, present, and future. In *SHARE*, volume 89, pages 9059–9061, 1997.

[Woj08] R. Wojtczuk. Subverting the xen hypervisor. In *BlackHat USA*, 2008.

[WR09] R. Wojtczuk and J. Rutkowska. Attacking intel® trusted execution technology. In *BlackHat DC*, 2009.