J Comput Virol (2010) 6:317–328
DOI 10.1007/s11416-009-0131-7

ORIGINAL PAPER

Practical overview of a Xen covert channel

Mickaël Salaün

Received: 17 December 2008 / Accepted: 31 July 2009 / Published online: 19 August 2009

Abstract Covert channels have been known since a long time and under various forms. Methods used by designers to exchange information with discretion depend mainly on their creativity. These streams of data are so stealthy that they can be easily used by some dishonest persons or malwares. In an (supposed) other world, the virtualization of operating systems has brought a higher flexibility in the deployment of server farms and shared hosting. It also brings hopes concerning security through partitioning. These two subjects are not so obviously linked, but for each form of new technologie we need to think about past errors to be able to bypass them before they come up. The designer may not know all threats, and new exploitation techniques can appear, more or less planned. In this context, this article explains technologies used by the Xen paravirtualizer about memory management and virtual guests. Thereafter, it explains how to exploit this mechanism to reach a new method of covert channel for virtual machines. Finally, experimental results show that the proof of concept can stealthily transfer data between virtual machines.

0 Introduction

Xen is now one of the most used virtualization systems in the shared hosting world. This solution has many advantages as its performance (due to the paravirtualization), its various configurations, the precision of its security policy, and moreover it is a free software. However, security problems exist for this architecture and a new one will be described here with the covert channels approach.

M. Salaün (✉)
ESIEA Laboratoire de virologie et de cryptologie opérationnelles, Laval, France
e-mail: salaun@esiea-recherche.eu

A lot of malwares and especially *rootkits* use some techniques to remain stealthy. The great majority of this techniques are publicly known but it can be very hard to prevent such infection. Today, new computers provide virtualization features and malwares are taking advantage of this. Parallel to this, the communication is an important point which is critical for some new kind of malware like K-ary virus [3]. Moreover, a covert channel wisely used can be a good strategy to bypass detections. It is then important to know such channels to be able to detect and avoid them.

For a better understanding, I will first define isolation, and specify where it is needed and what it requires. Then, the definition of the covert channels will help to understand the different existing methods to transfer information discreetly between several accomplices. After this, a description of the memory management of Xen will introduce the mechanism of a covert channel in this virtualizer.

1 Importance of isolation

1.1 Separate property

Mutualisation offers are by nature virtualization offers. You can find various solutions, from context insulation (VServer, OpenVZ...) up to a mix of virtualization methods (like paravirtualisation). In the case of virtualization solutions, it is fundamental for the provider to be able to partition the various machines which he rents out. Except for special needs, nobody wishes to disclose his data to his neighbours. In the same way, an exchange of virtual data between two machines can be interesting if it is controlled, otherwise it can be a problem.

In a virtualized server farm, communication between the numerous computers is in many cases essential for their

good performance. However, for providers of virtualized host solutions, some constraints are essential. Their customers are either neutral towards one another, or in cooperation, or competitors. I propose to focus on this last case.

1.2 Compromised system

In the hypothesis of a compromised rival's server, as in any attack aimed at spying, the prior objective of the attacker is to stay in place as long as possible. In other words, the installed backdoor must be as discreet as possible. Today, there are several methods allowing to remain stealthy, including techniques which consist in running a program entirely in memory throughout its execution.[1]

To be useful, a backdoor must be able to communicate with the outside. In some cases, the only means is to use the network. In the case of virtual hosted machines, the attacker can use another method: communication between virtual guests hosted on a same physical computer. In the virtual hosting solutions, it is not common that the provider offers such functionalities to his customers, when they did not give their consent.

A backdoor being stealthy for the operating system can be detected during network exchanges if these are correctly analysed.[2] In such a case, discreet communication between two virtual machines becomes critical for intrusion detection and also for the security of this operating system.

Some kind of malwares called K-ary virus [3] can be split into some inoffensive parts. If they can exchange information, they are so able to combine themselves to create a malicious payload. The communication channel used to exchange data between them can be many data streams including covert channels. If this kind of code using rootkit techniques is able to use such channels, it can be then possible to remain really stealthy and for a long time…

2 Covert channels

2.1 Definition

In confinement environments the main problems are the data access and leak. The main preoccupation of designers is the unauthorized access (or modification) to data [5]. For now it can be controlled through access rules if they are properly designed, well done and applied. It remains the leak problem which is not so simple to prevent. The data exchange is needed for many reasons, but sometimes it is unwanted and a leak can be due to a covert channel.

There are two interesting definitions about covert channel which complement one another:

- *Covert channels are those that "use entities not normally viewed as data objects to transfer information from one subject to another."* [4]
- *Given a nondiscretionary (e.g., mandatory) security policy model M and its interpretation I(M) in an operating system, any potential communication between two subjects I(Sh) and I(Si) of I(M) is covert if and only if any communication between the corresponding subjects Sh and Si of the model M is illegal in M.* [8]

This type of channel uses one or several legitimate streams to pass information between two entities through a way not initially envisaged by the original system designers. Interest to create such a channel is mostly to pass through a security policy forbidding such communication or to remain hidden from the rest of the system.

The simplest covert channels are undoubtedly those transmitting information to an accomplice thanks to a protocol created on top of legitimated data. For example, a simple text can hide some information with the insertion of spelling errors or other tricks. It can be easy to extract the hidden information if you know how to do, but if you are not aware of this channel it is impossible and the hidden message will remain invisible.

The steganography can be considered to be a form of covert channel given the invisibility for information transmission. There are several methods but the principle remains the same: to hide information in a media (image, video, sound) or any other data type. The main need is to be able to extract a hidden information from a piece of media. In this case, the media protocol is respected, but a specific information can remain inside the data. The initial goal can seem the same as a regular file, but it is turned aside from its traditional use.

2.2 Properties

The first feature of a covert channel is to remain hidden. Thanks to legitimate communication channels a stealthy channel can be set up. The most interesting feature of a communication channel is its bandwidth. The data stream can be used for multiple things according to its performances. However, it is obvious that bandwidth is linked to performances and can slow it.

There are three main covert channel classes [9]:

- Storage and timing channels
- Noisy and noiseless channels
- Aggregated and nonaggregated channels

[1] *Sanson the Headman* is a good example (http://sanson.kernsh.org).

[2] However, if a backdoor use a good network covert channel and use a low bandwith it can remain undetectable…

```
# xm list
Name                                ID     Mem VCPUs      State   Time(s)
Domain-0                             0     747     2      r-----  1236.1
dom1                                 3     128     1      -b----     8.2
dom2                                 4     128     1      -b----     7.6
```

Fig. 1 Existing domain listing

A potential covert channel is a storage channel if its scenario of use "involves the direct or indirect writing of a storage location by one process (i.e. a subject of I(M)) and the direct or indirect reading of the storage location by another process [6]. A potential covert channel is a timing channel if its scenario of use involves a process that" signals information to another by modulating its own use of system resources (e.g. CPU time) in such a way that this manipulation affects the real response time observed by the second process [6]. Both of this techniques need a synchronisation to be able to communicate and they differ by their media.

Noisy or noiseless channels are differing by their probability to correctly transmit data without any doubt. With probability 1, the receiver can decode the value sent. A noisy channel need error-correcting codes to be usable as a noiseless one. Therefore the resulting channel will have a lower bandwidth than the similar noise-free channel.

The sender needs to share synchronization data with the receiver. Multiple data variables, which could be independently used for covert channels, may be used as a group to amortize the cost of information synchronization. We say the resulting channels are aggregated. Depending on how the sender and receiver set, read, and reset the data variables, channels can be aggregated serially, in parallel, or in combinations of serial and parallel aggregation to yield optimal bandwidth [9].

Characteristics of covert channels are their stealthiness and their bandwidth. They are rival: more a bandwidth is important, more it will be potentially suspected, and can be revealed. However, more a covert channel allows to transfer a big amount of information, more it is considered as functional and critical.

The detection of a covert channel is an important point for its designers as well as for its detractors. More important the bandwidth is, less tactful the communication can be and a suspicious activity can be detected.

All of this is depending on a common channel between two accomplices. If we want a message to be read, we need to put it in an accessible place. Otherwise, it is impossible to share information. The main difficulty is to really control this places, but most of the time, it is depending on the architecture. The problem is to find all possible channels shared between accomplices. The next section will explain what is in common between two virtual machines hosted in a same physical one and so what can be used to create such a covert channel.

3 The Xen memory management

3.1 Prerequisites

3.1.1 Domains

The "domain 0", shortly named *dom0* is responsible for managing Xen and devices relations. It is aimed at being only used by the system administrator to manage other virtual domains.

It contains the drivers who permit to communicate with the devices. Indeed, these do not have to be especially conceived for Xen, but simply for the operating system of *dom0*. This one can be a Linux, FreeBSD, NetBSD or else OpenSolaris operating system. We can so use a lot of devices.

Next version of Xen will provides a feature to protect from a large number of problems induce by drivers. An architecture providing IOMMU or similar technology can control access to memory devices. It may delegate the management of drivers to another area that the *dom0*. The devices are dedicated to a single instance, no matter their particularities (network card, graphics card…). These architectures are very interesting, because they can so protect themselves from errors that can be present in drivers. So, bugs happening on the equipment or the driver are not directly propagated in *dom0*, and this one can come back to a stable state without crash (that will freeze the whole system).

The other goal of *dom0* is to provide tools allowing to control other domains. The utilities provided are accessible from the command `xm` (see Fig. 1). It allows to list, to create, to access, to change and to destroy the different domains.

The others domains are called "domain user" shortly named *domU*. They are users' virtual machines. If needed, they can be restricted with the Xen security policy.

3.1.2 Hypercalls

Any communication between guests and the Xen hypervisor is made with *hypercalls*. They allow to perform requests towards the hypervisor to do actions and to get information. Each of them is identified by a unique number and linked to one definition.

A call to a *hypercall* is made in the same way as classical system call which can be used in common operating systems. We need to invoke an interrupt event[3] for which it is

[3] Xen use the interruption 0x82 to communicate with the guest.

```
asm volatile("int $0x82"
             : "=a" (result), "=b" (ignore)
             : "a" (17), "b" (0));
```

Fig. 2 Version number retrieval with Xen 2: interruption

```
asm volatile("call hypercall_page + (17 * 32)"
             : "=a" (result), "=b" (ignore1), "=c" (ignore2)
             : "1" ((long)(0)), "2" ((long)(NULL))
             : "memory" );
```

Fig. 3 Version number retrieval with Xen 3: call

```
$ readelf -l mini-os

Elf file type is EXEC (Executable file)
Entry point 0x0
There are 2 program headers, starting at offset 52

Program Headers:
  Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
  LOAD           0x000080 0x00000000 0x00000000 0x10b0c 0x1dd24 RWE 0x20
  GNU_STACK      0x000000 0x00000000 0x00000000 0x00000 0x00000 RWE 0x4

 Section to Segment mapping:
  Segment Sections...
   00     .text .rodata .data .bss
   01
```

Fig. 4 Kernel header example

```
$ readelf -l xen-3.2-1-i386

Elf file type is EXEC (Executable file)
Entry point 0x100000
There are 1 program headers, starting at offset 52

Program Headers:
  Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
  LOAD           0x000080 0x00100000 0x00100000 0xc44d4 0x116000 RWE 0x40

 Section to Segment mapping:
  Segment Sections...
   00     .text
```

Fig. 5 Xen hypervisor header

necessary to initialize the register *EAX* with the value of the corresponding *hypercall*. Needed parameters are transmitted with other registers.

In the version 3 of Xen, calls to *hypercalls* have been altered for a cleaner use and especially a more important usability for the hypervisor which mape corresponding interruption in the guest memory. If a change had to take place in a future version (or to a debug/rootkit purpose), the guests would not need to be changed; only *mapping* of Xen would differ.

To show the use of one simple *hypercall* allowing to get the version of the common hypervisor, an assembly command (see Figs. 2 and 3) allows to call the corresponding hypercall. For this we need to give the correct number of the *hypercall xen_version* (set register *EAX* to 17) and give the good command number to get the value (set register *EBX* to 0). Following this call, we can get back the version number of Xen in the *result* variable.

3.1.3 The boot process

During the development of an operating system compatible with Xen, the first step is to declare various information in the kernel allowing to make link between this one and Xen. The kernel of a guest is in fact an *ELF* file (see Fig. 4) statically linked.

The *ELF* (Fig. 5) must have a specific part to be compatible with Xen. We can find it in the header section (*__xen_guest*).

```
.section __xen_guest
    .ascii  "GUEST_OS=Mini-OS"
    .ascii  ",XEN_VER=xen-3.0"
    .ascii  ",VIRT_BASE=0x0"
    .ascii  ",ELF_PADDR_OFFSET=0x0"
    .ascii  ",HYPERCALL_PAGE=0x2"
    .ascii  ",PAE=no"
    .ascii  ",LOADER=generic"
    .byte   0
```

Fig. 6 Information section of the guest OS

```
for ( i = 0; i < (PAGE_SIZE / 32); i++ )
{
    p = (char *)(hypercall_page + (i * 32));
    *(u8  *)(p+ 0) = 0xb8;     /* mov  $<i>,%eax */
    *(u32 *)(p+ 1) = i;
    *(u16 *)(p+ 5) = 0x82cd;   /* int  $0x82 */
    *(u8  *)(p+ 7) = 0xc3;     /* ret */
}
```

Fig. 7 Xen writing into the guest's memory the needed code for *hypercall* functions (standard)

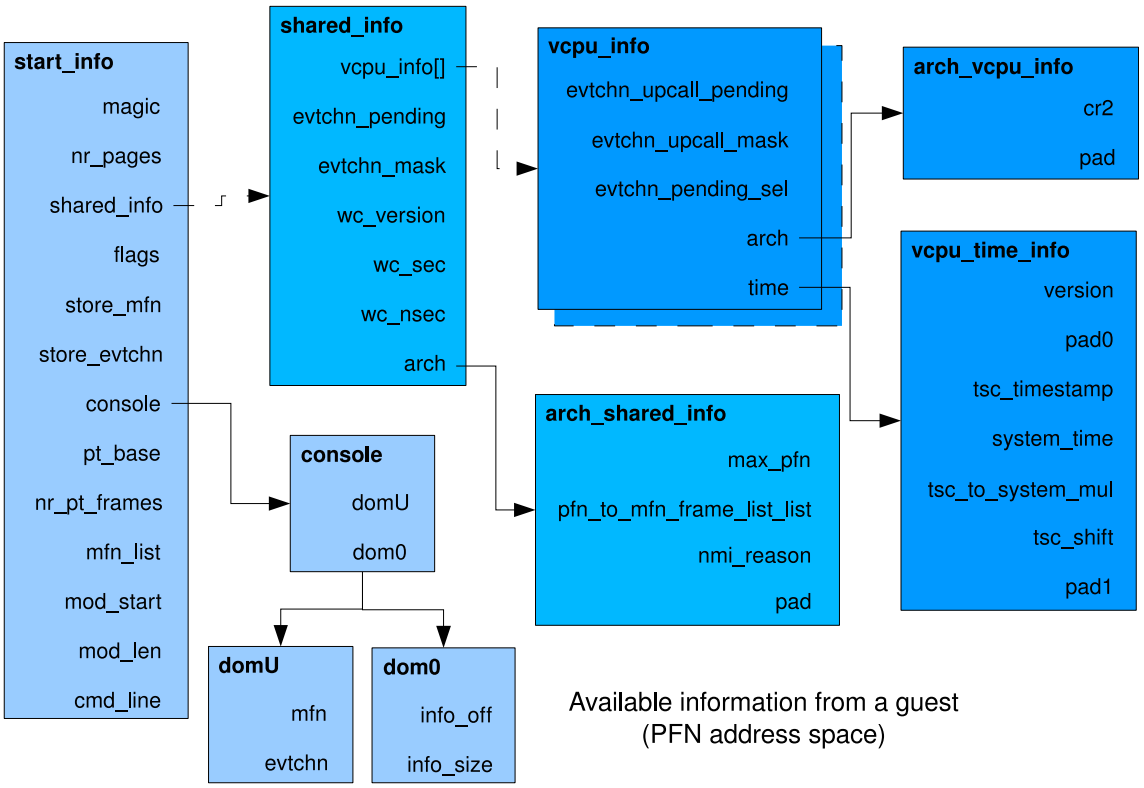

Fig. 8 Tree of the *start_info* structure

This field holds a string of char which specifies various information (Fig. 6) whereof the *offset* and the address of *hypercalls* table.[4] There are also some fields showing the compatibility level with the hypervisor.

Further in the guest code, we must allocate space to Xen where it can mape[5] functions to call *hypercalls* (Fig. 7).

During the boot process, Xen needs to allocate memory to allow execution and initialisation of data. The main step is the installation of structures (*start_info*, *shared_info*, *vcpu_info*) and their sharing with the OS (Fig. 8).

[4] The value of the page is 4 Ko: 12 bits shift.

[5] *xen/xen/arch/x86/x86_32/trap.h* : *hypercall_page_initialise*

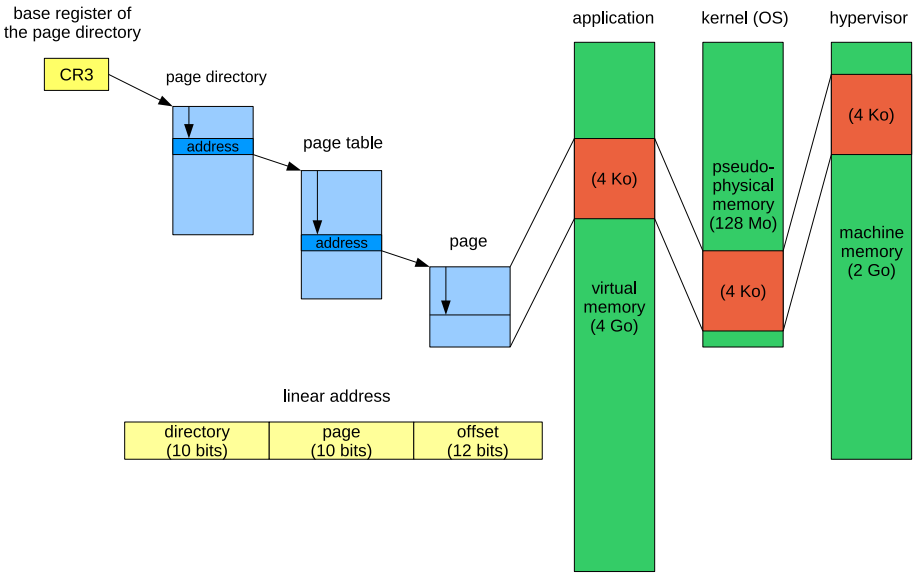

Fig. 9 Pagination example of the memory (32 bits paravirtualized architecture)

3.2 Memory property

3.2.1 Memory allocation

During initialisation, Xen allocates memory for its own use, but also for virtual memory spaces. These allocations are done in known places of memory. It allows then to extend address space for guests. Xen also needs to know the real owner of each memory zone to be able to apply isolation rules. So, each system owns his initial predefined zone, but it is also able to dynamically expand to a maximum defined size thanks to the *Balloon Driver*. This new feature is not needed for a common use and is not activated by default.

During a guest initialisation, an information zone (*Shared Info*) is copying to a specific memory place. This data permits to get all necessary information for a communication with its hypervisor. Some data about memory pages of the current domain (size, address…) are stored in this area.[6]

3.2.2 Pseudophysical memory

Contrary to traditional memory management mechanisms where two address levels (virtual addresses and physical addresses) are used, a paravirtualized system can use another level (machine addresses) doing the glue with the real machine memory and the pseudo-physical memory of virtual machine (Fig. 9).

This new step of abstraction affects the memory fragmentation, but the guest system can manage this transition itself. We have so an optimization of the memory allocation for the guest by the guest. Indeed, it can allocate enough contiguous memory space at the physical memory level to avoid fragmentation which will slow down for its own execution.

3.3 Isolation

It should be noted that it is possible to list the *mfn2pfn* table (*Machine Frame Number to Pseudo-physical Frame Number*) who permits to have a clear idea of the memory used on the physical machine by Xen and his guest. We can try to browse existing pages and to infer the different memory pages[7] (Fig. 10). From that it is easy to find the size of the physical machine memory.

This information is not immediately available because we caught exception depending on what we are trying to read. Indeed, Xen detects an illegal access to the mapping table and forgives access to it. On the other hand, we can get out from this problem with the creation of our own exception table. Like this, we have then the possibility to know where the exception comes from and in the test cases, we can jump in clean-cut addresses of our code depending on the action. This technique allows to free ourselves a bit more from the Xen control with the use of exceptions to get information.

4 XenCC

A communication mechanism exists between guests. It is called *XenStore* and allows to share memory pages with read or write mode. With this, we can exchange data between guests depending on their right to use it. Another unrecog-

[6] For more details, look at the start_info struct defined in the `xen/include/public/xen.h` file.

[7] Existing memory, not allocated, mapped for another domain, mapped for Xen or mapped for the current guest.

```
map NOread  : 00000000-00000000 (1)
other       : 00000001-000000ff (255)
page fault  : 00000100-00000235 (310)
other       : 00000236-0000023a (5)
page fault  : 0000023b-00000bdc (2466)
other       : 00000bdd-00000be1 (5)
page fault  : 00000be2-00000c4c (107)
map read    : 00000c4d-000025ff (6579)
other       : 00002600-000027ff (512)
map read    : 00002800-00002888 (137)
other       : 00002889-00002a88 (512)
map read    : 00002a89-00002aff (119)
other       : 00002b00-00010846 (56647)
map NOread  : 00010847-00010847 (1)
other       : 00010848-00017bff (29624)
map read    : 00017c00-00017c38 (57)
other       : 00017c39-00017e31 (505)
map read    : 00017e32-00017fff (462)
other       : 00018000-000181ff (512)
map read    : 00018200-00018360 (353)
other       : 00018361-00018361 (1)
map read    : 00018362-0001836b (10)
other       : 0001836c-00018438 (205)
map read    : 00018439-000185ff (455)
other       : 00018600-000198a1 (4770)
map NOread  : 000198a2-000198a2 (1)
other       : 000198a3-000198a7 (5)
map read    : 000198a8-000198a8 (1)
other       : 000198a9-00019bff (855)
map read    : 00019c00-00019c11 (18)
other       : 00019c12-0001afff (5102)
page fault  : 0001b000-0001b1ff (512)
other       : 0001b200-0001bfff (3584)
page fault  : 0001c000-0001c0a2 (163)
other       : 0001c0a3-0001c1ef (333)
page fault  : 0001c1f0-0007ffff (409104)
missing     : 00080000-003fffff (3670015)
```

Fig. 10 Organisation example of the physical memory (infer from a *domU*)

nized method, depending on the Xen architecture, permits to share data between guests.

The analysis of memory mechanisms allowed to build a tool on the same principle as *XenStore*, but without any "constraint" of the security policy. This covert channel is a proof of concept (PoC) to communicate between guests independently of the domain (*dom0* or *domU*). For this, we need to have rights to insert a module in the kernel, otherwise, to affect the kernel of our guest domain, which permits to use *hypercalls*.

4.1 Mechanism

It uses the mapping table between physical addresses and pseudo-physical addresses of the hypervisor. It can be noted that we need to use an *hypercall* (*mmu_update*) to establish relationship.

It is important to determine that this table is readable by all guests because it is the only one. Of course, only the address range owner can modify it. Is it about addresses, but by no means their data. Indeed, if we want to read at a place, we need to be able to map it in memory. Here, we can only map our own data (memory pages). This relation table normally holds addresses, but given that there is no check, we can write what we need: data.

The principle of the covert channel is simple: to write a tag in order to (notably) be able to find the place, and after that write data that we want to pass to our friend at the correct place. When data is written, nobody aware of our tag can find and extract information. So it is a half-duplex channel when two guests know their tag each other. This new communication channel is working well on Linux 2.6 x86-32. Readers who are interested can refer to the PoC code for more information.[8]

This idea was already noted down in 2006 at the *xen-devel* mailing-list;[9] but then, like so many problems, it was noticed that the practical part would be hard and so this threat was, until now, only theoretical. Finally, as developers thought, this method permits to override the Xen security policy.

4.2 Followed communication

4.2.1 Sharing protocol

The protocol built in the second version of this software is based on a header containing fields (Fig. 11):

[8] XenCC (GPL v3) available at the *Xen devel* mailing-list or at http://digikod.net/public/XenCC.

[9] http://lists.xensource.com/archives/html/xense-devel/2006-02/msg00001.html.

```
typedef struct
{
    unsigned long tag[PFN_HEADER_TAG_SIZE];
    unsigned long ack;
    unsigned long rest;
    unsigned long size;
    unsigned long data[XENCC_DATA_PAGES];
} rb_t;
```

Fig. 11 Description header of the shared data

```
// comment this for the second guest !
#define XENCC_ME_FIRST

// better with @ > PAGE_SHIFT bits (but not tactful)
#define XENCC_TAGS_1 {123456, 13641, 1616}
#define XENCC_TAGS_2 {151651, 1416, 469564}

#ifdef XENCC_ME_FIRST
#define XENCC_TAGS_A XENCC_TAGS_1
#define XENCC_TAGS_B XENCC_TAGS_2
#define XENCC_ME 1
#define XENCC_OTH 2
#else
#define XENCC_TAGS_A XENCC_TAGS_2
#define XENCC_TAGS_B XENCC_TAGS_1
#define XENCC_ME 2
#define XENCC_OTH 1
#endif
```

Fig. 12 Tag definition used by the two guests

- a tag to identify a communication stream,
- an acknowledge value,
- the remaining data size,
- the current data size.

For simplification purpose, this different fields have a base size of 4 bytes (for a 32 bits base system). The tag permits to identify a simplex stream. Indeed, although Pseudo-physical Frame Number (PFN) addresses are readable, we need a mechanism to find writing area of the accomplice among all values of addresses. The communicating guests can so identify themselves when they know the tag's accomplices. The tag is a succession of plain words (32 bits) (Fig. 12). If we want to identify a tag without any problem, it is enough to set it a greater value than the biggest address of a page ($2^{PAGE\_SHIFT} - 1$). However, this value can be easily suspected wrong because it can't refer to a page. It is so possible to choose any values of 4 bytes. The last point remains to initially share this one with the accomplice.

The communication is correctly established thanks to an acknowledge field which permits to know if the accomplice really reads the current buffer when he writes this same value in his header.

Two additional fields stand for the current and future data size. They allow to know remaining data to extract.

Indeed, data are following the header field. Their size having previously be defined, we know when to stop.

4.2.2 Driver loading

The code of this PoC is a device driver which creates `/dev/xencc`. It so permits to quickly test the exploit on different virtual machines. For now, a compilation is needed for each guest because of a static tag definition. You must be careful to differentiate each driver from an other to avoid self speaking. So, the only thing to change is the tag value with a commented (or not) the `XENCC_ME_FIRST` definition in the source file (Fig. 12). After the compilation, it remains to insert the generated module inside the kernel creation of the communicating device (Fig. 13).[10]

When the driver is loaded, it first try to allocate pages needed for the round buffer. This operation may be unsuccessful if their is not enough pages continuously free in memory. This is why we can't have a too big buffer.

4.2.3 Writing in the mfn2pfn table

To have a complete device, we need to define a structure *fops* to make functions available. Their goal is to permit different interesting actions which can be made such as: open, close, write and read the device. (Fig. 14).

[10] You need *udev* to automatically create the device.

```
dom1:~/xencc_0.2# make
make -C /lib/modules/2.6.18-6-xen-686/build M=/root/xencc_0.2 modules
make[1]: Entering directory '/usr/src/linux-headers-2.6.18-6-xen-686'
  CC [M]  /root/xencc_0.2/xencc.o
  Building modules, stage 2.
  MODPOST
  CC      /root/xencc_0.2/xencc.mod.o
  LD [M]  /root/xencc_0.2/xencc.ko
make[1]: Leaving directory '/usr/src/linux-headers-2.6.18-6-xen-686'
dom1:~/xencc_0.2# insmod xencc.ko
xencc loaded guest 1
xencc order: 4
xencc alloc_pages: c17afc00
xencc page_to_pfn: 0000f2e0
xencc pfn_to_mfn: 00004dd0
xencc allocate_mfn : 00004dd0
xencc max 9
xencc pfn0: 00004dd0 -> 00025063
xencc pfn1: 00004dd1 -> 00000588
xencc pfn2: 00004dd2 -> 00072a3c
xencc pfn3: 00004dd3 -> 00000000
xencc pfn4: 00004dd4 -> 00000000
xencc pfn5: 00004dd5 -> 00000000
xencc pfn6: 00004dd6 -> 00000000
xencc pfn7: 00004dd7 -> 00000000
xencc pfn8: 00004dd8 -> 00000000
xencc nb_success: 9
xencc rb_mfn 00004dd0
```

Fig. 13 Device driver loading

```
static struct file_operations xencc_fops = {
    .owner = THIS_MODULE,
    .read = xencc_read,
    .write = xencc_write,
    .open = xencc_open,
    .release = xencc_release,
};
```

Fig. 14 Actions definition

The `xencc_write` function goal is to copy data from the userspace to the `mfn_cc_write` function. This one writes into the *mfn2pfn* table.

4.2.4 Data extraction

Once data are stocked in the table, the accomplice host can read inside to get the message. To find the correct index, it is needed to read all entries and once it is found, we store it for not having to search it next time. However, it is preferable to reread the tags each time to assure that data are still for us. Thanks to the *size* field, we know how many data need to be read and the *rest* field indicates if we need another pass or if it is finished.

To allow his accomplice to update his data, we need to signal that the reading is done for each tour. For this, the *ack* field value is copied in our header. The other guest is repeatedly reading our header so the read event will be caught and a data update will be done. This synchronisation is the most time consuming task, but in an isolation context like this, there is no other way to do.

4.3 Communication session

Put into practice, we can see on the Figs. 15 and 16 the generated output for a (little) message written by the second guest and a reading from his accomplice.

If the debug is enabled,[11] we can see the driver behavior, notably what is written in the *mfn2pfn* table (header and data).

4.4 Experimental results

One of the most important points in a communication channel is the bandwidth it can reach. For covert channels, this is a characteristic that demonstrate their usability. This was in addition one discussed point with some people working on Xen.

Given that we use a *hypercall* (*mmu_update*) which is copies a complete data row (normally addresses) with one hypervisor call, the bandwidth is relatively as big as the

[11] Take care of the slowdown due to the log file explosion and of the possibles errors cause by the debug messages!

```
dom2:~/xencc_0.2# echo MSG > /dev/xencc
xencc open
xencc xencc_write count:4 *offp:0 offp:d051bfa4
xencc write to guest 1
xencc mfn_cc_write d106c6f0, 4, 0
xencc max 1
xencc pfn0: 0003edaf -> 00000001
xencc pfn1: 0003edb0 -> 00000000
xencc pfn2: 0003edb1 -> 00000004
xencc pfn3: 0003edb2 -> 0a47534d
xencc nb_success:4 length:4
xencc write
xencc pfn_find_tag
xencc mfn 00004dd0 : 00025063 (0)
xencc mfn 00004dd1 : 00000588 (1)
xencc mfn 00004dd2 : 00072a3c (2)
xencc find tag!
xencc oth_mfn 00004dd0
xencc RST OK
xencc write new
xencc release
```

Fig. 15 Guest 1 write (short debug)

```
dom1:~/xencc_0.2# cat /dev/xencc
xencc open
xencc read from guest 2
xencc read offp 0
xencc pfn_find_tag
xencc re-searching tag...
xencc mfn 0003edac : 0001e240 (0)
xencc mfn 0003edad : 00003549 (1)
xencc mfn 0003edae : 00000650 (2)
xencc find tag!
xencc oth_mfn 0003edac
xencc oth_header ack:1 rest:0 size:4
xencc pfn_ack
xencc ACK 1
xencc nb_success: 1
xencc xencc_data_size 4
MSG
xencc read from guest 2
xencc read offp 0
xencc pfn_find_tag
xencc mfn 0003edac : 0001e240 (0)
xencc mfn 0003edad : 00003549 (1)
xencc mfn 0003edae : 00000650 (2)
xencc find tag!
xencc oth_mfn 0003edac
xencc oth_header ack:1 rest:0 size:4
xencc waiting...
xencc pfn_ack
xencc ACK 2
xencc nb_success: 1
xencc xencc_data_size 0
xencc release
```

Fig. 16 Guest 2 read (short debug)

read and writes process in memory. The part that takes the most time is the acknowledge waiting from the other guest. Logically, the "buffer" size growth allows a significant increase of bandwidth.

During the second version of this PoC, it was put in place a *ring buffer* process to allow the data transfer, either by segmented copies managed by the user, but with a pseudo-continuous data stream. It's so possible to transfer entire file, doesn't matter its size, from a virtual machine to another. Nevertheless the developed protocol isn't optimised, it gives an idea of performance of such a covert channel use.

Bandwidth tests have been done with custom file transfer. To see differences that imply, different file sizes have been transferred. The size buffer repercussion, in other words the needed pages number repercussion, was measured for a range of buffer size.

To experiment this, the most important is to be synchronised between the two domains. Indeed, XenCC is synchronous and waits until the message has been read. It is needs because of the round buffer. It need to wait until it receives an acknowledgment from his accomplice to continue the transfer with next parts of data.

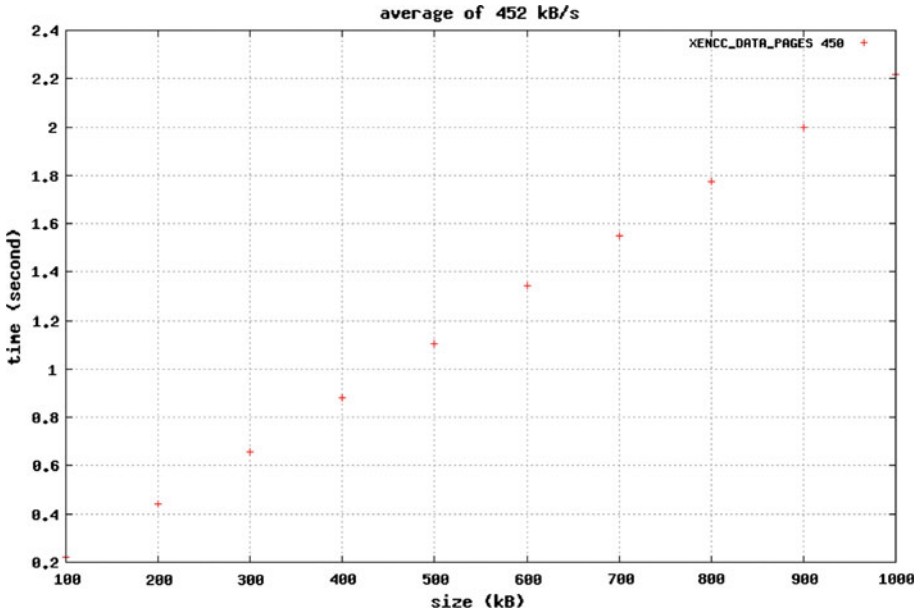

Fig. 17 Data copy with a 450 pages buffer size

The conclusion, although easily predictable, is simple: the available bandwidth depends on the buffer size. The bandwidth increases twice faster if the buffer size is doubled. With one page buffer (so 4 useful bytes), it reaches 1 kB/s. Tests show that there is no problem until it reaches 450 pages for the data field, that means an approximately bandwidth of 452 kB/s! Of course, this depends on the virtual machines configuration, but for this conditions, the maximum bandwidth is big. Logically, the transferred size doesn't matter the transfer speed. The bandwidth graph shows that it is linear (Fig. 17).

The guests system load is insignificant (an average of some tenth of percent usage for a 2 GHz processor). So, we can conclude that the bandwidth is significant but also constant.

The drawback of this method is the "virtual consumption" of memory needed for a little amount of transmitted data. Indeed, we allocate an entire page table of memory where we only use "addresses" and not reserved data. We must so transmit data little by little. With a good synchronisation, we have however a really speed channel in light of its stealthiness.

4.5 Counter measures

For now, there is no implemented solution to detect this sort of communication. It is however possible to do some statistics about *hypercalls* usage, but it is not sure to be a good solution depending on the *domU* usage…Indeed, it might be possible that a *domU* use a lot of *mmu_update* for it's one use, and the covert channel can also be use tactfully to minimize exchanges. However, an interesting idea is to look at *mmu_update* of all domains to come up similarity usage. It is so possible to suspect some relations between different *dom*, but the better thing is to prevent this kind of communications.

In order to prevent similar transfer, a solution for Xen would be to check addresses validity for the *PFN*. Like this, there is a write limitation, but the drawback is time consumption for each call to change an address. However, it will be pretty simple to bypass this check with an adapted algorithm who respects this new rule by legitimating such addresses.

There is another way to manage the pagination with the use of *shadow page tables*. In this case, the virtualized guest doesn't manage his machine pages. Instead, it is Xen who catches guests' mappings and replaces with Xen's one. To use this method, the pages table must be marked read only that will result with an exception which can be caught by the hypervisor. The main purpose is to let Xen manage machine memory. The drawback of this is the performance falling if the address translation is software.

The table *mfn2pfn* is a good idea to increase performances of memory mapping, but it will be wise to create a table for each guest. They will have the same protection mechanism that common memory and will be unreachable for other guests.

In light of the table, it seems that there is no big performances problem in comparison with the commutation speed (performed by Xen) and the memory space needed by a table.

The most appropriate solution is so to give a customized table for each domain. Like this, each guest will only have his addresses readable and up to date. So it will be impossible to read other guest table, but only to know where we are

in the physical machine memory, and so be able to optimize our allocations like with the common solution. This method implementation has the table commutation cost for each context change between guests. The table size would not be a problem in comparison with hits size.

If this comment is applied, XenCC will be unusable. This method which consists in isolating guests from one another is also interesting to prevent reading of table to infer a basic map of the machine memory use (Fig. 10).

Conclusion

Any shared system not having the goal of sharing information between guests has to isolate them from each other. Any break of this rule would generate unforeseen effects. In the case which we have just seen, insulation has been set up, but some details of the implementation enable to bypass it. So, we have a kind of communication, which can be called covert channel, between several accomplice virtual machines. This "detail" could be considered as minor, but if we add tools like backdoors, the problem of communication becomes critical.

Xen is one of the most interesting and competitive virtualization systems. Its possibilities are growing continuously and the number of users also. However, we should not neglect security to the advantage of performance. Nowadays, some implementation "details", as I just explained, seem to interest only a small minority of Xen developers, which is definitely a pity. On the other hand, the visible aspect of security is on the right way and new functionalities such as *stubs domains* are very promising.

Inherently, covert channels will never absolutely be excluded from some environments. Indeed, to exclude this possibility, a system entirely isolated from the outside world would be needed, but having no means of communication, it would become useless. It is however possible to reduce significantly the use of covert channels by a careful understanding of this problem. With virtualizers, or more generally with operating systems, the data transfer control is a key condition for a full security policy.

References

1. Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I., Warfield, A.: Xen and the Art of Virtualization. University of Cambridge Computer Laboratory. http://www.cl.cam.ac.uk/research/srg/netos/papers/2003-xensosp.pdf (2003)
2. Chrisnall, D.: The Definitive Guide to the Xen Hypervisor. Upper Saddle River (2008)
3. Filiol, E.: Formalisation and implementation aspects of K-ary (malicious) codes. J. Comput. Virol. **3**(2), 179–185 (2007)
4. Kemmerer, R.A.: Shared resource matrix methodology: an approach to identifying storage and timing channels. ACM Trans. Comput. Syst. **1**(3), 256–277 (1983)
5. Lampson, B.W.: A Note on the Confinement Problem. Xerox Palo Alto Research Center (1973)
6. National Computer Security Center.: Department of Defense Trusted Computer System Evaluation Criteria. DoD 5200.28-STD (1985)
7. Ormandy, T.: An Empirical Study into the Security Exposure to Hosts of Hostile Virtualized Environments. Google, Inc. http://taviso.decsystem.org/virtsec.pdf (2007)
8. Tsai, C., Gligor, V.D., Chandersekaran, C.S.: A formal method for the identification of covert storage channels in source code. IEEE Symposium on Security and Privacy (1987)
9. U.S. Department of Defense.: Covert Channel Analysis Of Trusted Systems (Light Pink Book). The Rainbow Books. http://www.fas.org/irp/nsa/rainbow/tg030.htm (1993)
10. The Xen Team.: Xen Users' Manual. University of Cambridge, UK. http://www.cl.cam.ac.uk/research/srg/netos/xen/readmes/user/ (2008)
11. The Xen Team.: Xen Interface Manual. University of Cambridge, UK. http://www.cl.cam.ac.uk/research/srg/netos/xen/readmes/interface/ (2008)