Profiling User Activities On Guest OSes in a Virtual Machine Environment

by

Enrico Perla

A Dissertation submitted to the University of Dublin,

in partial fulfillment of the requirments for the degree of

Master of Science in Computer Science

2008

Declaration

I, the undersigned, declare that this work has not previously been submitted as an exercise for a degree at this, or any other university, and that unless otherwise stated, is my own work.

Enrico Perla

September 1, 2008

"A man travels the world over in search of what he needs, and returns home to find it."

– George Moore

To Franco, Lallamaria and Michela

Acknowledgments

To inode, lambdawar, akira, revenge, iotun and antani...

... talking with you is always special.

Enrico Perla

University of Dublin, Trinity College

September 2008

Profiling User Activities On Guest OSes in a Virtual Machine Environment

Enrico Perla
University of Dublin, Trinity College, 2008

Supervisor: Stephen Farrell

Virtual Machines (VM) are becoming the norm in large scale server environments. Typically, it is assumed that guest operating systems (OSes) running under the host OS cannot interfere with, or observe, one another. Clearly however, if a process can escape the *jail* of the guest OS and run in the host OS, then it could invalidate this assumption. This may not be an easy task for the attacker. We have taken Xen, a widely used opensource virtualization solution, and we have investigated whether a malicious guest OS may usefully succeed in profiling activity on other guest OSes or the host OS without having to break out of the jail. We discovered three design flaws that allow a guest OS to identify guests running on the same host OS, map them to their network address and, in some cases, take control over their network activity. All the attacks have been glued into a single, highly automated, tool, *xenophobia*.

Contents

List of Figures

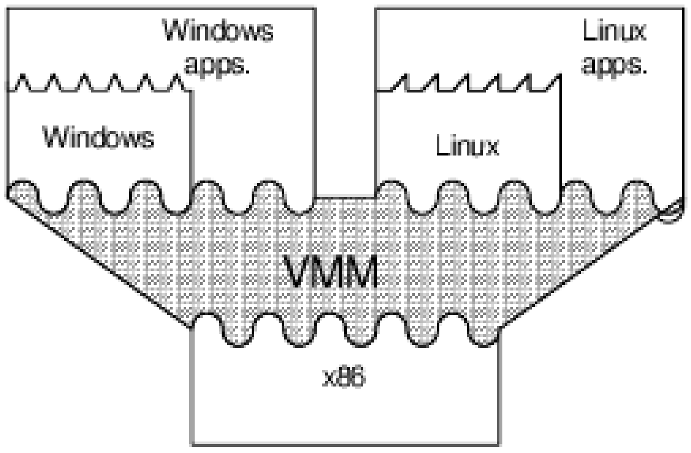
Chapter 1

Introduction

The term *Virtualized environment* has multiple meanings. A common distinction [1] is between *process level* virtual machines, aimed at containing a single process and that permit independence from the underlying operating system (ex. Java [2], .NET Framework [3]), and *system level* virtual machines, where a set of hardware resources is virtualized to allow different *guest* operating systems to run on the same machine at the same time. In this thesis we focus on system level virtual machines and, in particular, we will analyse a range of *security issues* related to the *Xen Hypervisor*, an OpenSource project that is more and more establishing itself as one of the leading projects in the field. This is shown by, for example, the recent purchase of XenSource by CitriX [4] for $500M and the choice of Sun Microsystems to base one of their own virtualization solutions [5] , xVM, on the Xen's project.

The general idea behind *system level* virtualization is to interpose a piece of software between the hardware and the operating system that will take control of the available physical resources (CPU, memory, I/O devices) and will manage to partition or share them among different running operating system instances. One of the reasons virtualization systems have gained more and more interest recently is the security enhancement that they provide, especially thanks to *isolation* and *replication* (Fig. 1.1).

Figure 1.1: System Level Virtual Machine Monitor (taken from *"Virtual Machines"*, Smith and Nair)



Guests are *isolated* one from the other. That means that a malicious guest should not be able to interfere with the correct behaviour of the others. Since creating multiple guests is a *cheap* operation (cheaper, indeed, than having multiple physical machines...), different services can each receive their own isolated system and a compromise of one of them should not affect the others. Guests can also be *replicated*, to improve the fault-tolerance of the system. Moreover, suspicious software can be tested on a *twin guest* and the effects can be monitored before trusting it.

From the examples above (which are far from being a complete overview of the security advantages that system virtual machines may introduce) it should be clear why virtual machines are getting more and more attention and deployment. This leads, logically, also to a growing interest in attacking VMs. Attacks range from detecting [6] the presence of a virtual machine monitor, exploiting or *dossing* (D.o.S., Denial of Service) the VMM and backdooring[1] it. Less work has so far been done so far to analyse the design decisions that may help a guest in *breaking* the isolation property of the virtual machine environment *without* raising its own privileges.

This "*lack*" of work is particularly visible when we come to talk about the Xen Hypervisor. Xen uses a design in which very little code is kept inside the hypervisor itself and a lot of the work is left to a privileged domain, known as *domain zero (dom0).* All the other, unprivileged, domains are known as *domain U (domU).* A couple of tools and papers have been presented [7] [8] that aim at attacking the various domU *from the dom0* or that aim at implementing hypervisor-level backdoors/rootkits[2]. All this work is based on the assumption that the attacker has already managed to break into the dom0 and has full privileges on it, which may be a hard task to accomplish and, thus, a relatively uncommon situation. In this thesis we will focus on what a domU can do against other domains *without raising its privileges*, that is, without managing to break into dom0. We will show how, thanks to a couple of design flaws, other domains running under the same VMM can be identified and some attacks that can be carried out thanks to the information discovered. These attacks may allow an attacker to get complete

[1] *backdooring* refers to the programs that an attacker installs to maintain access on a compromised host

[2] a *rootkit* is the set of tools that an attacker uses once the box has been compromised

control over the outgoing network traffic of the victim and evade common IDS (Intrusion Detection Systems) configurations.

1.1 Outline

This thesis is divided in four main chapters:

Chapter 2 provides background information about modern operating systems and the approaches used to virtualize them. This chapter ends up with an overview of the Xen VMM.

Chapter 3 presents a survey of the most common attacks that can be carried out against a Modern Operating System. The discussion then moves to evaluating how virtual machines affect these kind of attacks, showing the security improvement that VMMs can give but also analysing the fact that VMs can themselves become a new target for attackers and expose new vulnerabilities. The discussion ends with an overview of the security features of Xen.

Chapter 4 is the heart of the thesis. Three different attacks are described in detail, together with an in-depth description of the affected Xen internals.

Chapter 5 describes how all the theoretical attacks discussed in the chapter four can be efficiently implemented into a single, highly automated, tool.

Chapter 2

Background

2.1 Modern Computer Architecture

Any virtualization system is built upon a real system and it is up to the virtual machine monitor (VMM) to map virtualized resources onto physical ones. Understanding the behaviour of the components of a real system is thus of critical importance in understanding and attacking a virtual machine environment. A modern computer architecture can be basically divided in three main components: processor(s), memory and I/O devices.

The *processor* is responsible of fetching the instructions to be executed from the main memory, decoding and then executing them. Modern processors usually have a superscalar [9] design: multiple instructions can be fetched and decoded in a single clock cycle and execution may happen out-of-order (that is, instructions may be re-arranged for performance reasons by the processor itself). The processor *Instruction Set Architecture (ISA)* usually defines a set of registers (or other fast-memory storage) and a set of instructions that can be executed. The processor has usually at least two different modes of operation: *user* and *supervisor*. The available ISA in user mode is generally a subset of the one available in supervisor mode. The instructions available only in supervisor mode are referred to as *privileged* or reserved. An attempt to execute a reserved instruction when in user mode usually results in a trap; on some architectures, as the x86 [10], there are privileged instructions that if executed in user mode do not trap and, instead, get considered as no-op or have a different behaviour (ex. POPF) [11]. As we will see in the following section about the principles of virtualizability, this is a fundamental property to be taken in account when designing a VMM.

We said that the processor fetches the instructions from the *memory*. Moreover, any given program running on the system needs to keep some data for a given amount of time. All these tasks (and more) are accomplished by the *memory subsystem*. This subsystem is composed of a hierarchical set of components: processor registers, cache memory, main memory (RAM) and disk storage[1]. Starting from the registers and going towards the disk storage, the components become slower, more capacious and less expensive. From the point of view of memory management, the first two components (processor registers and cache memories) are usually managed directly by the hardware[2], while the main memory and the disk are handled by the Operating System software.

[1]We consider here the disk storage as a component of the memory subsystem because swapping is a common practise [12] [13] in any modern operating system. Strictly speaking, at this point of the discussion, the disk should be considered an I/O device. Similar considerations can be made for the processor register, they are part of the processor, but they basically act as a very small and very fast cache

[2]TLBs (Translation Lookaside Buffers) have not been mentioned explicitly but are considered as part of the generic *cache memories approaches*. TLBs can be both hardware controlled (as in the x86 [10] architecture) or software controlled (as in the UltraSPARC [14] architecture)

Modern computers have lots of devices: disks, ethernet cards, mouse, keyboard, video, sound cards, etc. The `I/O subsystem` is made of all the buses that connect those devices to the processor and memory. There is a set of standard buses, as the PCI or the USB ones, so that new devices can be manufactured and integrated with less effort. The bus is used as the channel of communication between the devices and the processor/memory: commands to the devices are issued on the bus and the results go back through it again. To handle multiple devices on the same bus, usually a bus arbiter [15] is present. Modern I/O devices generally have a controller [15] too, which stands in between the physical device and the bus, to improve performances.

2.2 Modern Operating System Implementation

The second step towards better understanding the implementation and the reasons behind virtualization systems is an analysis of the relevant design principles of modern operating systems. It should be of no surprise to discover that some virtualization concepts are present in operating systems design. A first example is the *virtual memory* implementation: every running process is given the illusion of having the whole *memory address space* for itself. The size of the address space depends on the number of bits that can be used to specify a given memory address (as stated by the ISA definition [15]): since the memory is usually byte addressable, if there are n bits available the maximum size is 2^n bytes. Common sizes for modern systems are 32 or 64[3] bit. Obviously, especially for the 64 bit case, there

[3]It is common, for performance reasons, in 64 bit architectures and operating systems to use only a subset of the addressable space, for example 48 bits (as is the case on the UltraSPARC architecture)

can not be such an amount of physical memory in the system. To achieve this illusion, the physical memory is divided in fixed-size small units, called *page frames* [12] [13]. Only the necessary pages are allocated to requesting processes and pages can be *swapped* out to disk and paged back in when necessary. This whole approach, which is by far the most common, goes under the name of `demand paging` [12] [13].

Virtualization principles can be seen not only in memory management, but also in filesystem/disk handling. In fact, virtual memory and filesystem implementation are tightly linked, as is demonstrated by the page cache [12] [13], which uses the virtual memory to keep the most commonly used disk contents in memory and thus increase performance. Moreover, disks are large, so it is common to partition them, creating a set of smaller "virtual disks". Each partition can be formatted with a different filesystem, so that different operating system can coexist on the same physical hard drive (even if only one can be active).

In modern operating systems there is a distinction between *kernel land* (the area where the kernel resides and executes) and *user land* (the area where all the normal programs run) [16] [17] [18]. The kernel is the only piece of code running with supervisor privileges and thus is the only piece of code that has access to the whole ISA instruction set. Interrupt and trap service routines reside in kernel land and are generally accessed through vectors/tables. Interrupts and traps are also the common way for a user land process to request a kernel service or for an I/O device to signal that an operation has been performed or a problem was encountered.

I/O devices are handled by the kernel through so called *device drivers*, add-on modules that can be loaded or activated at runtime, to react to changes such as the insertion of a USB drive. Those device drivers generally run in supervisor mode.

2.3 Design Principles of System Virtual Machines

Modern operating systems give the illusion of multiple processes running at the same time by switching the available resources from one process to another every now and then. System virtual machines bring this idea one step further, creating multiple isolated environments where different operating systems (or different instances of the same operating system) run *at the same time*. It is the job of the VMM (Virtual Machine Monitor) to control and manage the access to the shared physical resources on the *host* machine among the various *guests*. Access to hardware devices can be granted in different ways: the same device can be partitioned among the different operating systems (ex. hard disk) or the access to the device can be fully given only to the actual running image and switched every time (e.g. keyboard). Since the VMM is a piece of software, some devices can even be totally emulated in software, without the need of an hardware counterpart (e.g. virtual ethernet devices). Popek and Goldberg in their paper *"Formal Requirements for Virtualizable Third Generation Architectures"* [19] define three mandatory properties for the VMM:

- **Fidelity** the environment for the execution of the processes has to be "*essentially identical with the original machine*", with two notable exceptions: differences are accepted if due to timing dependencies or due to a restricted set of available system resources.
- **Performance** the VMM must cause a minor performance impact for the running processes.

- **Safety** the VMM is the only piece of software that controls and handles access to the physical devices.

It is worth mentioning also that the processor's ISA can be *emulated*, so that the guest operating system will run on a different instruction set than the host one. This kind of virtual machine, usually referred to as *emulator*, will not be covered here. The interested reader should check [1] [20].

2.3.1 Native VM systems

The analogies between an operating system and its applications and the VMM and its guests do not stop here. As was discussed before, the operating system runs at a higher privilege level (supervisor mode) in respect to all the userland applications. This allows the kernel to efficiently take full control over the applications, scheduling them on and off the CPU and managing the eventual traps and interrupts from the I/O devices and the running processes themselves. For the same efficiency and control reasons, the first natural approach to design a VMM is to have it run at a *higher privilege level* respect to all the guests. The VMM is thus installed directly on the hardware and is the only piece of software which runs with full privileges. All the guest operating systems are installed on top of the VMM and run at a lower privilege level. Since in most of the architectures (with the notable exception of the IA-32 [10] one) there are only two privilege levels (supervisor mode and user mode) and since the guest operating system has to run at a higher level than theuserland applications that run on top of it, the VMM has to *emulate* the supervisor mode for the guest kernel.

This kind of approach is the one taken by Popek and Goldberg in their analysis and is usually referred to as *classic virtualization* or *native VM* *system.* Popek and Goldberg's paper goes on with a formal analysis of the conditions for classic ISA virtualizability. For simplicity, the real machine is assumed to consist of an uniformly addressable memory and a processor, capable of running in only two modes, supervisor and user and a subset of the instruction set is available only in supervisor mode.

An instruction is defined as *privileged* if executes properly in supervisor mode and traps when issued in user mode. The instruction *must trap*, it can not have a different behaviour in the two modes (for example, behave like a noop if the processor is in user mode). An instruction is defined as *sensitive* if it affects or is affected by the hardware, in particular *control sensitive* instructions are the ones that attempt to change the configuration of a given resource while *behaviour sensitive* instructions are the ones that show a different behaviour depending on the state of a given set of resources. An instruction which is not sensitive is defined *innocuous.* Innocuous instructions can be executed directly on the hardware[4], while sensitive instructions need an intervention by the VMM. With this in mind we can now report the Popek and Goldberg theorem for classic virtualizability:

"For any conventional third generation computer, a virtual machine monitor may be constructed if the set of sensitive instructions for that computer is a subset of the set of privileged instructions"

The fundamental idea stated by this theorem is that the VMM needs a reliable and efficient way to gain control every time a sensitive instruction is

[4]Since the number of innocuous instructions is usually way bigger than the number of sensitive ones, the VMM can execute directly on the hardware a large number of instructions, sensibly improving performances

executed by the guest operating systems. If the sensitive instruction is also privileged, it will *always* trap when executed in user mode and the VMM will have a way to gain control and emulate its behaviour on the guest operating system. This is the main reason why the IA-32 architecture is not virtualizable in classic sense: there are instructions which are sensitive but not privileged. The classic example is the POPF instruction, which copies a 16-bit value saved on the top of the stack to the EFLAGS register. When this instruction is executed in user mode, the value of the interrupt-enable flag (which is modifiable only in supervisor mode) saved on the stack is ignored and not copied in, but no trap is issued. A detail analysis of the virtualizability of the IA-32 architecture (along with a list of the other POPF-like sensitive but not privileged instructions) can be found in Robin and Irvine's paper *"Analysis of the Intel Pentium's Ability to Support a Secure Virtual Machine Monitor"* [11]

The fact that the IA-32 (and, actually, most of the other common architectures) is not virtualizable in native sense does not mean that it is not virtualizable at all. Techniques usually used in process level virtual machines (as Java or .NET) can be employed to solve this problem: the instruction stream can be all interpreted (very inefficient) or it can be scanned and every *critical* instruction can be *patched*, for example inserting a trap or some other way to bring the VMM in. A VMM behaving in this way is usually referred to as an *hybrid virtual machine system.*

2.3.2 Hosted VM systems

In Native VM systems the VMM is installed on bare hardware. That means that the VMM must be the first piece of software installed on the machine.

This might be an undesirable case for users that already have their operating system running and/or that may want to run the virtualization software only occasionally. In this scenario, *hosted VM systems* become quite interesting. The VMM is installed on top of an already available operating system and it just relies on the underlying *hosting* OS services to handle the physical devices. The VMM runs just like any other application at a lower privilege level than the hosting operating system. From the discussion so far it should be quite straightforward to realise that *pure* hosted VM systems are very inefficient. The common approach is to have at least some part of the VMM running as part of the hosting operating system. This can be achieved in two ways:

- the underlying kernel sources can be patched and recompiled.
- a Loadable Kernel Module (LKM) or device driver can be written and loaded at runtime. The code inside an LKM or a device driver runs in supervisor mode inside the kernel.

This kind of virtual machines is usually referred to as *dual-mode VM systems* and is quite popular. For example, the VMWare [21] family of virtualization software is a dual-mode VM system.

2.3.3 Paravirtualization

Paravirtualization is a virtualization approach where the guest operating system is made aware of the presence of a VMM, to which it communicates. This kind of approach greatly simplifies the work of the VMM code, because the running guest will cooperate with the virtualization software to solve common virtualization problems, for example, the sensitive but not privileged instructions, or the handling of the virtual memory of the guest system. As a natural consequence, paravirtualization permits a boost in the overall performance of the guest operating systems.

The paravirtualization approach is at the base, of the Xen project [22] [23], that is the target of this thesis.

2.4 Xen

Xen is an open-source VMM for the x86 architecture. We can divide Xen into two major branches: one, usually referred to as HVM, is designed to run on virtualization-capable processors and permits running unmodified operating systems, the other, based on paravirtualization, aims to achieve close-to-native performance on the guest operating systems running on the x86 platform, a traditionally complex architecture to virtualize efficiently. Xen was originally developed by the System Research Group at the university of Cambridge, as part of the XenoServers project, and its architecture was first presented and described in the 2003 paper *"Xen and the art of Virtualization"* [22]. The October of the same month the first public version of Xen was released.

Since then, Xen has developed to version 3.3 [24], which has extended the support for SMP machines [15], included the support for Intel Physical Address Extension (PAE) [10] (to handle more than 4GB of RAM memory on 32 bit machines) and introduced the support for hardware virtualization technology, so that unmodified guest operating systems can be run under Xen.

We will focus on the paravirtualization-based Xen implementation, in which the guest operating systems are made aware of the presence of the

VMM (hypervisor). It is important to note that only the guest kernel has to be made aware of the existence of the hypervisor, all the userland applications do not need any change. Xen comes with official *patches* for operating systems, Linux [25] and NetBSD [26]; in this thesis we will focus on the *XenLinux* project [27], which combines Xen with Linux. The central notion in Xen is the one of *domains*, the guest OSes that the hypervisor maintains. The approach that Xen takes is to have basically two different kind of domains, with different privileges:

- **Domain 0 or dom0**, this domain is started directly by Xen at startup and is the one on top of which all the other domains are built and executed. The key idea behind this approach is to have inside the hypervisor only the logic to start/manage/stop domains and to leave to a *more privileged* domain the responsibility of handling the physical devices. This approach has the double advantage of simplifying the VMM code and obtaining a wide hardware support (every device driver written for the domain 0 guest kernel can be used).

- **Domain U or domU**, this kind of domain is the guest domain *in the traditional sense.* These domains are created and started from the dom0 through an userland process called *xend* [28] [29]. Commands are issued to xend over an HTTP interface and an userland command-line tool, *xm*, is provided for this purpose.

NetBSD and Linux are not the only two operating systems supported. Ports of other operating systems (e.g. Solaris) are available. The Xen port for the OpenSolaris operating system is known as xVM.

Chapter 3

State of the Art

The ultimate goal of an attacker is to gain control over the system and/or steal information from it. Usually the two things are tightly linked: grabbing as much information as possible from the running operating system is common way to start a successful attack and obtaining elevated rise of the privileges is the main step towards reaching hidden and, eventually, more sensitive information. This chapter describes a recent typology of vulnerabilities and attacks that have been carried out against Modern Operating Systems and outlines where and why Virtual Machines can help towards improving overall system security. But VMs are still another layer of code that can be targeted and may have (even yet unknown) weaknesses and issues. This is still an area that is gaining more and more interest from the community just as VM solutions are becoming more and more widespread. The last section focuses on Xen and the related security approaches and issues seen so far.

3.1 Systems Security Attacks

Writing exploits to raise privileges has always been a fascinating art. In the beginning it was mainly done in userland. Running daemons were audited and their flaws exploited to gain remote access to a system and local root-privileged processes were targeted to raise the local privileges. At some point, operating system designers started to insert security patches in the kernel, with the aim of stopping or at least making harder this kind of attacks [30] [31]. Randomisation was added to memory allocation, entire portions of memory were made non-executable (emulating the non-executable bit on architectures, as the x86, which do not support it naively) and daemons started to be chroot'ed and jailed, to mitigate the effects of a successful exploit [32] [30]. The chroot/jail approach is particularly interesting for our discussion on virtual machines system. A chroot'ed process is a process which runs with a *restricted* and *crafted* view of the machine. The process can not access files outside this virtual area and an attacker successfully exploiting such a process is fundamentally *jailed* inside it.

Many different techniques have been developed to break a chroot environment. The FreeBSD jail [33] implementation and the GRSecurity [34] chroot enhancement aim to stop most (if not all) of this attacks. But even in the most secure configuration available, there is always an entity running outside everychroot/jail (and out of most of the other protections available): the **kernel**.

Kernel attacks, exploits and shellcode[1] have been (and still are) very popular in the latest years [35] [36] [37]. In such exploits, the correctness of

[1]A shellcode is a sequence of assembly instructions to which the attacker redirects the flow of the exploited software

the design is mandatory: a single mistake can make the whole system crash. On the other side, the possibilities available to an attacker once it is able to execute code at kernel level are just limited by imagination. Many reliable exploits for different operating systems have been released and techniques to increase the chances of success (even up to the always-dreamed *one-shot*, 100% reliable, exploit) have been presented in different papers [37] [38] [39].

Even if it is probably the most effective one, exploiting is just one step inside a complete, well performed, attack to a system. As we said at the start of this chapter, information and control are the two main goals of an attacker. It is common for an attacker, once he has gained full privileges over the system, to install some kind of code that will grant him a safe way to *remain* on it. This code usually takes the name of *backdoor* and is usually part of a larger set of codes which goes under the name of *rootkit*.

A common part of a good rootkit is a *sniffer* [40]. This kind of code silently stays on the compromised box and steals information. Everything can be sniffed: network communications, programs executed, I/O transfers and so on. To see why such a program may be so powerful imagine a classic scenario of a university system being compromised. The system may have a couple of thousands of accounts, for students and teachers that are immediately exposed to the attacker. Now imagine a sniffer that can read the password inserted by each user once he logs in: many users use the same password (or use the same SSH dsa/rsa key citeOpenSSH) to login to different boxes, e.g. the internal lab, their home box, some account on another university they are collaborating with, and so on. With just such a simple sniffer the attacker can quickly (and easily) gain access to many systems.

As systems protections become more complex and more secure, finding a hole and writing a reliable exploit for it is becoming more and more difficult. The ability to sniff or gain somehow information from leaks in the system is, thus, of mandatory importance. While the bad actors improve their techniques and tools, the administrators do not just sit and watch their host getting compromised and sniffed, but work on improving their tools too. One set of tools that we will encounter (and attack) later on goes under the name of *IDS*, *Intrusion Detection System* [41]. Such a program, that can be implemented in software, in hardware or with a mix of the two, checks the accesses to a given system or the data that passes on a local network. In this latter case, it is usually referred to as *NIDS*, *Network* Intrusion Detection System. One of the most famous NIDS is probably *snort* [42]. It is important to point out that the role of the (N)IDS is not to stop/prevent an attack, while to just figure out *if* and *where* it is happening.

A NIDS can be combined with tools that do try to prevent the attacks, like a *firewall* [43]. Such a program defines a set of rules that must be matched by a given session that wants to pass through it. While an attacker can usually figure out the presence of a *firewall*, it usually should not be able to know if an IDS is in place or not. In some configurations, the information gathered by an IDS is used to determine the firewall rules. [44]

Just for completeness, there are two more sets of security tools that it is worth mentioning; one set implements MAC (Mandatory Access Control)/Policy [45] based checks to achieve a fine grained control of the privileges assigned to each running process, the other set comprises all those patches that aim to make the life of exploit developers harder and harder, making the stack and the heap non-executable or randomising the address of libraries, DLLs and modules. [32]

3.2 The System Virtual Machines Hope

In the last section two important security issues were introduced:

- How can we **protect** the kernel or somehow have some **control** over it? That is, how can we defend from an attacker capable of directly targeting a kernel vulnerability ?
- how can we **mitigate** the effects of a successful exploit? In other words, how can we reduce the amount of information available to an attacker after he gains control of the system? How can we limit his range of action?

One can see a system Virtual Machine as a *chroot'ed environment* for an operating system instance. Each instance is *isolated* from the others and can not affect them directly. This *isolation property* is one reason why system virtual machines are gaining so much interest from the security community.

The VMM stands above the running kernel and can thus keep control or protect it. A VMM might be instructed to stop the running image if an attempt is made to modify a sensitive area of the kernel or if any suspicious unexpected activity is registered. Moreover, virtual machines allow for a separation of the services: each daemon that has to be run (webserver, sshd, database, etc.) can be installed on a different guest. This approach has two main advantages:

- **simplicity**: the configuration of the operating system can be tailored to the service that has to be run. It is possible to have dedicated operating system instances for every single service.
- **compromise impact mitigation**: the successful exploitation of one service will give the attacker access only to the guest operating system

running that specific service and will not let him take control on the other services. For example, the compromise of a webserver would not give access to the database system.

The security advantages of virtual machines do not end here. The hypervisor can constantly monitor the execution of a guest and can carry out fine grained logging about its activity. This means that the *history* of a running image can be recorded and, given that the logging has been sufficiently detailed, replayed identically, greatly helping in analysing the behaviour of a malware or in understanding attacker activity. Another interesting aspect is the one of *replication*: different replicas of the same running image can be maintained, so that if a service gets compromised it can be isolated and stopped and a fresh image with the same configuration can be instantiated. In environments such as distributed computing or grid environments, such approaches to create *trusted* virtual machines images have been studied. [46]

A last area in which virtual machines have gained quite a lot of popularity is in the creation of *sandboxes* and *honeypots*. [47] A sandbox is a system where a potentially harmful piece of code (malware) can be executed and analysed. An antivirus designer can, for example, expose a set of vulnerable sandboxes and then monitor the effects and the behaviour of viruses in that environment. Basically the same principle is behind honeypots too: a vulnerable box is exposed to the Internet with the aim of registering the activity of some attacker and eventually discover of unreleased exploits or rootkits.

For all these reasons and potential applications, system virtual machines are gaining increasing popularity. Despite all the advantages they give,VMM are still another piece of code. Just as they add an extra-layer of security, they introduce new potential weaknesses and give new ground to play over to the attackers. This is the topic of the next section.

3.3 Attacking System Virtual Miachines

In the previous section, among the various examples of use (from a security perspective) of system virtual machines, we cited the sandbox/honeypot/-malware analysis scenario. This approach assumes that an attacker or a malware is not able to realise that he is on a virtualized guest. The first range of *attacks* that has been carried on against virtual machines go under the name of *detection.* Many different detection approaches for a wide range of system virtual machines and emulators have been described by Peter Ferrie in his paper *"Attacks on Virtual Machines Emulators"* [6]. The idea behind this kind of attack is that the VMM must have some impact on the behaviour of the guest operating system and that this impact might be exposed even to the userland of the guest: examples are detectable addresses/values of hardware specific tables (ex. IDT, Interrupt Descriptor Table or LDT, Local Descriptor Table), sequences of privileged instruction that will have a different behaviour or more subtle execution/TLB/memory caching time-based attacks.

Detecting a virtual machine monitor in action is indeed interesting for malware developers and attackers: a given virus could decide to remain silent if it realises to be on a virtualized environment. The *next step* in terms of possible attacks is to target directly the VMM to grab information about the other running guest systems (somehow breaking the *isolation* principle or to exploit the VMM itself and thus *raise* the privileges and gain full control over all the guests. Less work has been done so far in this direction.

An interesting paper on this topic was released by Tavis Ormandy [48]. Potential attacks are divided in three different categories:

- Attacks that lead to a *full compromise* of the VMM. The guest environment is capable of exploiting a flaw inside the VMM and raise its own privileges up to the ones of the hypervisor. At that point the attacker has full control over *all* the running guests.

- Attacks that lead to a *leak* of information from the VMM. The attacker gains sensitive information about the hypervisor or the other running guests, breaking the *isolation* property. This kind of information can help to carry on a subsequent attack aiming to full compromise another guest or the VMM.

- **D.o.S.** (Denial of Service) attacks. The attacker is not able to compromise the VMM but manages to crash it or exaust its resources, forcing the other running guests to starvation.

Once again, many different system virtual machines are targeted, both auditing the source code or using a *fuzzer*. [49] The results range from errors inside the interpretation code (sequence of bytes that crash or gets misinterpreted by the VMM) to errors in the handling of I/O devices.

A final range of *attacks* that can involve virtual machines is exploiting VMM features to write high-level rootkits. Examples [50] [51] of this kind of rootkit have been seen using the virtualization extension available on the Intel and AMD processors.

3.4 Xen Security

The Xen design has a couple of interesting approaches from the security point of view. First of all, Xen exploits the x86 characteristic of having more than two privilege levels. In fact, the x86 architecture has four privilege rings. Userland code usually resides at ring3, while kernel-based code stays in ring0. Both ring1 and ring2 are separated and more privileged than ring3 and are not used by any modern operating system. Xen takes advantage of this by putting itself in ring0 and locating all the guest operating systems in ring1. The userland code on the guest systems executes at ring3. Thanks to that, interference from the userland to the guest kernel land and from guest kernel land to the hypervisor are prevented by the hardware. This approach speeds up the overall execution of the guest system, simplifies the hypervisor code and improves the security of the hypervisor and of the virtualization environment.

Xen has furthermore a compact design. The code of the supervisor keeps to a couple of thousand lines, which makes auditing it easier and spot/fix eventual bugs. A lot of work is offloaded to the domain 0, which has to manage the I/O resources and provide access to them. So the domain 0 environment can be considered, to full extent, *critical code*. This situation resembles the one that has been traditionally seen on Unix systems, where there is one user, the root/superuser one, that has full privileges and full control over the OS. The classic approach to mitigate this problem in high-security environments is the *decomposition* [52] of the privileges. For example, one can think to have multiple *controlling domains* for different I/O resources, so that a bug in the handling of one of it would not expose the whole hypervisor to the control of the attacker. This is one of the ideas at the base of the **XSM** (Xen Security Modules) project [53], which aims to create a generalised security framework for Xen. XSM has been introduced from the 3.2 release of Xen. Is based on the approach used by the LSM (Linux Security Modules) project: a set of hooks is distributed in critical areas of the hypervisor code and security modules can use those hooks to get the control and take security measures when one of those codepaths is executed. Among the security modules available, two of them provide Xen a policy-based Mandatory Access Control (MAC) system: the **IBM ACM/sHype** project and **NSA/Flask**. [53] The ACM/sHype one was formerly a standalone project that ported the sHype Hypervisor Security Architecture developed by IBM to Xen. The second one is the porting of the SELinux Flask [54] policy system developed by the NSA (National Security Agency) [55].

Chapter 4

Xen Internals and Design Issues: the Path to Profiling

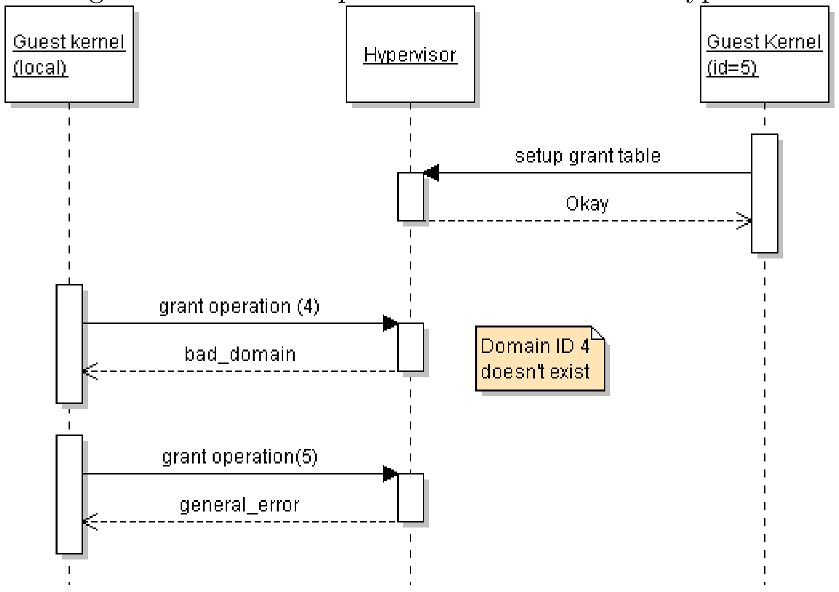
In this chapter we will cover some design issues that "*affect*" Xen and how they can be used for *fun and profit.* A couple of Xen internal subsystems will be described in detail. It is important to note that the discussion and the attacks presented here apply mostly, if not only, to a paravirtualized system. Whenever the attack might be carried on against HVM domains, that will be explicitly stated. These attacks allow a malicious guest to gather information about other running guests, profile their activity, and, eventually, take control over the network traffic that the victim guest generates.

4.1 An overview of the attacks

Before moving to a detailed description of both the design issues that make possible the attacks and the internals of the attacks themselves, we give here a high-level overview of the three attacks. This section aims at giving a quick grasp of each technique and so most of the technical details will be left out. The interested reader should check them in the following sections, each one dedicated to one of the attacks.

The first attack relates to the possibility of enumerating the active domains on a system and discover which domain IDs have been assigned. Domain IDs range from 0 to 65536 (16-bit value) and are assigned by the Hypervisor in incremental order whenever a domain boots. We present here a *sneaky* way to achieve it, evaluating the error code returned from the Hypervisor in response to a specific hypercall (see Fig. 4.1)

Figure 4.1: Guest Maps Domain IDs Based on Hypervisor Replies

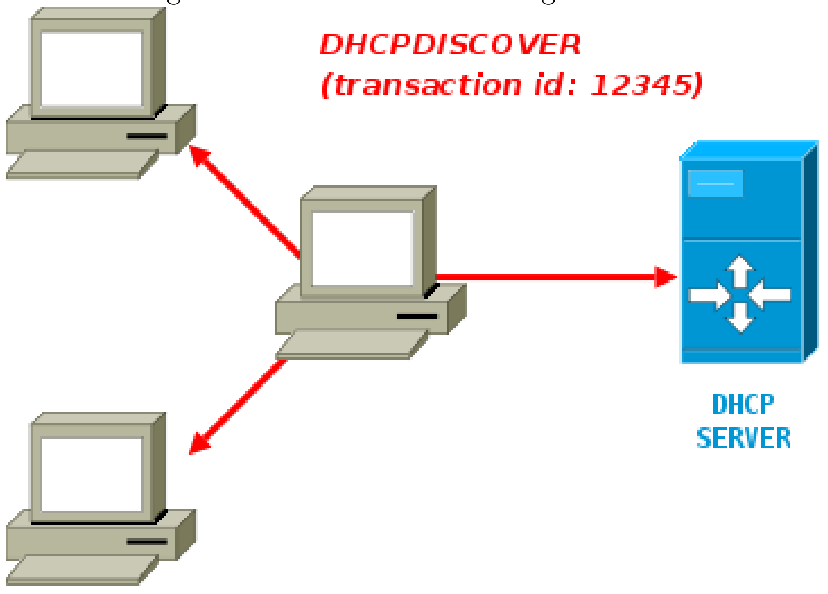


Since Xen assigns domain IDs in incremental order, by looping up to their maximum value and issuing the *grant operation* hypercall it is possible to list the assigned domain IDs (and so know how many domains exist). It is worth mentioning that a *legal* way to enumerate domain IDs is available on Xen, too, but it might get filtered by an XSM module.

The second attack described permits the monitoring of part of the activity of a given remote guest (remote guests are identified by their domain ID). It exploits a set of design decisions inside XenStore, a hierarchical string-based namespace which is at the base of the Xen environment. The attack that is carried on using those design decisions is an improvement of a traditional DHCP [56] spoofing attack. [57]

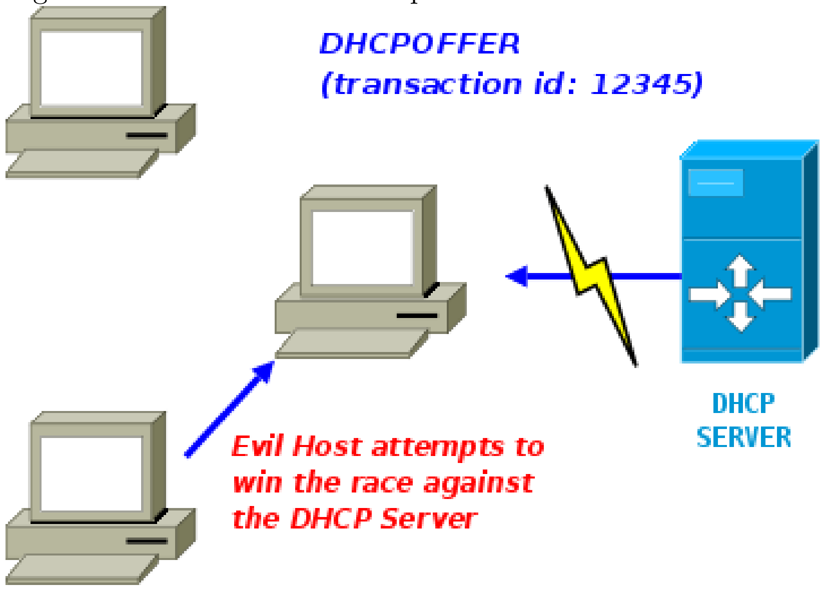
In a normal situation, a given host broadcasts a DHCPDISCOVER, to learn about how to configure its networking stack, as shown in Fig. 4.2 (what IP has been assigned to him, which is the default gateway address, which are the nameservers addresses, etc)

Figure 4.2: A Host Broadcasting a DHCPDISCOVER



All the other hosts on the network receive it, but only the DHCP Server replies, with a DHCPOFFER packet. The 32-bit transaction ID is randomly chosen by the sending host and is used to identify the DHCP session (it will get included in every DHCP packet exchanged). The idea behind **DHCP Spoofing** is to sit on the network, wait for a broadcasted packet and try to win the race against the DHCP Server. If the *evil host* manages to reply before the DHCP Server (Fig. 4.3) it may manipulate the values that the requesting host is going to use to configure its network environment. A classic manipulation is, for the evil host, to send its own address as the address of the default gateway, so that all the outgoing traffic generated by the requesting host will pass through the evil one. Another approach is modifying the nameservers, so that all the DNS [58] requests will go through the evil host, which will be able, in turn, to point the victim to sites under its control.

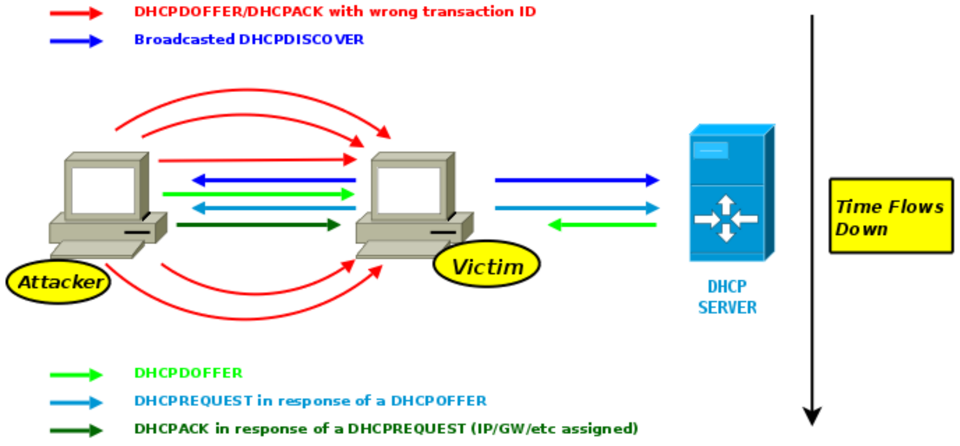
Figure 4.3: An Evil Host Attempts to Win the Race with the DHCP Server



The success of this attack is based, as we said, in winning the race. The attack we present here aims at increasing the chances of success by sending the DHCPOFFER (and DHCPACK) packets *in advance*, as it is shown in

Fig. 4.4

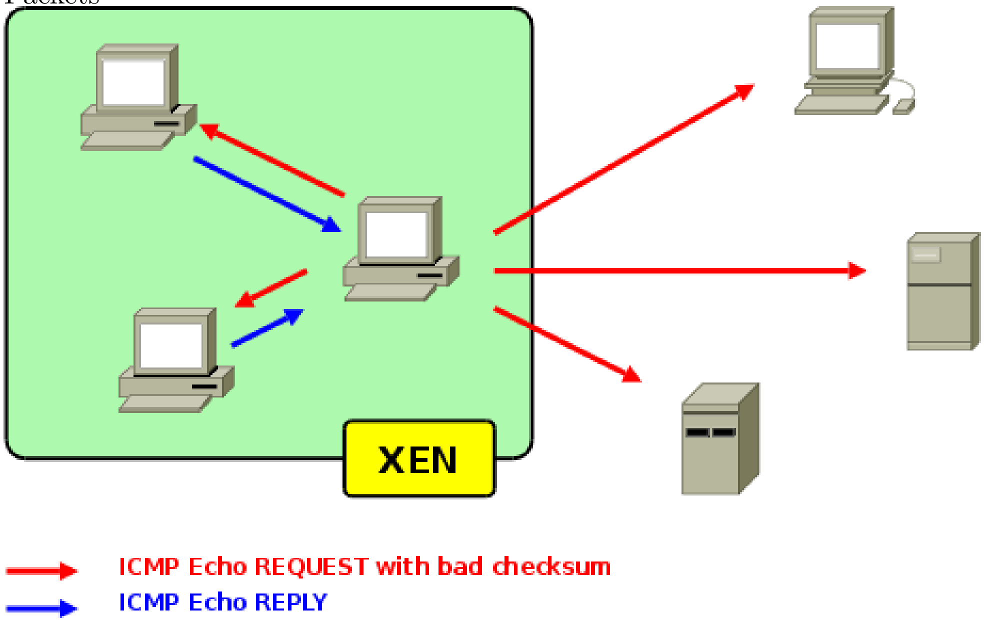
Figure 4.4: The DHCP-Prediction Attack


DHCPDOFFER/DHCPACK with wrong transaction ID
Broadcasted DHCPDISCOVER
Attacker
Victim
DHCP SERVER
Time Flows Down
DHCPDOFFER
DHCPREQUEST in response of a DHCPOFFER
DHCPACK in response of a DHCPREQUEST (IP/GW/etc assigned)

Thanks to a design issue in XenStore we are able to figure out when a domain is booting and so we start flooding the network with DHCPOFFER and DHCPACK packets, predicting the transaction id (whose randomness is based on the MAC address and the local time). As soon as the domain sends a DHCPDISCOVER packet, it will already find the answer. The legal DHCP Server will not even have the chance to reply to the first packet.

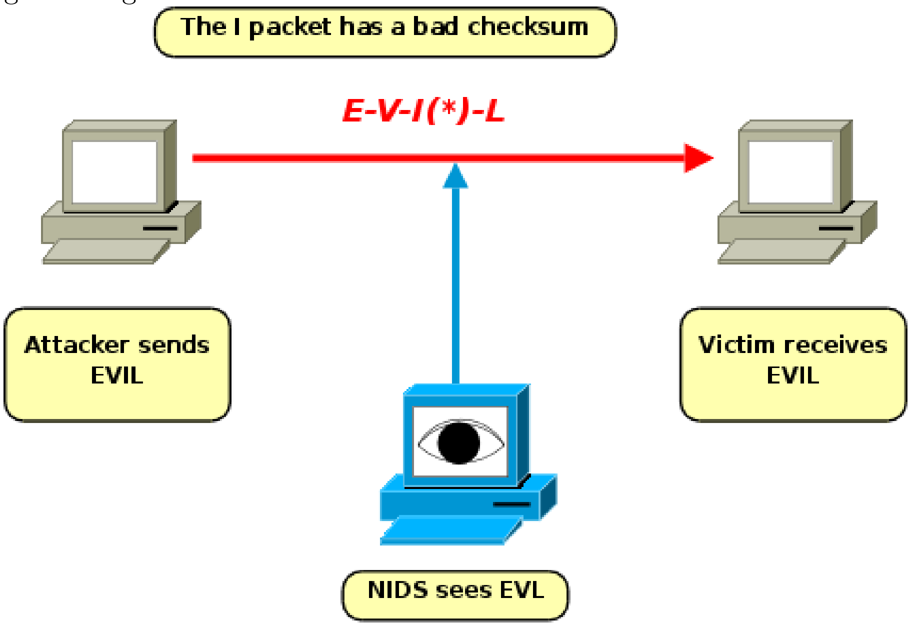
The last attack we present is based on a classic design decision for virtualized environment networks: since the packets *travel* only in memory, the possibility of a corruption is very rare. This permits the virtual hosts to save some CPU cycles by not computing at sending time and not validating at receive time the layer four checksum. [59] Xen gives the possibility of computing or not the checksum at sending time and, by default, avoids doing it for TCP and UDP packets. Thanks to its implementation, a module loaded in the guest can *arbitrarily* play with the checksum of *any* packet, computing it wrong but marking the packet as valid and so getting it *accepted* by another guest. As we show in Fig. 4.5, in a mixed network, with a number of hosts hosted on the same physical machine and a number of hosts on other boxes, this attack can be used to figure out which IPs are associated to *local* hosts.

Figure 4.5: Only Guests On the Same VMM Reply to Bad Chksum'ed Packets



For this attack we use ICMP Echo Request packets, but any other packet that implies an answer will do fine. A second *attack* that we can perform with this feature is carried against NIDS and is based on the widely known concept of *insertion and evasion.* [60] The basic idea is to send packets that a NIDS will ignore, but that the remote host will accept (evasion) or, viceversa, send packets that the NIDS will consider and the remote host will discard (insertion). Since we control both the checksum and what the remote host will do with the packet, we can decide to play evasion or intrusion depending on NIDS configuration: if the NIDS is configured to not take in account bad checksummed packets, we just hide traffic from its analysis as shown in Fig. 4.6, while if it does, we just insert bogus traffic that the remote host will reject, thus escaping from the IDS signature analysis. This trick can be used in conjunction with the DHCP attack to hide the DHCP traffic from particular NIDS configurations or in any other classic way, for example to hide the exploitation payload of a web-based attack.

Figure 4.6: Classic Evasion example: The attacker sends "EVIL", where "I" is a packet with a bad checksum, and the NIDS sees "EVL", thus not generating an alert



Now that we have finished with this quick overview, we move to a detailed description of all the attacks. This overview should be enough, though, to understand at least the basic ideas behind the design of **xenophobia**, our attacking tool, in case the reader would want to skip the technical details.

4.2 Xen building blocks: grant tables

Despite having become a very modern and advanced VMM, Xen is still based on a handful of simple building blocks (hypercalls, grant tables, event channels). [29] Among those building blocks we focus now on the grant tables implementation.
Grant tables are the way Xen implements shared memory and communication between domains. A grant table is an array of `grant_entry`, as detailed in Fig. .

Figure 4.7: struct grant_entry

```
<include/xen/interface/grant_table.h>
struct grant_entry {
    /* GTF_xxx: various type and flag information.  [XEN,GST] */
    uint16_t flags;
    /* The domain being granted foreign privileges. [GST] */
    domid_t  domid;
    /*
     * GTF_permit_access: Frame that @domid is allowed to map
         and access. [GST]
     * GTF_accept_transfer: Frame whose ownership transferred by
         @domid. [XEN]
     */
    uint32_t frame;
};
</>
```

Grant tables allow a domain to perform two different operations, specified by the value of the `flags` member:

map/share a page frame a domain exports a grant entry to another, which in turns maps the page frame inside its address space. Both the domains keep a reference to the page frame inside their address space. (GTF_permit_access)

transfer a page frame the page frame is moved from one domain to another, that is, only the *destination* domain will have, at the end, a reference to the page inside its address space. (GTF_accept_transfer)

The grant table itself is only a way to expose information (usually in conjunction with the Xenstore[1]): all the operations related to it (mapping, transferring, setup) are carried on through a dedicated *hypercall*, shown in Fig. 4.8

Figure 4.8: HYPERVISOR_grant_table_op

```
<include/asm-i386/mach-xen/asm/hypercalls.h>
HYPERVISOR_grant_table_op(
        unsigned int cmd, void *uop, unsigned int count)
{
   return _hypercall3(int, grant_table_op, cmd, uop, count);
}
</>
```

This single hypercall can perform a range of different operations, identified by the `cmd` parameter. The second parameter, `uop`, is an array of structures which contains the specific data relative to the chosen operation and the `count` parameter specifies how many of those are there.
A complete list of all the operations (and associated structs) available can be quickly drawn checking `include/xen/interface/grant_table.h`. We will focus on one of them, `GNNTABOP_map_grant_ref`, while discussing our first attack: *enumerate guest domains on the same box.*

[1]the XenStore implementation (and its weaknesses) will be discussed in the next section

4.2.1 Enumerating Guest Domains via Grant Table Operation Error Codes

The first attack we present is a way to enumerate the number of domains that are hosted on the machine and to identify which are their domain IDs. The vector of the attack is the `GNNTABOP_map_grant_ref` operation, which is used by a domain to map a frame exposed by another domain inside its own address space (Fig.).

Figure 4.9: GNTTABOP_map_grant_ref Operation and Related Struct

```
<include/xen/interface/grant_table.h>
#define GNTTABOP_map_grant_ref         0
struct gnttab_map_grant_ref {
    /* IN parameters. */
    uint64_t host_addr;
    uint32_t flags;              /* GNTMAP_* */
    grant_ref_t ref;
    domid_t  dom;
    /* OUT parameters. */
    int16_t  status;             /* GNTST_* */
    grant_handle_t handle;
    uint64_t dev_bus_addr;
};
typedef struct gnttab_map_grant_ref gnttab_map_grant_ref_t;
DEFINE_XEN_GUEST_HANDLE(gnttab_map_grant_ref_t);
```

The `struct gnttab_map_grant_ref` is used as the second parameter in the aforementioned `HYPERVISOR_grant_table_op` hypercall. As you can see, this struct is used both to pass parameters to the specified operation and store the return values.
The *remote* domain that will be contacted is specified by setting the `dom` parameter to its ID, while the grant reference is identified by the `ref` parameters, which is basically its index inside the remote domain grant table.

The result of the operation is returned inside the `status` member, while `handle` is filled with a reference to the mapped grant entry if the operation was successful. The `status` value is what we are interested in. A list of the possible status values is provided in Fig. 4.10.

Figure 4.10: Grant Table Operations Return Codes

```
<include/xen/interface/grant_table.h>
#define GNTST_okay              (0)  /* Normal return.
                          */
#define GNTST_general_error     (-1) /* General undefined error.
                   */
#define GNTST_bad_domain        (-2) /* Unrecognsed domain id.
                     */
#define GNTST_bad_gntref        (-3) /* Unrecognised or
    inappropriate gntref. */
#define GNTST_bad_handle        (-4) /* Unrecognised or
    inappropriate handle. */
#define GNTST_bad_virt_addr     (-5) /* Inappropriate virtual
    address to map. */
#define GNTST_bad_dev_addr      (-6) /* Inappropriate device
    address to unmap.*/
#define GNTST_no_device_space   (-7) /* Out of space in I/O MMU.
                   */
#define GNTST_permission_denied (-8) /* Not enough privilege for
     operation.  */
#define GNTST_bad_page          (-9) /* Specified page was
    invalid for op.    */
#define GNTST_bad_copy_arg     (-10) /* copy arguments cross page
     boundary.   */
#define GNTST_address_too_big (-11) /* transfer page address too
     large.      */
</>
```

As we can see, there is a specific error which specifies an *unrecognised domain id*. To confirm that we can use this return value to precisely distinguish a failing operation due to the nonexistence of a domain id from one failing (or succeeding) for another reason, we need to take a closer look at

the implementation of the hypercall inside the Xen code.
Once the hypercall is issued, the control is *passed* to the hypervisor. This is just the same as is on modern operating systems (ex. Linux) when a syscall is issued and the control is *passed* from the userland to the kernelland.
In *hypervisor-land*, the hypercall is identified and the correct helper function is dispatched: in our case, the control reaches `do_grant_table_op`, located in `common/grant_table.c` in the main Xen tree, which in turns validates the `cmd` parameters of the hypercall and calls the appropriate function, that is `gnttab_map_grant_ref`.
As it is usually the case in kernel programming, this function is an error-checking wrapper which handles the possibility of executing multiple operations with a single call. For every operation `__gnttab_map_grant_ref` is called.

The two checks shown in 4.11 are the the first two that this function performs on the received `op` parameter. The check at [1] is easy to pass, since we do control `op->flags`. The second check [2] is the one that is of interest. This check fails if the domain is not in the domains list that the hypervisor keeps and the status is set to `GNTST_bad_domain` at [3]. The remaining code of the function is not reported for brevity, but this kind of error is not returned in any other case. By issuing a set of `GNNTABOP_map_grant_ref` with *increasing* domid values it is thus possible to find the other domains hosted on the system and the associated domain ID by checking the return value.
It is important to note that this is not the only way to list domains. A *legal* way to list domains also exists, using XenStore and the
IS_DOMAIN_INTRODUCED request shown in Fig. 4.12

Figure 4.11: Error Checking Inside map_grant_ref

```
<common/grant_table.c>
static void
__gnttab_map_grant_ref(
    struct gnttab_map_grant_ref *op)
{
[...]
int              rc = GNTST_okay;
[...]
if ( unlikely((op->flags & (GNTMAP_device_map|GNTMAP_host_map))
    == 0) ) [1]
{
    gdprintk(XENLOG_INFO, "Bad flags in grant map op (%x).\n",
        op->flags);
    op->status = GNTST_bad_gntref;
    return;
}
if ( unlikely((rd = rcu_lock_domain_by_id(op->dom)) == NULL) )
    [2]
{
    gdprintk(XENLOG_INFO, "Could not find domain %d\n", op->dom)
        ;
    op->status = GNTST_bad_domain; [3]
    return;
}
</>
```

We mentioned our way because it is more *sneaky* and unintended. As we will see later, only a part of the XenStore has been fully secured against leaking information, but that might be fixed (and is likely to be) in future releases.

While the possibility of listing domains and relative domids may sound not that useful at a first glance (especially because, for example, we do not know with which domid are associated the services that we might see on the network), we will see in the following sections how it can be used in order to better perform other attacks. Moreover, we will show how we can manage to map some of the service on a network to the specific domain they are

Figure 4.12: XenStore do_is_domain_introduced

```
<tools/xenstore/xenstored_domain.c>
void do_is_domain_introduced(struct connection *conn, const char
     *domid_str)
{
   int result;
   unsigned int domid;
   if (!domid_str) {
            send_error(conn, EINVAL);
            return;
    }
    domid = atoi(domid_str);
    if (domid == DOMID_SELF)
            result = 1;
    else
            result = (find_domain_by_domid(domid) != NULL);
    send_reply(conn, XS_IS_DOMAIN_INTRODUCED, result ? "T" : "F"
        , 2);
}
</>
```

hosted on.

4.3 Split Drivers, XenStore and XenBus

When we talk about *user activity* we end up calling in cause device drivers. User *interaction* is done through peripherals (keyboard, mouse, etc.) and usually aims at using directly or indirectly some other peripherals (hard disk, network card, sound card, etc.). This section gives a detailed explanation of the Xen device model and shows how we can gather information from it that will lead to successfull attacks.

4.3.1 The Split Driver Model

As we said in the previous chapter, *paravirtualization* is a virtualization approach in which the guest kernel is made aware of the existence of the

virtual machine monitor and cooperates with it. From a practical point of view, this translates to a direct modification of the kernel sources to make it VMM-aware.
While a paravirtualization based monitor might use emulation for its devices, it makes a lot more sense to apply the paravirtualization approach to devices too. This is exactly what happens in Xen, where a set of *generic device interfaces*, one for each common category (block device, network device, ...), is provided to guests.
Those *generic devices* have to be:

fast to justify their use instead of emulated ones. This is not hard to achieve, since memory is usually a very reliable and fast way of communicating, compared to network or hard drives.

simple kernel developers should be able to integrate those devices quickly and easily inside their systems to encourage porting of the operating systems they work on.

The Xen approach is based on the *split driver model* [29]: every generic device has a *frontend*, implemented inside the guest and to which the guest talks, and a *backend* driver, which in turn communicates directly with the hardware device. The backend driver in Xen is usually hosted on domain 0, but projects to have separated non-dom0 driver domains are gaining more and more popularity. The reason is mostly due to security and fault tolerance: since the dom0 is the most privileged and the only fundamental domain in a Xen environment, the more we manage to take away from it, the more we gain in isolation.[2]

[2]funny enough, one of the most interesting *attacks* that we can carry on against XenStore, as we will see in a short, is made possible due to a design decision aimed at supporting domain drivers...

Backend and frontend devices communicate through shared memory. A particular data structure, the *ring buffer*, implemented on top of the grant table mechanism is used. The ring buffer is basically a producer-consumer based data-structure, which uses free running counters to avoid having to wrap back the pointers (for producer and consumer) when the last position of the buffer is reached. The ring buffer implementation can be found inside `include/public/io/ring.h`. *Event channels* are used to implement asynchronous access to devices. They are the hypervisor software-based equivalent of interrupts. During device setup an event channel is allocated and bound between the driver domain and the guest, which then can decide to use it instead of polling continuously.

Now that we roughly know how device drivers work we still need to answer to two important questions to understand how a guest can use them:

- how does a guest discover devices?
- how do front/backend pairs get initialised, remain synchronised and get torn down?

The answer to the first question is *XenStore*, the answer to the second is *Xenbus* which we now describe in some detail.

4.3.2 XenStore

When a machine boots, the operating system needs to be made aware of the physical devices that are available on the system. This is achieved on the x86 architecture by using a set of BIOS service functions while still in real mode and on other architectures, like the UltraSPARC, using the Open-Firmware exported device tree. As it can be imagined, a Xen guest has no way to directly access the BIOS routines, so there must be another way to learn about the available devices. This is the *XenStore*.

The XenStore is a hierarchical namespace whose entries contain strings maintained by the domain 0. We can examine its contents from inside the domain 0 through a set of userland tools that use a local Unix domain socket, as shown in Fig. 4.13.

Figure 4.13: Extract of XenStore tree

```
bender:~# xenstore-ls
tool = ""
 xenstored = ""
vm = ""
 00000000-0000-0000-0000-000000000000 = ""
  on_xend_stop = "ignore"
  shadow_memory = "0"
  uuid = "00000000-0000-0000-0000-000000000000"
  on_reboot = "restart"
[...]
local = ""
 domain = ""
  0 = ""
   [...]
  1 = ""
   vm = "/vm/4ef6cc83-ea3c-1106-2a35-b6cda5ebd44d"
   device = ""
    vbd = ""
     2049 = ""
      virtual-device = "2049"
      device-type = "disk"
      protocol = "x86_32-abi"
      backend-id = "0"
      state = "4"
      backend = "/local/domain/0/backend/vbd/1/2049"
      ring-ref = "8"
      event-channel = "6"
[..]
```

Each entry is identified by a path in the Unix way, for example `/local/domain/1/vm` (Fig. 4.13. Each entry in the XenStore has an associated permissions set. We can get a primer on permissions using the `xenstore-ls -p` command, whose output is shown on Fig. 4.14.

Figure 4.14: Example of XenStore permissions on entries

```
vm =              . . . . . . . . . . . . . . . . . . . . . . . . . (n0)
00000000-0000-0000-0000-000000000000 = "" . . . . . . . . (n0)
 on_xend_stop = "ignore"  . . . . . . . . . . . . . . . . (n0)
 shadow_memory = "0"  . . . . . . . . . . . . . . . . . . (n0)
 uuid = "00000000-0000-0000-0000-000000000000"  . . . . (r0)
```

Permissions are identified by a character and a domid value. The character can be *n* - no perm, *r* - read, *w* - write or *b* - both, while the domid identifies the specific domain to which the permission refers. Permissions are not enforced for domain 0, just as with a traditional Unix system when it comes to the root user accessing the filesystem.

The XenStore is exposed to guest operating systems as a standard driver, with its own ring buffers and event channel, and it is exactly through this mechanism that it can be accessed. Nevertheless, the guest system needs the XenStore at boot time to discover all the other devices, so it must be advertised somehow. To achieve that, Xen communicates the address of a page to the newly created domain which contains the information necessary to locate the XenStore grant references and event channel. This page is known as the **start_info page** and is shown on Fig. 4.15.

As can be seen, a lot of boot-relevant information is exposed through the

Figure 4.15: How XenStore is Mapped Thanks to start_info

```
<linux/include/xen/interface/xen.h>
struct start_info {
  /* THE FOLLOWING ARE FILLED IN BOTH ON INITIAL BOOT AND ON
     RESUME.     */
  char magic[32];              /* "xen-<version>-<platform>".
                  */
  unsigned long nr_pages;      /* Total pages allocated to this
     domain.  */
  unsigned long shared_info;   /* MACHINE address of shared info
     struct. */
  uint32_t flags;              /* SIF_xxx flags.
                              */
  xen_pfn_t store_mfn;         /* MACHINE page number of shared
     page.    */
  uint32_t store_evtchn;       /* Event channel for store
     communication. */
 [...]
}
</>
<linux/arch/i386/kernel/head-xen.S>
 .org VIRT_ENTRY_OFFSET
 ENTRY(startup_32)
        movl %esi, xen_start_info
        cld
</>
<linux/arch/i386/kernel/setup-xen.c>
start_info_t *xen_start_info;
EXPORT_SYMBOL(xen_start_info);
</>
```

start_info page. The way the kernel gets notified of the address of this page is architecture specific, in the x86 case, the address of it is passed through a register upon entering the kernel.

One of the most interesting features of the XenStore is the implementation of *watches* on paths. A domain can register a callback function that will be called any time the *watched path* changes. The *watch* implementation will be described in more detail in the following sections since it is at the base, of both of the XenBus and the DHCP attack.

4.3.3 XenBus

The last thing we have to describe is how the setup, synchronisation and shutdown of devices is performed. This is achieved via a pseudo-protocol built on top of XenStore which is known as *XenBus*.[3]

To see the relationship between the watch mechanism and the XenBus protocol we describe how a frontend and a backend device get connected. A `struct xenbus_device` (Fig. 4.16) is associated to each device that uses the XenBus protocol.[4]

Figure 4.16: struct xenbus_device

```
<linux/include/xen/xenbus.h>
struct xenbus_device {
   const char *devicetype;
   const char *nodename;
   const char *otherend;
   int otherend_id;
   struct xenbus_watch otherend_watch;
   struct device dev;
   enum xenbus_state state;
   struct completion down;
};
</>
```

Information about the device itself (frontend or backend) and the *otherend* (respectively, backend or frontend) is kept inside this struct. More-

[3]The XenBus term is used in different scenarios: on Linux it is the whole interface to XenStore, while when talking about generic device drivers it refers to the synchronisation protocol built on top of XenStore. While this distinction is not fundamental for the discussion, it is worth mentioning.

[4]The console driver and the XenStore do not use the XenBus protocol, but instead get initialized at boot time using the information provided by the start_info page.

over, the state of the device is tracked, and exposed via XenStore too, and the watch on the otherend state is also stored there. The `state` value and the associated `watch` are the heart of the XenBus protocol. The device can be in one of seven states, ranging from `XenbusStateInitialising` to `XenbusStateClosed` and `XenbusStateReconfigured`.

During device initialization and tear down the frontend and backend pass from state to state and use the watch mechanism to synchronize. During the normal operation of the device, the *state* value will be `XenbusStateConnected`. We can see how the watch is setup and what callback function is called by looking at `xenbus_probe.c`, reported in Fig. 4.17 and in Fig. 4.18

Figure 4.17: otherend_changed Watch Callback

```
<linux/drivers/xen/xenbus/xenbus_probe.c>
static void otherend_changed(struct xenbus_watch *watch,
                             const char **vec, unsigned int len)
{
   struct xenbus_device *dev =
      container_of(watch, struct xenbus_device, otherend_watch);
   struct xenbus_driver *drv = to_xenbus_driver(dev->dev.driver)
      ;
   enum xenbus_state state;

   /* Protect us against watches firing on old details when the
      otherend
      details change, say immediately after a resume. */
   if (!dev->otherend ||
      strncmp(dev->otherend, vec[XS_WATCH_PATH],
         strlen(dev->otherend))) {
      DPRINTK("Ignoring watch at %s", vec[XS_WATCH_PATH]);
      return;
   }
   state = xenbus_read_driver_state(dev->otherend);
   [...]
   if (drv->otherend_changed)
      drv->otherend_changed(dev, state);
}
```

Figure 4.18: How otherend_watch Callback and Related Watch is Setup

```
static int watch_otherend(struct xenbus_device *dev)
{
        return xenbus_watch_path2(dev, dev->otherend, "state",
                                  &dev->otherend_watch,
                                      otherend_changed);
}

<note: error checking omitted for simplicity>
int xenbus_dev_probe(struct device *_dev)
{
   struct xenbus_device *dev = to_xenbus_device(_dev);
   struct xenbus_driver *drv = to_xenbus_driver(_dev->driver);
   const struct xenbus_device_id *id;
   int err;
   if (!drv->probe) {
                err = -ENODEV;
                goto fail;
   }
  id = match_device(drv->ids, dev);
  [...]
  err = talk_to_otherend(dev);
  err = drv->probe(dev, id);
  err = watch_otherend(dev);
}
</>
```

For each device during the *device discovery* step, `xenbus_dev_probe` is called. This function checks if the device is available (if there is an available *match*) and tries to *talk* to it. If the device is available, `watch_otherend` is called to setup a watch on the remote `state` entry inside XenStore (`dev->otherend` keeps the path in XenStore of the remote backend/frontend). `otherend_changed` is the callback function: a single callback function is provided for both frontend and backend case since the *driver specific* otherend function will be called at the end of it by `drv->otherend`. During the setup phase, the backend device moves from Initializing, then to Wait and in the end to Initialised. Everytime it does a transition, the `state` value inside XenStore is changed accordingly and the watch fires. The frontend callback function will check this value and only when it is set to Initialized it will start the connection process. It should be clear now powerful the watch mechanism is and how synchronisation is achieved through this mechanism.

4.3.4 Exploiting the XenStore/watch design: the DHCP attack

The XenStore has a central role in the whole Xen environment and is thus a very interesting target, especially when the aim is profiling the activity of a given guest. The first attack that one might attempt, looking at the pseudo-filesystem design, is, once again, based on error checking. A quick look to XenStore code shows that it can return both -EACCES (permission denied on a given entry) and -ENOENT (the given entry does not exist). Unfortunately, a closer look at the code reveals that this kind of attack is prevented by the functions showed in Fig. 4.19

The comments pretty much say it all. You are allowed to read/write a node (given that you have correct permissions on it) even if it is inside a path you do not have permissions over. Thanks to this design, for example, the frontend inside a domU can read the state value inside the backend entry of a dom0 (or a driver domain).

The same permissions mechanism is used for every kind of xenstore function that aims to access a given path. The *watch* implementation uses it as well, but it has a *design issue*, shown in Fig. 4.20.

Since EACCES is allowed, a given guest is allowed to set a watch on *any entry* inside the XenStore. This is a major issue, because, as we saw before,

Figure 4.19: XenStore Entry Permissions Checking Code

```
<tools/xenstore/xenstored_core.c>
static enum xs_perm_type ask_parents(struct connection *conn,
    const char *name)
{
   struct node *node;
   do {
      name = get_parent(name);
      node = read_node(conn, name);
      if (node)
        break;
   } while (!streq(name, "/"));
   /* No permission at root? We're in trouble. */
   if (!node)
     corrupt(conn, "No permissions file at root");
   return perm_for_conn(conn, node->perms, node->num_perms);
}
/* We have a weird permissions system.  You can allow someone
    into a
 * specific node without allowing it in the parents.  If it's
    going to
 * fail, however, we don't want the errno to indicate any
    information
 * about the node. */
static int errno_from_parents(struct connection *conn, const
    char *node,
                              int errnum, enum xs_perm_type perm
                                 )
{
   /* We always tell them about memory failures. */
   if (errnum == ENOMEM)
      return errnum;
   if (ask_parents(conn, node) & perm)
      return errnum;
   return EACCES;
}
</>
```

the XenStore is heavily used by all guests. This approach can not be used to bruteforce XenStore entries, though, because ENOENT is also ignored at [1]. Setting a watch on nonexistent entry is, in fact, perfectly legal and the callback will fire only if that entry is created or if some entry in the path

Figure 4.20: Error Checking on Watch Setup

```
static void add_event(struct connection *conn,
                      struct watch *watch,
                      const char *name)
{
   /* Data to send (node\0token\0). */
   unsigned int len;
   char *data;
   if (!check_event_node(name)) {
   /* Can this conn load node, or see that it doesn't exist? */
      struct node *node = get_node(conn, name, XS_PERM_READ);
      /*
       * XXX We allow EACCES here because otherwise a non-dom0
       * backend driver cannot watch for disappearance of a
          frontend
       * xenstore directory. When the directory disappears, we
       * revert to permissions of the parent directory for that
          path,
       * which will typically disallow access for the backend.
       * But this breaks device-channel teardown!
       * Really we should fix this better...
       */
      if (!node && errno != ENOENT && errno != EACCES) [1]
         return;
   }
   [...]
</>
```

that leads to it changes.

This is quite interesting, because we may set a watch on an existing domain-id entry (some paths inside XenStore are totally predictable given you know the domid and we know it thanks to our grant table bruteforcing approach) and monitor part of the activity of the domain. But, which kind of activity ?
To answer to this question we need to see what entries inside XenStore change during the lifetime of a guest. Those are:

- **Frontend/backend state**: We can track the setup of an interface
- **Memory/target**: This value is used by the balloon driver to keep in synch with memory extension and shrinking
- **Any domain specific value**: Setting a watch on the main domain directory will raise a watch event when that entry is deleted, that is, when the specific domain is destroyed.
- **third party value** If a third party project used the XenStore to coordinate different guest domains, we can track part of its activity too.

So far, so good. There is though a hard limit on the number of watches, specified by the `quota_nb_watch_per_domain` inside `xenstored_core.c` and set to 128. That means that no more than 128 watches can be active at the same time. While this is a large number, one might need a larger number to track the activity of many different domains, especially if numerous watches on domain specific and/or nonexistent entries have to be used to track the bootup and shutdown of specific domids.
Luckly the Xen Hypervisor provides two *pseudo-watches*:

@introduceDomain This watch fires every time a domain is introduced (that is, the INTRODUCE operation is executed over the XenStore). Practically, this translates in the callback being called right before a domain boots.

@releaseDomain This watch fires every time a domain crashes or shuts down. Moreover it fires on the RELEASE operation linked to domain destruction. Practically, this translates in the callback being called *twice* every time a domain reboots.

The good news is that the same permission (non)checking is carried out for those two pseudo-watches and so any domainU can track precisely when any other domain *boots* or *shuts down*. How can we use that ?

The DHCP protocol [58] is used to dynamically allocate network configuration details (IP address, nameserver, gateway) to domains. A DHCP server runs on a host in the same local network as the requesting host and listens for requests. Typically, every time a system boots, it broadcasts a request, the DHCP server sees that request and it just replies with a free IP address and the other configuration details.

DHCP spoofing attacks are not a new topic at all [57]. The basic idea behind them is to start a *malicious* DHCP server and just wait for broadcast DHCP requests. Every time one of those is seen, the malicious DHCP server tries to win the race against the official one, by sending an answer to the requesting host. The reason why this may work quite well is that the malicious DHCP server does not care about checking the free list of IP addresses (the attacker specifies manually which IP address to use) and so can be a little quicker than the legal host.

A standard DHCP spoofing attack has two drawbacks though:

- There is no way to know when a given system will boot, so the rogue DHCP server has to run continuously. It is therefore exposed to sysadmin scans on the network.
- The rogue DHCP server may not win the race. Especially in a VMM environment, where there is a time-sharing scheduling among different guests, our malicious DHCP server might get to the CPU too late.

It should be straightforward to see how our XenStore/watch *trick* can improve the efficiency of a DHCP spoofing attack solving the first drawback: we keep our malicious DHCP server shut down and we setup a @introduceDomain watch. Every time it fires and calls our callback function, we start the rogue DHCP server and we set an approximately 30 second timer. If no DHCP request arrives during this time-window (which should be enough for a system to boot and launch the dhcp client), we shut the DHCP server down again and wait for the next time our watch will fire.

Thanks to this approach, our malicious DHCP server is almost always down and will not be visible to an occasional scan by a sysadmin. Is that all ? Is there anything we can do against the second drawback and *cheat* a bit in the race ?

To answer to this question we need to take a closer look at the DHCP protocol and the most common dhcp client implementations. A normal DHCP session consists of 4 messages exchanged between the server and the client.

- The client broadcasts a **DHCPDISCOVER** packet and generates a transaction ID to identify the answer to its packet.
- The server replies with a **DHCPOFFER** packet, which carries the same transaction ID and which contains the MAC address of the destination client and an offered IP address. Inside this packet a list of DHCP options for the client is availiable, e.g. the gateway address, the netmask, the IP address of the DHCP server, the lease time (after how much time should the client broadcast a new request), the nameserver, etc.

- The client replies with a **DHCPREQUEST** packet, which carries the same transaction ID, which asks for a specific IP address (usually the one offered by the server inside the DHCPOFFER request)
- The server acknowledges the request with a **DHCPACK** packet (or refuses it with a **DHCPNACK**). The transaction ID in the packet is always the same as earlier.

This is the standard sequence of packets. Other sequences are possible, for example, when renovating lease, a client simply sends a **DHCPREQUEST** packet and waits for a **DHCPACK** from the server.

Aside from two pieces of information, the *MAC address* of the client and the *transaction ID*, all the other contents of the packets are predictable. That means that, if we find a way to predict the transaction ID and the MAC, we might start flooding the network for 20 seconds or so and automatically win any possible race!

Knowing the MAC address, depends on the Xen configuration. The MAC address of the virtual network interface of a guest can be randomly generated by Xen at runtime or can be statically specified inside the Xen configuration file,

```
vif=[ 'mac=00:16:3e:01:01:01,bridge=mybridge' ]
```

This second option is mandatory if we want a specific address to be assigned, by the DHCP server, to a specific host. The MAC address is the only way the DHCP server has to identify it the specific host.

The 32-bit transaction ID is a little more tricky, but can be predicted as well. Its *randomness* depends on two things:

Figure 4.21: dhclient Transaction ID Computation

```
<dhcp3-3.1.1/client/dhclient.c>
/* Make up a seed for the random number generator from current
   time plus the sum of the last four bytes of each
   interface's hardware address interpreted as an integer.
   Not much entropy, but we're booting, so we're not likely to
   find anything better. */
   seed = 0;
   for (ip = interfaces; ip; ip = ip -> next) {
    int junk;
    memcpy (&junk,
    &ip -> hw_address.hbuf [ip -> hw_address.hlen -
       sizeof seed], sizeof seed);
    seed += junk;
    }
    srandom (seed + cur_time);
</>
```

- The MAC address of the machine.
- The time value (the random number generator seed is initialized with the equivalent of `srandom((time(NULL))+mac_based_value)` at execution time).

As we can see from the dhclient.c code in Fig. 4.21, one of the most common dhcp clients available on linux and the default one on Debian and Ubuntu, the last four bytes of the mac address associated to each interface are used. In this attack we assume that the guest is not *virtually multihomed*, that is, that only one virtual interface is created inside it.

Guessing the time value is not a hard task, in fact, by design all the guests share the same clock value (an attempt to change the clock in a single guest will fail). Moreover, as we can see from the man page shown in Fig. 4.22, the `time` system call returns the number of seconds since the

Figure 4.22: Extract of 'time(2)' Manual Page

```
 time_t time(time_t *t);

DESCRIPTION
    time() returns the time since the Epoch (00:00:00 UTC,
         January 1,1970), measured in seconds.
```

Epoch, which in turn means that the seed value will change monotonically each second.

This is more than enough to attempt a (successful) bruteforce attack. If Xen is configured with static MACs for guests, a malicious code could stay in the background and collect all the MAC addresses associated with the running domains. Everytime a @releaseDomain watch fires, the code could quickly figure out which is the missing MAC address, wait for a subsequent @introduceDomain and start the attack, looping every second, attempting to hit the right transaction ID. The detailed implementation of the attack will be covered in the next chapter, along with a working example.

4.4 Xen Paravirtualized Network Driver

Network communication is likely to be one of the most intensive activities that a given guest will perfom. The Xen paravirtualized network driver is a classic example of a *split driver*: on the domU side, the *netfront* driver is placed at the lower level of the OS network layer code and links to the dom0[5] *netback* part, which is the only part of the driver directly connected,

[5]We consider here the *standard* (and nowadays fairly common) architecture which sees the physical network driver hosted on the dom0. Keep in mind that an architecture with a Domain Driver different from the dom0 might be in place. For the rest of the discussion and especially for the attacks presented, does not matter which configuration is in place:

through the OS specific network layer code, with a physical device.

This design has a couple of advantages:

- **Support for a wide range of physical devices**: The netback driver maps into the hosting operating system network layer and thus *indirectly* supports all the device drivers that the OS supports.

- **In-memory communication between guests**: The netback driver does not have to move the packet down to the physical layer if the communication is between guests, memory can be just transferred from the sender to the receiver. This approach is:

  Safer : Memory errors are way more rare than network transmission errors

  Faster : *Memory passing* can be optimised: by default Xen uses today a copy mechanism in which the Hypervisor copies the data from the sender buffer to the receiver buffer. Since the Hypervisor has a full view of the whole memory address space this operation involves less TLB flushing than moving a page from the address space of the sender to the receiver space (this approach, called *flip*, is also available in Xen). Some optimisation is also possible on the *network side*, most notably, there is no MTU (Maximum Transfer Unit) limit and the computation of the checksum can be avoided (memory error can be considered sufficiently unlikely).

Concentrating on the code, we see that two *ring buffers* are allocated between netfront and netback (one for sending and one for receiving packets) and an event channel is setup to notify the other end that some data is

the attack works on both the configurations.

available to be parsed.[6] In case of high traffic load, the event notification can be suppressed and polling can be used instead, to improve performances.

Data is exchanged between netfront and netback in terms of *sk_buff* [61] structures, the basic structure of the linux networking code.

4.4.1 Network Level Tricks: the Bad Checksum Story

In a traditional network environment, packets travel from host to host to reach their destination. On this path, packets pass through different physical connections and get parsed and analysed by different systems. Errors, both in software and in hardware, are possible. Such errors might corrupt the content of the packet and so deliver wrong information. The best thing an operating system can do in an error situation is to discard the packet and, depending on the *reliability* of the transport protocol, eventually trigger a re-transmission. To identify potential transmission errors, the protocols involved in the communication typically add a *checksum* **??** to their headers. A checksum is just some sort of hash value that is computed depending on the contents of the packet/header. The common checksum function that is used in the network case is the one-complements of the binary sum of all the field-values taken in account. [61]

One of the optimisations that can be done on guest-to-guest communication is known as *checksum offloading*. Modern operating systems already have code to handle this kind of situation, because some recent network cards implement checksum computation and validation in hardware. Moreover, almost all operating systems also implement a local network loop device, a virtual network device usually bound to 127.0.0.x. In such a device all the

[6]the event channel mimics the behaviour of a NIC (Network Interface Card), which raises an interrupt to signal to the CPU/OS that there is some data to handle.

Figure 4.23: Linux Network Devices Checksum Related Flags

```
<linux/include/netdevice.h>
#define NETIF_F_IP_CSUM  2  /* Can checksum only TCP/UDP over
    IPv4.*/
#define NETIF_F_NO_CSUM  4  /* Does not require checksum. F.e.
    loopback.*/
#define NETIF_F_HW_CSUM  8  /* Can checksum all the packets.*/
</>
```

communications happens in memory, so errors can be considered as absent[7], and so the operating system can save cycles of CPU by simply ignoring (not computing, not checking) the checksum.

The Linux kernel associates a *checksum capability flag* to every network device, so that the other functions involved in transmitting and receiving packets know what to do when it comes to checksum computation. The available flags are listed in Fig. 4.23

It is important to note that these flags refer to the checksum computed on layer four protocols (TCP, UDP, etc) and that this checksum covers both the header and the data. Depending on the flag associated with the network device that gets used at sending or receiving time, the kernel behaves differently. As we already said, the basic structure used through all the linux networking code is the `struct sk_buff`. Two fields of this struct refer to the checksum value and computation: `sk_buff->csum` and `sk_buff->ip_summed`. The meaning of those two members varies depending on whether the packet is being sent or received:

At receiving time , the `csum` member holds the layer four checksum (or

[7]Memory corruption is not impossible, but if it is happening the operating system is likely to have many more problems than just wrong packets on the localhost...

whatever is believed to be the layer four checksum) while the `ip_summed` field specifies what should be done with this value. This flag can have three possible values:

- **CHECKSUM_NONE**: The `csum` value is invalid and has to recomputed in software.
- **CHECKSUM_HW**: The `csum` value has been computed by the hardware, but the software needs to compute and add the pseudo-header checksum to this value and then verify if it is valid.
- **CHECKSUM_UNNECESSARY**: The software does not have to verify the `csum` value, all the validation has been done in hardware.[8]

At sending time , the `csum` member points to the checksum field in the protocol header, so that the hardware device knows where to store it if it has to compute and insert it. The `ip_summed` value specifies what the device has to do:

- **CHECKSUM_NONE**: Everything has been done in software, so the device need not be involved.
- **CHECKSUM_HW**: The device has to compute the checksum on the header and the payload.

The choice made by Xen network device developers is to flag the related virtual device as *NETIF_F_IP_CSUM* capable. That means that the kernel *will offload* the computation and validation of the checksum of TCP/IP and UDP/IP packets. This feature can be configured on Xen virtual network

[8]This is the case, also, for in-memory devices like the loopback one.

Figure 4.24: sk_buff Members Added by Xen

```
<linux/include/linux/skbuff.h>
#ifndef CONFIG_XEN
                ipvs_property:1;
#else
                ipvs_property:1,
                proto_data_valid:1,
                proto_csum_blank:1;
#endif
</>
```

devices by setting to 1 a per-device option called *feature-no-csum-offload.* By default the value of this option is one. In conjunction with that, Xen extends the `sk_buff` struct with a couple of members, shown in Fig. 4.24

`proto_data_valid` specifies that the protcol data was validated since arriving at localhost. The `proto_csum_blank` specifies that the checksum is left blank and, if the packet has to leave the localhost, it has to be computed. Those two parameters, along with the two described above (`ip_summed` and cksum), are used by Xen to identify and *optimise* (via checksum offload) guest-to-guest communication. When a guest wants to send a packet, the kernel constructs the `sk_buff` struct associated with the packet normally up to the transmit moment, honouring the *NETIF_F_IP_CSUM* flag. At transmit time, the sk_buff is placed inside one (or more) of the pages shared between netfront and netback and a *notification* carrying the grant reference is placed in the ring buffer with a set of flags describing the packet. Two flags are related to checksum computation and get set depending on sk_buff members values, as shown in Fig. 4.25

An event is, eventually, generated and the netback code kicks in, copy-

Figure 4.25: Checksum Related Flags at Guest Sending Time

```
<linux/drivers/xen/netfront.c>
   tx->flags = 0;
 [...]
   if (skb->ip_summed == CHECKSUM_HW) /* local packet? */
      tx->flags |= NETTXF_csum_blank | NETTXF_data_validated;
#ifdef CONFIG_XEN
   if (skb->proto_data_valid) /* remote but checksummed? */
      tx->flags |= NETTXF_data_validated;
#endif
</>
```

Figure 4.26: Checksum Handling Inside Netback When Netfront Transmits

```
<linux/drivers/xen/netback.c>
      /*
       * Old frontends do not assert data_validated but we
       * can infer it from csum_blank so test both flags.
       */
       if (txp->flags & (NETTXF_data_validated|NETTXF_csum_blank
           )) {
          skb->ip_summed = CHECKSUM_UNNECESSARY;
          skb->proto_data_valid = 1;
       } else {
          skb->ip_summed = CHECKSUM_NONE;
          skb->proto_data_valid = 0;
       }
</>
```

ing in the struct and deciding what to do. If the destination is a local guest, the `sk_buff` is then put on the shared pages of the netfront driver of the destination host, otherwise the struct is queued for transmission on the dom0 network interface. It is interesting to see how the netback behaves in respect to checksum computation in the snippet of code reported in Fig. 4.26

Two things are important to notice:

- `skb` is the same sk_buff struct that was passed in by netfront

- If `NETTXF_data_validated` or `NETTXF_csum_blank` is set, the packet checksum is not checked

Now, this is what happens when checksum offloading is used for inter-domain communication. But since we control the guest kernel, we can arbitrarily decide to set as *validated* any given packet, just by playing with *flags* and textitskb values. Why would we do so ?

There are at least two interesting attacks that we can carry on abusing this possibility:

Host mapping : We can identify systems that are hosted on the same virtual machine host as we are. In fact we can send bad-checksummed packets that are supposed to generate a response from the destination host. Systems on the same virtual machine system will ignore the bad checksum and reply to the packet, while systems that are on the same network but physically separated will drop the packet. Since we can play with any kind of packet, we can use for example an ICMP Echo Request.

NIDS evasion and insertion : Insertion and evasion attacks are well known. They were first theorised by Ptacek and Newsham in 1998 [60]. The idea is to make a NIDS (Network Intrusion Detection System) ignore packets that the remote system will accept (insertion) or evaluate packets that the remote system will ignore (evasion). In the first case it is possible to perform attacks (e.g. a scan of the remote system) by simply using packets that the NIDS will ignore, in the second case it is possible to fool NIDS recognition patterns by mixing *bad* packets with *good* ones.

The second attack is particularly interesting, because it is virtually unstoppable with the current Xen design using a single NIDS: since we control how the netfront will behave we can *decide* if the bad checksummed packet will be accepted by the remote host. If the NIDS is configured to keep track of bad checksummed packet and evaluate them in its pattern matching functions, we can simply mark the skbuff payload as non-validated and the remote guest host will check the checksum and drop the packet. If, instead, the NIDS is configured to not consider bad checksum packets, we can generate *stealth* traffic by simply marking the skb data as validated and set a bad checksum in the header.

We have described the theory behind our attacks against other Xen domUs. It is now time to move to the description of the tool that we developed to perform them: *xenophobia.*

Chapter 5

The Code Implementation: xenophobia

This chapter describes how the attacks have been implemented and glued together in a single tool, capable of automating their execution. The tool is divided in three parts:

- A loadable kernel module : **xenophobia.ko**
- A userland daemon : **xenophobiad**
- A userland user text interface : **xphb**

The libraries used to code the tool are:

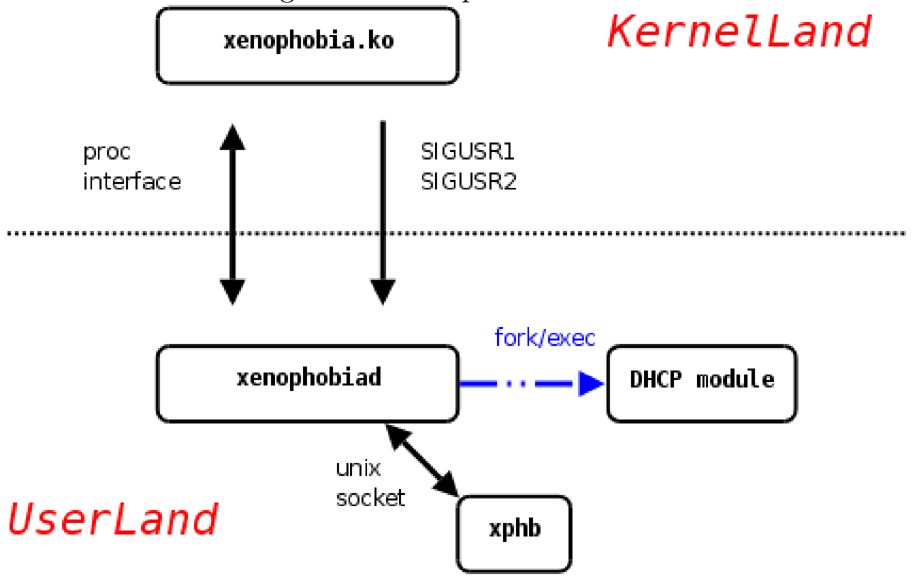
- **libnet** Is a packet creation, manipulation and injection tool [62] originally written by Mike Shiffman.
- **libpcap** Is a packet-capturing framework used by the widely known `tcpdump` linux application. [63]

- **Berkeley DB** is an embeddable database which does not require SQL. It is now maintained and released by Oracle.

5.1 xenophobia architecture

Xenophobia is composed of three main parts and an helper module, which implements the dhcp attack. Its architecture is shown on Fig. 5.1

Figure 5.1: xenophobia Architecture



xenophobia.ko is a Linux Loadable Kernel Module (LKM). Since XenStore must be contacted by a guest from kernel land, a kernel module is mandatory. xenophobia.ko implements the setup of the two watches (*initDomain* and *releaseDomain*) and the hook of the network transmitting function of the netfront driver, so that the checksum games can be played. Moreover, xenophobia.ko communicates with userland in two ways:

Signal passing: Every time a watch fires in kernel land (that is, a domain boots or gets destroyed), the kernel module needs to notify the daemon about that, so that the logic of the attacks can be implemented in userland. This is accomplished using *signals*. `SIGUSR1` and `SIGUSR2` are, respectively, associated to the init and the release of a domain.

/proc interface: a set of proc entries is created so that information from kernel land can be retrieved and the behaviour of the module itself can be configured from userland. The implemented entries are all into **/proc/xenophobia** and are:

- **domains**: Keeps a list of the assigned and alive domaind IDs (obtained through the grant table trick)
- **cksum**: Configures the use of the kernel modification of the checksum of all the packets generated from a process which has the "`xphb`" string in its name (useful to use the insertion/evasion checksum based attack with third-party code, like nmap [64], that has not to be recompiled or modified, but just renamed) and if the packets must be marked as *valid*, that is, if the remote host will check or not the checksum.
- **pid**: Keeps the pid of the running `xenophobiad` daemon. The userland daemon sets this value so that the kernel module knows where to send the signals.

xenophobiad is the userland daemon which implements most of the logic and the automation of the attacks. It can be run in two different modes: *active*, which attempts to perform the dhcp spoofing attack every time there might be the possibility, and *passive*, which simply sits in and monitors the evolution of the network, to collect information about running guests. The active mode is the default one and is controlled, inside the code, by the `do_dhcp` variable.

Whenever the daemon is launched, it saves its pid in the /textit/proc/xenophobia/pid entry and it executes a *map* of the local network to discover which IP have the domains hosted on the same virtual machine. This part is implemented sending *ICMP echo request* messages with a bad checksum that will be received and parsed correctly only by *local*[1] systems. The MAC address and the IP address of the host (together with other information, all kept inside the `struct host_entry`) are stored in the DB, keyed by the IP address.

Before starting to loop forever, the daemon sets up handlers for:

- `SIGUSR1` to handle any domain init watch fired in kernel land. The handling function is called `initdomain_handler`. When this watch fires, depending on the settings and the state of the network, the function starts the DHCP attack or simply executes a new mapping of the network.

- `SIGUSR2` to handle any domain release watch fired in kernel land. The handling function is called `releasedomain_handler`. When this watch fires, the function starts probing the recorded domains, attempting to quickly find the one which is missing, so that, in the future, the DHCP attack can be performed. Probing is once again based on *ICMP echo request* packets.

- `SIGCHLD` to handle the exit of any child process. The handling function is called `sig_chld`. The DHCP attack is implemented in a sep-

[1]We use the term *local* here to indicate systems hosted on the same physical Virtual Machine

arate module which gets executed by the daemon through the classic *fork()/exec()* sequence. The handler kicks in when the children exits and cleanly handles this situation, to avoid zombie processes. Right after cleaning resources, it starts a new mapping of the network, since we do not know *which* domain is booting, we might have started an unsuccessful attack against a new host.

At the end of the initialisation phase, the daemon sets up a local *Unix socket* that is used by the other userland tool, **xphb**, to communicate with the daemon and retrieve information or configure its behaviour, as shown in Fig. 5.2

Figure 5.2: xphb Commands

```
bender:/root/xenophobia# ./xphb
Usage : ./xphb <command> [param]
Commands :
        help            shows this output
        dhcp [on|off]   activates/deactivates dhcp attack
        info            gets detailed info about active domains
        close           shuts down xenophobiad daemon
bender:/root/xenophobia#
```

The *dhcp* command is used to switch the daemon from *passive* mode to *active* mode and viceversa at runtime.

The *dhcp module* is contained in the source code `xb_dhcp_module.c` and is executed from inside xenophobiad with the function `create_dhcp`, which expects the name of the module as a parameter. By default, it is `xb_dhcp`. The dhcp module gets two parameters from the command line (and a bunch of others compiled in from the `dhcpmodule.h` include file):

- The **MAC Address** of the target (-m switch).
- The **IP Address** that you want to assign to the target (-d switch).

The first thing the module does is to prepare the DHCPOFFER and DHCPACK payload that will be used during the attack. After that, it calculates the *candidate transaction id* for the DHCP session, using the current time and the MAC as the seed for the `srandom()` function.

The module then starts to loop for a given amount of seconds, defined by the `ROUNDS` value in `dhcpmodule.h`, flooding the network with DHCPOFFER and DHCPACK packets. If the booting guest is the *expected* one (that is, the one with the correct MAC address) and performs a DHCPREQUEST, it will find the DHCPOFFER and DHCPACK packets already there and the DHCP spoof attack will succeed.

To increase the chances of success, the *previous* value of the transaction id is sent together with the new computed one every time the *time* moves forward of one second. Moreover, since, depending on the client and/or on the fact that a leased/recorded value is already saved, the dhcp client codes may perform one or two `random()` calls before setting the transaction id, packets containing both the values are sent.

One may consider to use the *checksum trick* in conjunction with this attack, to evade particular NIDS configurations. In case this is the aim, it is just a matter of changing the name of the module from *xb_dhcp* to *xphb_dhcp* and change the name passed as a parameter to the `create_dhcp` function (the kernel module will take care of the rest).

The xenophobia code is available upon request under GPL Licence. [65]

Chapter 6

Conclusions and Future Work

6.1 Conclusions

Virtualization technology has a lot of appeal for universities, hosting providers and, more generally, service providers, because it permits them to get rid of a lot of large and, sometimes, cumbersome physical hardware, while still providing multiple separate environments for students and customers. The Xen project has been estabilishing as one of the leading projects in the virtualization field. We have presented **xenophobia**, a tool that implements and automates a set of attacks against Xen-based environments. These attacks can be carried out from inside an unprivileged domain, targeting other unprivileged domains, without requiring the attacker to raise its own privileges. Despite being quite simple in their ideas, the attacks presented may have significant impact. The configuration required to carry out the DHCP attack, with each host configured to have a static MAC address, is a common configuration in many environments. A successful DHCP spoofing attack allows the attacker to force all the outgoing traffic or/and DNS requests to go through his own host.

The presented issue with checksum validation should at least warn users willing to place a NIDS on the dom0, to be able to check the internal, in-memory, traffic of a Xen virtualized environment. As we previously described, since there is no option that the receiving end can set to *refuse* bad checksummed traffic, *intrusion* or *evasion* should nearly always be possible. Moreover, the checksum based technique allows an attacker to identify the hosts that are physically on the same box and, by passively waiting for (re)boots, map them to their domain ID. This might be helpful in many attack situations: we have only discussed how it could improve the DHCP based attack.

It is important to mention that, with the spreading of virtualization-capable hardware architectures like AMD Pacifica [66] and Intel VT (Virtualization Technology) [67], HVM (Hardware-Virtualization based) domains are going to be more and more common than paravirtualized ones, for example because unmodified guest OSes can be run on them (e.g. Windows). Notwithstanding this, paravirtualization ideas still have application: paravirtualized drivers are faster than a fully emulated device and might still be used, for example, for networking. An example of this, on Xen, is the Windows GPL PV [68] drivers project.

In the following two subsections we describe a set of features that are likely to be merged in a future version of xenophobia and a set of fix/contermeasures that might by applied to block or mitigate the effect of the attacks.

The Xen security group was privately contacted before the release of this work.

6.2 Extending the tool

The tool design is modular enough to easily add new modules, for example to implement new attacks based on the watch mechanism.

The passive mode of the daemon could be improved by collecting (and storing in the db) more information about the running domains, for example adding the association of domain IDs to hosts or the result of a port scan [64] against the victim domain. The logic to introduce the ID-domain mapping is mostly already there and the integration should be simple. Having this part merged in would give a little advantage at domain destroy time: instead of checking all the IP addresses for the not-responding one, the daemon could simply request a list of the available domains from the kernel (checking the */proc/xenophobia/domains* entry) and look for the missing domain id. If all the hosts have the correct domID associated, the lookup of the correct MAC/host to attack at boot time would be quicker and would not require any more a network probing.

The logic handling the execution of the DHCP attack could also be improved, allowing a user to decide to perform the attack only on some hosts and/or to control the number of times the attack should be performed, if unsuccessful (consider the case in which the target host shuts down, a new one is booted and *then* the target boots again). This implies adding code to check if the DHCP attack was successful in the first place: this might be achieved by making the remote host generate some traffic and see if we can see that or by grabbing the network traffic and evaluating the DHCPREQUEST packet. This second case would not work if we are attempting to completely spoof our request.[1] The struct associated to each database entry could be extended to include these parameters and the userland *xphb* tool could be used to configure the behaviour of the daemon at runtime.

Considering profiling user activity, another possible set of attacks that might look promising are ones based on *timing* issues. Since a Xen guest can check the RTC (Real-Time Clock) and can figure out when and if it was scheduled out (by looking at the values inside the `shared_info` page) during the execution of an operation, it could use this information to measure response time on some Hypervisor-related operations.
Some testing code capable of mapping the used and free grant table references of a remote guest checking the response time differences of the associated hypercall was developed, but was not included because it was not mature enough. Moreover, an effective, advantageous, use of this approach has not been investigated further.

6.3 Mitigating the attacks

As we said previously, the Xen security team has been notified of the issues reported in this work and a couple of *potential solutions* have been proposed. Due to lack of time, no code patches were sent. The suggested solutions are:

- Add a Driver Domain member to the structure representing the alive

[1]Keep in mind that Xen Hypervisor knows the binding between a MAC address and the host it was assigned to and so an admin could use this information to prevent the spoofing of network traffic. This is the default configuration with a Solaris dom0.

domains and check against the value of this member inside the watch related function, instead of allowing EACCES and ENOENT for any domain.[2]

- Add a configuration option that a domU guest may set specifying that its network driver should always check the checksum value. This value should be used in conjunction with disabling the checksum offloading on all the guests, in case the service provider wants to stop *local* network mapping and the IDS evasion and insertion attacks.[3]

The domain ID exposure might be fixed using the XSM (Xen Security Modules) project, but the XSM call should be moved first in the sequence of checks that the grant hypercall performs. The Xen security team considers this one as a low priority issue[4] and in fact it is. It becomes useful only in conjunction with other attacks, as the XenStore one. It is worth mentioning though that some service provider might be interested in not showing to customers how many virtualized guests are active on the host and that this piece of information might become useful in performance-based attacks, that is, attacks that try to profile the activity of a guest based on the use of hardware resources.
The DHCP attack might be detected by the legal DHCP Server or a traffic analyzer by evaluating the contents of the DHCPREQUEST message that the victim broadcasts, checking if the gateway address or the nameservers address are different that the ones legally announced. An attacker can,

[2]If one is sure that he will not use Driver Domains, he might just remove the EACCES and ENOENT check from the source code.
[3]A Loadable Kernel Module can be also used for this purpose, following the approach that xenophobia uses on the sender end to play with the checksum.
[4]And a legal, XenStore based, way to enumerate domain IDs is available, as we said in Chapter 3

though, circumvent this detection by changing slightly the DHCP module code and sending a DHCPREQUEST with *legal* parameters and then sending a DHCPACK with modified nameservers and/or gateway address. This attacking option will be added in the next release of xenophobia. A traffic analyzer on the dom0 might still detect the attack checking the attacker DHCPOFFER and DHCPACK packets, but the attacker might manage to evade it using the checksum-based evasion approach. An admin might stop DHCP based attacks by filtering out DHCP packets from inside the dom0. For example, if the DHCP server is an external server, all the internal traffic carrying DHCPRESPONSE and DHCPOFFER should be dropped. External hardware/software solutions, as, for example, the Cisco DHCP Snooping [69] one, would not work, because no external device can check the in-memory traffic between guests.

Bibliography

[1] J. Smith and R. Nair, *Virtual Machines: Versatile Platforms for Systems and Processes.* Morgan Kaufmann, June 2005.

[2] (2008) Sun java virtual machine. [Online]. Available: http://java.sun.com/

[3] (2008) Microsoft .net framework. [Online]. Available: http://www.microsoft.com/NET/

[4] CitriX press office. (2007, Aug.) Citrix to enter server and desktop virtualization markets with acquisition of xensource. [Online]. Available: http://citrix.com/English/NE/news/news.asp?newsID=680808

[5] C.-H. Yen. (2007, Nov.) Solaris operating system hardware virtualization product architecture. 820-3703.pdf. [Online]. Available: http://www.sun.com/blueprints/1107/

[6] P. Ferrie. (2006) Attacks on virtual machines emulators. [Online]. Available: www.symantec.com/avcenter/reference/Virtual_Machine_Threats.pdf

[7] J. Rutkowska and W. Rafal. (2008) Xen 0wning trilogy. [Online]. Available: http://invisiblethingslab.com/bh08/

[8] N. A. Quynh. (2007) Hijacking virtual machine execution for fun and profit. [Online]. Available: http://video.google.com/videoplay?docid=1854946164969106185

[9] (1997) Superscalar microprocessor architecture. [Online]. Available: http://www.freepatentsonline.com/5603047.html

[10] (2008) Intel 64 and ia-32 architectures software developer's manuals. [Online]. Available: http://www.intel.com/products/processor/manuals/-

[11] J. Robin and C. Irvine, "Analysis of the intel pentium's ability to support a secure virtual machine monitor," 2000. [Online]. Available: http://citeseer.ist.psu.edu/robin00analysis.html

[12] A. S. Tanenbaum, *Modern Operating Systems.* Upper Saddle River, NJ, USA: Prentice Hall PTR, 2001.

[13] A. Silberschatz, P. B. Galvin, and G. Gagne, *Operating System Concepts.* New York, NY, USA: John Wiley & Sons, Inc., 2001.

[14] (2005) Ultrasparc user's manuals. [Online]. Available: http://www.sun.com/processors/documentation.html

[15] A. S. Tanenbaum and J. R. Goodman, *Structured Computer Organization.* Upper Saddle River, NJ, USA: Prentice Hall PTR, 1998.

[16] R. McDougall and J. Mauro, *Solaris(TM) Internals: Solaris 10 and OpenSolaris Kernel Architecture (2nd Edition) (Solaris Series).* Sun Microsystems Press, 2006.

[17] D. Bovet and M. Cesati, *Understanding The Linux Kernel.* Oreilly & Associates Inc, 2005.

[18] R. Love, *Linux Kernel Development (2nd Edition) (Novell Press).* Novell Press, 2005.

[19] G. J. Popek and R. P. Goldberg, "Formal requirements for virtualizable third generation architectures," *SIGOPS Oper. Syst. Rev.*, vol. 7, no. 4, p. 121, 1973.

[20] (2008) Qemu internals. [Online]. Available: http://bellard.org/qemu/qemu-tech.html

[21] (2008) Vmware: Virtualization via hypervisor, virtual machine and server consolidation. [Online]. Available: http://www.vmware.com

[22] P. Barham, "Xen and the art of virtualization," 2003. [Online]. Available: http://citeseer.ist.psu.edu/article/barham03xen.html

[23] (2008) Xen hypervisor. [Online]. Available: http://www.xen.org

[24] (2008) Xen 3.3. [Online]. Available: http://bits.xensource.com/oss-xen/release/3.3.0/xen-3.3.0.tar.gz

[25] (2008) The linux kernel. [Online]. Available: http://www.kernel.org

[26] (2008) Netbsd. [Online]. Available: http://www.netbsd.org

[27] (2008) Xenlinux. [Online]. Available: http://wiki.xensource.com/xenwiki/XenLinux

[28] (2008) Xend internals. [Online]. Available: http://wiki.xensource.com/xenwiki/XendInternals

[29] D. Chisnall, *The Definitive Guide to the Xen Hypervisor.* Prentice Hall Open Source Software Development Series, 2007.

[30] T. de Raadt. (2003) Openbsd: Buffer overflow solutions. [Online]. Available: http://kerneltrap.org/node/573

[31] I. Molnar. (2003) Linux: Exec shield overflow protection. [Online]. Available: http://kerneltrap.org/node/644

[32] PaxTeam. (2008) Documentation for the pax project. [Online]. Available: http://pax.grsecurity.net/docs/aslr.txt

[33] P.-H. Kamp and R. N. M. Watson. (2000) Jails: Confining the omnipotent root. [Online]. Available: phk.freebsd.dk/pubs/sane2000-jail.pdf

[34] GRSecurity. (2008) Grsecurity features. [Online]. Available: http://www.grsecurity.net/features.php

[35] iSEC. (2008) isec released vulnerabilities. [Online]. Available: http://www.isec.pl/vulnerabilities.html

[36] MOKB. (2007) Month of kernel bugs. [Online]. Available: http://projects.info-pull.com/mokb/

[37] twiz and sgrakkyu. (2007) Phrack64: Kernel exploiting notes. [Online]. Available: http://www.phrack.org/issues.html?issue=64&id=6#article

[38] iSEC. (2003) Linux kernel do_brk() vulnerability. [Online]. Available: www.isec.pl/papers/linux_kernel_do_brk.pdf

[39] noir. (2002) Phrack60: Smashing the kernel stack for fun and profit. [Online]. Available: http://www.phrack.org/issues.html?issue=60&id=6#article

[40] D. Song. (2000) Dsniff. [Online]. Available: http://monkey.org/~dugsong/dsniff/

[41] J. McHugh, A. Christie, and J. Allen. (2000) Defending yourself: The role of intrusion detection systems. [Online]. Available: www.cert.org/archive/pdf/IEEE_IDS.pdf

[42] (2008) Snort - the de facto standard for intrusion detection/prevention. [Online]. Available: www.snort.org

[43] D. W. Chadwick. (2004) Network firewall technologies. [Online]. Available: www.itsec.gov.cn/webportal/download/2004_network_fw_tech.pdf

[44] (2002) Guardian active response for snort. [Online]. Available: http://www.chaotic.org/guardian/

[45] The FreeBSD Documentation Project. (2008) Freebsd handbook, chapter 16, mandatory access control. [Online]. Available: http://www.freebsd.org/doc/en/books/handbook/mac.html

[46] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh, "Terra: a virtual machine-based platform for trusted computing." ACM Press, 2003, pp. 193–206.

[47] N. Provos. (2003) A virtual honeypot framework. [Online]. Available: http://www.citi.umich.edu/techreports/reports/citi-tr-03-1.pdf

[48] T. Ormandy. (2006) An empirical study into the security exposure to hosts of hostile virtualized enviornments. [Online]. Available: http://taviso.decsystem.org/virtsec.pdf

[49] Ilja van Sprundel. (2005) Fuzzing. [Online]. Available: http://events.ccc.de/congress/2005/fahrplan/attachments/683-slides_fuzzing.pdf

[50] D. A. D. Zovi. (2006) Hardware virtualization rootkits. [Online]. Available: http://www.blackhat.com/presentations/bh-usa-06/BH-US-06-Zovi.pdf

[51] J. Rutkowska. (2006) Subverting vista kernel for fun and profit. [Online]. Available: http://www.blackhat.com/presentations/bh-usa-06/BH-US-06-Rutkowska.pdf

[52] D. Murray. (2007) Improving xen security through domain-zero disaggregation. [Online]. Available: http://xen.org/files/xensummit_fall07/22_DerekMurray.pdf

[53] G. Cocker. (2006) Xen security modules. [Online]. Available: http://xen.org/files/summit_3/coker-xsm-summit-090706.pdf

[54] (2008) Selinux/flask. [Online]. Available: http://www.nsa.gov/selinux/index.cfm

[55] (2008) Nsa - national security agency. [Online]. Available: http://www.nsa.gov/

[56] R. Droms, “Dynamic Host Configuration Protocol,” RFC 2131 (Draft Standard), Mar. 1997, updated by RFCs 3396, 4361. [Online]. Available: http://www.ietf.org/rfc/rfc2131.txt

[57] A. Ornaghi and M. Valleri. (2003) Man in the middle attacks. [Online]. Available: http://www.blackhat.com/presentations/bh-europe-03/bh-europe-03-valleri.pdf

[58] J. Postel, "Domain Name System Structure and Delegation," RFC 1591 (Informational), Mar. 1994. [Online]. Available: http://www.ietf.org/rfc/rfc1591.txt

[59] A. Menon, A. L. Cox, and W. Zwaenepoel. (2006) Optimizing network virtualization in xen. [Online]. Available: http://www.usenix.org/events/usenix06/tech/menon/menon_html/paper.html

[60] T. H. Ptacek and T. N. Newsham, "Insertion, evasion, and denial of service: Eluding network intrusion detection," Tech. Rep., 1998.

[61] C. Benvenuti, *Understanding Linux Network Internals.* O'Reilly Media, Inc., 2005.

[62] (2007) The libnet packet construction library. [Online]. Available: http://www.packetfactory.net/libnet/

[63] (2008) Tcpdump/libpcap public repository. [Online]. Available: http://www.tcpdump.org/

[64] (2008) Nmap free security scanner for network exploration and hacking. [Online]. Available: http://nmap.org/

[65] (2007) Gnu general public license (gpl). [Online]. Available: http://www.gnu.org/copyleft/gpl.html

[66] (2008) Industry leading virtualization platform efficiency.

[Online]. Available: http://www.amd.com/us-en/Processors/ProductInformation/0,,30_118_8796_14287,00.html

[67] G. Neiger, A. Santoni, F. Leung, D. Rodgers, and R. Uhlig. (2006) Intel virtualization technology: Hardware support for efficient processor virtualization. [Online]. Available: http://download.intel.com/technology/itj/2006/v10i3/v10-i3-art01.pdf

[68] (2008) Xenwindowsgplpv - xen wiki. [Online]. Available: wiki.xensource.com/xenwiki/XenWindowsGplPv

[69] (2008) Understanding and configuring dhcp snooping. [Online]. Available: http://www.cisco.com/en/US/docs/switches/lan/catalyst4500/12.1/12ew/configuration/guide/dhcp.pdf