Introducing Ring -3 Rootkits

Alexander Tereshkin and Rafał Wojtczuk

Black Hat USA, July 29 2009
Las Vegas, NV
1. Introducing **Ring -3**

2. **Getting** there

3. Writing useful **Ring -3 rootkits**
A Quest to Ring -3
Ring  3  Usermode rootkits
Ring  0  Kernelmode rootkits
Ring -1  Hypervisor rootkits (Bluepill)
Ring -2  SMM rootkits
Ring -3?
What is this?
Yes, it is a chipset (MCH)
(More precisely Intel Q35 on this picture)
Did you know it's also a standalone web server?
Many (all?) vPro chipsets (MCHs) have:

☑ An Independent CPU (not IA32!)
☑ Access to dedicated DRAM memory
☑ Special interface to the Network Card (NIC)
☑ Execution environment called Management Engine (ME)
Your chipset is a little computer. It can execute programs in parallel and independently from the main CPU!
Where is the software for the chipset kept?
On the SPI-flash chip (the same one used for the BIOS code)

It is a separate chip on a motherboard:
Of course one cannot reflash the SPI chip at will! vPro-compatible systems do not allow unsigned updates to its firmware (e.g. BIOS reflash).
But see our talk tomorrow about breaking into the Intel BIOS ;)}
Anyway:

- The chipset runs programs.
- The programs are stored in the (well protected) flash memory, together with BIOS firmware.
So, what programs run on the chipset?
Intel Active Management Technology (AMT)

http://www.intel.com/technology/platform-technology/intel-amt/
### Processor Information

**Processor 1**

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Manufacturer</td>
<td>Intel(R) Corporation</td>
</tr>
<tr>
<td>Family</td>
<td>Intel® Pentium® D Processor</td>
</tr>
<tr>
<td>Socket</td>
<td>J1PR</td>
</tr>
<tr>
<td>Version</td>
<td>Intel(R) Core(TM)2 Duo CPU E8400 @ 3.00GHz</td>
</tr>
<tr>
<td>ID</td>
<td>13829424153406539386</td>
</tr>
<tr>
<td>Maximum socket speed</td>
<td>4000 MHz</td>
</tr>
<tr>
<td>Speed</td>
<td>3000 MHz</td>
</tr>
<tr>
<td>Status</td>
<td>Enabled</td>
</tr>
<tr>
<td>Upgrade method</td>
<td>Unknown</td>
</tr>
<tr>
<td>Populated?</td>
<td>Yes</td>
</tr>
</tbody>
</table>
Remote Control

Power state: On
Send a command to this computer:

- Turn power off*
- Cycle power off and on*
- Reset*

Select a boot option:

- Normal boot
- Boot from local hard drive

*Caution: These commands may cause user application data loss.

Send Command
Connect & Control

In this window, you can connect to an Intel® AMT computer. Once connected, you can control the computer remotely, remotely turn on or off the Intel® AMT computer you are connected to, control network policies and filters, boot the computer to a remote drive, view the hardware asset inventory, and read the computer’s event log.

Connection | Remote Control | Intel® Management Engine | Networking

- Intel® Management Engine
- Computer Hostname / Domain: iDB0.somedomain.org
- Intel® AMT Version: 3.2.1
- Intel® Management Engine Peer
- User Accounts
- Interaction Type
- Certificate & CRL Store: 0 certificate(s), 0 trusted root(s)
- Kerberos® Setup
- Remote Access

Set Time...  WSMAN Browser...
Connect & Control

In this window, you can connect to an Intel® AMT computer. Once connected, you can control the computer remotely.

Manageability Terminal Tool - 192.168.0.22

Serial-over-LAN - Connected

ISOLINUX 3.61 2008-02-03 Copyright (C) 1994-2008 H. Peter Anvin

[1] RescueSystem
[2] RescueSystem - load cd into RAM
[3] memtest86

boot:
Loading /isolinux/vmlinuxx.........................
Loading initrdx.................................
If abused, AMT offers powerful backdoor capability: it can survive **OS reinstall** or other OS change!
But AMT is turned off by default...
There are a few methods to enable AMT...
... but most require physical presence during the BIOS boot.
We do have ideas how to do it remotely,
But let's skip it and talk about something better...
But turns out that some AMT code is executed *regardless of whether AMT is enabled* in BIOS or not! And we can hook this code (see later)!
Injecting Code into AMT/ME
Ok, so how we get our code executed inside AMT/ME environment?
Top Of Memory (TOM), e.g. 2GB

TOM - 16MB
Memory Remapping on Q35 chipset

This DRAM now accessible from CPU at physical addresses: `<REMAPBASE, REMAPLIMIT>` Otherwise it would be wasted!
Applying this to AMT case
remap_base = 0x100000000 (4G)
remap_limit = 0x183ffffff
 touud = 0x184000000
reclaim_mapped_to = 0x7c000000

AMT normally at: 0x7f000000,
Now remapped to: 0x1030000000 (and freely accessible by the OS!)

(Offsets for a system with 2GB of DRAM)
Fixed? No problem - just revert to the older BIOS!
(turns out no user consent is needed to downgrade Intel BIOS to an earlier version - malware can perfectly use this technique, it only introduces one additional reboot)
How about other chipsets?
This attack doesn't work against the Intel Q45-based boards. The AMT region seems to be additionally protected. (We are investigating how to get access to it...)
Writing Useful Ring -3 Rootkits
Justifying the "Ring -3" name
Independent of main CPU
Can access host memory via DMA (with restrictions)
Dedicated link to NIC, and its filtering capabilities
Can force host OS to reboot at any time (and boot the system from the emulated CDROM)
Active even in S3 sleep!
Unified execution environment
A few words about the ARC4 processor (integrated in the MCH)

- RISC architecture
- 32-bit general purpose registers and memory space
- "Auxiliary" registers space, which is used to access hardware
- On Q35 boards, the $0x01000000$–$0x02000000$ memory range (of the ARC4 processor) is mapped to the top 16MB of host DRAM
The ARC compiler suite (arc-gnu-tools) used to be freely available (a few months ago)...

Now it seems to be a commercial product only:

http://www.arc.com/software/gnutools/

(we were luckily enough to download it when it was still free)
Better portability between different hardware than SMM rootkits
(Unified ARC4 execution environment)
Getting our code periodically executed
Executable modules found in the AMT memory dump:
(names and numbers taken from their headers)

<table>
<thead>
<tr>
<th>Module</th>
<th>Base Address</th>
<th>Code Address</th>
<th>Entry Point Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOADER</td>
<td>0x000000..0x0122B8</td>
<td>0x000050..0x0013E0</td>
<td>0x000050</td>
</tr>
<tr>
<td>KERNEL</td>
<td>0x0122D0..0x28979C</td>
<td>0x012320..0x05F068</td>
<td>0x031A10</td>
</tr>
<tr>
<td>PMHWSEQ</td>
<td>0x2897B0..0x28DDF0</td>
<td>0x289800..0x28CAD8</td>
<td>0x28A170</td>
</tr>
<tr>
<td>QST</td>
<td>0x28DE00..0x2A79E8</td>
<td>0x28DE50..0x29B3F4</td>
<td>0x291B48</td>
</tr>
<tr>
<td>OS</td>
<td>0x2A7A00..0x88EE28</td>
<td>0x2A7A50..0x5ADA48</td>
<td>0x4ECC58</td>
</tr>
<tr>
<td>ADMIN_CM</td>
<td>0x88EE40..0x98CCF8</td>
<td>0x88EE90..0x91A810</td>
<td>0x8B2994</td>
</tr>
<tr>
<td>AMT_CM</td>
<td>0x98CD10..0xAA35FC</td>
<td>0x98CD60..0xA2089C</td>
<td>0x9BB964</td>
</tr>
<tr>
<td>ASF_CM</td>
<td>0xAA3610..0xAB4DEC</td>
<td>0xAA3660..0xAAD59C</td>
<td>0xAABC58</td>
</tr>
</tbody>
</table>
This function from the KERNEL module is called quite often probably by a timer interrupt handler.
Accessing the host memory
Programming internal DMA hardware in JTAG debugger to copy 64 bytes from 0x73000 host phys addr to internal memory

DMA-ed malicious VM Exit handler

source: Yuriy Bulygin, Intel, Black Hat USA 2008
AMT code can access host memory via DMA
But how to program it? Of course this is not documented anywhere...
(And the rootkit can't just use ARC4 JTAG debugger, of course)
Idea of how to learn how AMT code does DMA to host memory
We know that AMT emulates "Virtual CDROM" that might be used by remote admin to boot system into OS installer...
...we can also debug the AMT code using function hooking and counters...
Our debugging stubs
(The counter_* variables are also located in the AMT memory -- we read them using the remapping trick)

Most of the functions can be spotted by looking for the following prologue signature:

```
04 3E 0E 10  st blink, [sp+4]
```
So we can boot off AMT CDROM e.g. a Linux OS and try to access the AMT virtual CDROM...
...at the same time we trace which AMT code has been executed.
Q: How is the AMT CDROM presented to BIOS/OS?
A: As a PCI device...
00:03.2 IDE interface: Intel Corporation PT IDER Controller (rev 02) (prog-if 85 [Master Sec0 Pri0])
Subsystem: Intel Corporation Unknown device 4f4a
Flags: bus master, 66MHz, fast devsel, latency 0,

IRQ 9
I/O ports at 2480 [size=8]
I/O ports at 24a4 [size=4]
I/O ports at 2478 [size=8]
I/O ports at 24a0 [size=4]
I/O ports at 2440 [size=16]
Capabilities: [c8] Power Management version 3
Capabilities: [d0] Message Signalled Interrupts:
Mask- 64bit+ Queue=0/0 Enable-
We have traced BIOS accesses to AMT CDROM during boot; it turned out that BIOS did not use DMA transfers, it used PIO data transfers :(
Fortunately, the above PCI device fully conforms to ATAPI specifications; as a result, it is properly handled by the Linux `ata_generic.ko` driver

(if loaded with `all_generic_ide` flag)
f9q35 kernel: ACPI: PCI Interrupt 0000:00:03.2[C] -> GSI 18 (level, low)
        -> IRQ 18
f9q35 kernel: scsi6 : ata_generic
f9q35 kernel: scsi7 : ata_generic
f9q35 kernel: ata7: PATA max UDMA/100 cmd 0x2480 ctl 0x24a4 bmdma 0x2440 irq 18
f9q35 kernel: ata8: PATA max UDMA/100 cmd 0x2478 ctl 0x24a0 bmdma 0x2448 irq 18
f9q35 kernel: ata7.00: ATAPI: Intel Virtual LS-120 Floppy UHD Floppy
        , 1.00, max UD
f9q35 kernel: ata7.01: ATAPI: Intel Virtual CD, 1.00, max UDMA/100
f9q35 kernel: ata7.00: configured for UDMA/100
f9q35 kernel: ata7.01: configured for UDMA/100
f9q35 kernel: scsi 6:0:0:0: Direct-Access Intel Virtual Floppy
        1.00 PQ: 0 A
f9q35 kernel: sd 6:0:0:0: [sdb] Attached SCSI removable disk
f9q35 kernel: sd 6:0:0:0: Attached scsi generic sg2 type 0
f9q35 kernel: scsi 6:0:1:0: CD-ROM Intel Virtual CD
        1.00 PQ: 0 A
[root@f9q35 ~]#
[root@f9q35 ~]#
[root@f9q35 ~]#
[root@f9q35 ~]#
We can instruct `ata_generic.ko` whether to use or not DMA for the virtual CDROM accesses → we can do the **diffing** between two traces and find out which AMT code is responsible for DMA :)
This way we found (at least one) way to do DMA from AMT to the host memory
struct dmadesc_t {
    unsigned int src_lo;
    unsigned int src_hi;
    unsigned int dst_lo;
    unsigned int dst_hi;
    unsigned int count;
    unsigned int res1;
    unsigned int res2;
    unsigned int res3;
} dmadesc[NUMBER_OF_DMA_ENGINES];

void dma_amt2host(unsigned int idx, /* the id of DMA engine */
    unsigned int amt_source_addr,
    unsigned int host_dest_addr,
    unsigned int transfer_length) {

    unsigned int srbase = 0x5010 + 4 * idx;
    memset(&dmadesc[idx], 0, sizeof dmadesc[idx]);
    dmadesc[idx].src_lo = amt_source_addr;
    dmadesc[idx].dst_lo = host_dest_addr;
    dmadesc[idx].count = transfer_length;
    sr(srbase + 1, &dmadesc[idx]);
    sr(srbase + 2, 0);
    sr(srbase + 3, 0);
    sr(srbase + 0, 0x189);
}
10.5.2 **CMD—Command Register**

- **B/D/F/Type:** 0/3/2/PCI
- **Address Offset:** 4-5h
- **Default Value:** 0000h
- **Access:** RO, R/W
- **Size:** 16 bits

**Reset:** Host System reset or D3->D0 transition of function.

This register provides basic control over the device's ability to respond to and perform Host system related accesses.

---

### Intel® Manageability Engine Subsystem Registers

<table>
<thead>
<tr>
<th>Bit</th>
<th>Access</th>
<th>Default Value</th>
<th>RST/ PWR</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>R/W</td>
<td>0b</td>
<td>Core</td>
<td><strong>Bus Master Enable (BME):</strong> This bit controls the PT function's ability to act as a master for data transfers. This bit does not impact the generation of completions for split transaction commands.</td>
</tr>
<tr>
<td>1</td>
<td>RO</td>
<td>0b</td>
<td>Core</td>
<td><strong>Memory Space Enable (MSE):</strong> PT function does not contain target memory space.</td>
</tr>
<tr>
<td>0</td>
<td>R/W</td>
<td>0b</td>
<td>Core</td>
<td><strong>I/O Space enable (IOSE):</strong> This bit controls access to the PT function's target I/O space.</td>
</tr>
</tbody>
</table>

source: intel.com
Unfortunately, upon reboot, the **BME bit** for IDER device is cleared, which prevents DMA transfers...

However: rootkit can detect that a host reboot is in progress (because DMA transfers fail to work), and force reboot to AMT CDROM, that will set BME bit and resume OS boot.
Possibly, using other ME PCI device bypasses the BME limitation?

(there is nothing about BME bit in Yuriy Bulygin's talk on DeepWatch from BH US 2008)

This would allow for SRTM bypass (AMT could inject/replace already-measured code while it's executing)

But we haven't found any other way to do DMA without BME so far...
Putting it all together
Host OS (e.g. Windows)

Hypervisor (optional)

SMM

Chipset ME/AMT:
All code executed by the chipset's ARC4 processor, even if the host in sleep mode!

Host Memory:
All code executed on the host CPU(s)

Hooked AMT function that is executed periodically (regardless of whether AMT is enabled or not in the BIOS)

AMT rootkit

DMA access
What about VT-d? Can the OS protect itself against AMT rootkit?
DMA REMAPPING

• VT-d capable chipsets have one or more DMA-remapping engines virtualizing Directed I/O access [12]
• Internal devices are also a subject to DMA-remapping
• Chipset has dedicated register-set for each DMA-remap unit accessible by software as MMIO range which software can use to protect certain memory regions from certain I/O devices
• Rootkit can create DMA-remapping page tables to translate addresses of DMA requests issued by embedded uC (identified by its PCI B/D/F) to different host physical addresses
  – or read/write protect entries in DMAr pages tables
  – or mark context-entry as not Present to cause translation fault
  – or enable PLMR/PHMR DMA-protected regions to prevent any DMA
• And relocate code/data (VMExit handler, VMCS ..) to memory protected by DMAr page tables or to PMR regions
SO WHAT CAN WE DO ABOUT THIS ??

• DMA-remapping unit can distinguish DMA requests issued by DeepWatch internal device function inside embedded uC
• by its requester id from DMA requests issued by other internal functions
• and not translate them
• Or disable and lock DMA-remapping of DeepWatch device function if DeepWatch is used
• And allow only trusted software like SMX authenticated code modules (Intel® TXT) to enable and program DMA-remap engine for DeepWatch
So, if Intel allowed its AMT/ME code to bypass VT-d (in order to allow rootkit detectors in the chipset), then our AMT rootkit would automatically gain ability to bypass VT-d as well!
We have verified that Xen 3.3+ uses VT-d in order to protect its own hypervisor and consequently our AMT rootkit is not able to access this memory of Xen hypervisor.

(But still, if ME PCI devices are not delegated to a driver domain, then we can access dom0 memory)
Powerful it is, the VT-d
Still, an AMT rootkit can, if detected that it has an opponent that uses VT-d for protection, do the following:

- Force OS reboot
- Force booting from Virtual CDROM
- Use its own image for the CDROM that would infect the OS kernel (e.g. xen.gz) and disable the VT-d there
How to protect against such scenario?
Via Trusted Boot, e.g. SRTM or DRTM (Intel TXT)
(Keep in mind that we can bypass TXT though, if used without STM, and there is still no STM available as of now)
Final Thoughts
We do like many of the new Intel technologies (VT-x, VT-d, TXT), ...
But AMT is different in that it can potentially be greatly abused by the attacker
(VT-d or TXT can potentially be bypassed, but they cannot help the attacker!)
But keep in mind that our attack doesn't work on the latest Q45 chipsets - a sign that Intel treats the security seriously...
You do not want this privileged code to fall into enemy's hand, do you?

source: http://freemasonry.bcy.ca/anti-masonry/
ITL

INVISIBLE THINGS LAB

http://invisiblethingslab.com