Closer to metal: Reverse engineering the Broadcom NetExtreme's firmware

Guillaume Delugré
Sogeti / ESEC R&D
guillaume(at)security-labs.org

HITB 2011 - Amsterdam

Purpose of this presentation

Hardware trust?

- Hardware manufacturers are reluctant to disclose their specifications
- You do not really know what firmwares do behind your back
- Consequently you cannot really trust them...

Previous works

- **A SSH server in your NIC**, Arrigo Triulzi, PacSec 2008
- **Can you still trust your network card?**, Y-A Perez, L. Duflot, CanSecWest 2010
- **Reversing the Broadcom NetExtreme firmware**, G. Delugre, Hack.lu 2010
- **Runtime Firmware Integrity Verification: What Can Now Be Achieved**, Y-A Perez, L. Duflot, CanSecWest 2011

Purpose of this presentation

What is this presentation about?

- Reverse engineering of the Broadcom Ethernet NetExtreme firmware
- Building an instrumentation toolset for the device
- Developing a new firmware from scratch

Why?

- To have a better understanding of the device internals
- To look for vulnerabilities inside the firmware code
- To develop an open-source alternative firmware for the community
- To develop a rootkit firmware embedded in the network card!

Plan

Where should we begin?

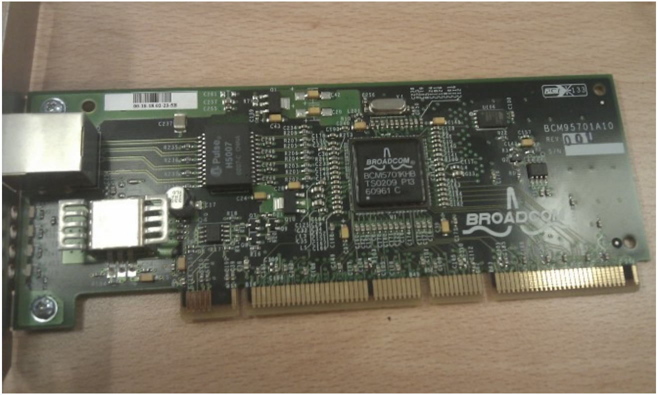
About the target

- Targeted hardware: Broadcom Ethernet NetExtreme NIC
- Standard range of Ethernet cards family from Broadcom
- Massively installed on personal laptops, home computers, enterprises. . .

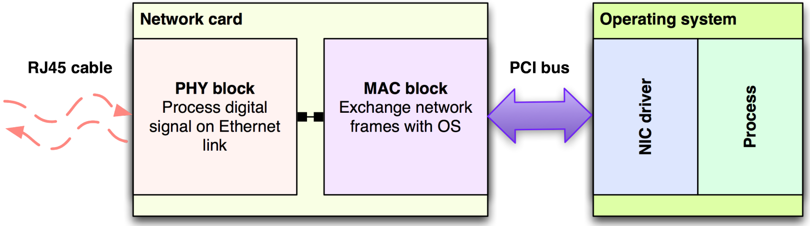
Sources

- Broadcom device specifications (incomplete, sometimes erroneous)
- Linux open-source kernel module (tg3)
- A firmware code is published as a binary blob in the kernel tree
- It is actually not loaded by the Linux driver

The targeted device



NIC overview



Device overview

Core blocks

- The PHY block
  - DSP on the Ethernet link
  - Passes raw data to the MAC block
- The MAC block
  - Processes and queues network frames
  - Passes them to the driver

MAC components

- one or two MIPS CPU
- a non-volatile EEPROM memory
- a volatile SRAM memory
- a set of *registers* to configure the device

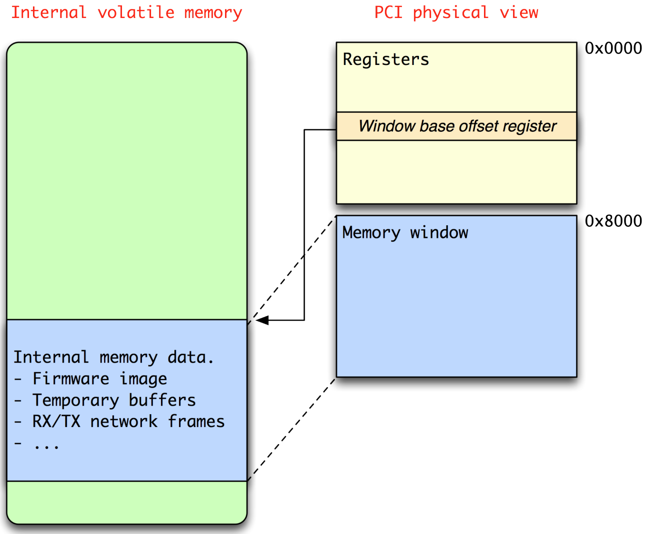
Communicating with the device

PCI interface

- Cards are connected to the **PCI bus**
- Device is accessible using memory-mapped I/O
- Mapped on 16 bits (64 KB)
  - First 32 KB are a direct mapping onto the device registers
  - Last 32 KB constitute a R/W window into the internal volatile memory
  - The base of the window can be set using a register
- EEPROM memory can be accessed in R/W using a dedicated set of registers

We have access to registers, volatile and EEPROM memory through the PCI bus.

Physical PCI view



Different kinds of memory

EEPROM

- Manufacturer's information, MAC address, . . .
- Firmware images
- **Non-documented** format

Volatile memory

- Copy of the firmware image executed by the CPU
- Network packet structures, temporary buffers

Registers

- **MANY** registers to configure and control the device
- Some of them are non-documented

Plan

Instrumenting the device

We want to

- Get easy access to all kinds of memory
- Dump the executing firmware code
- Inject and execute some code
- Test it
- Debug it

At first we have to easily access the device's memory, so we are going to write a little **kernel module**.

Plan

Linux Kernel Module

Basics

- At boot time, the BIOS assigns each device a physical memory range
- The OS maps this range onto a virtual address range
- In MMIO mode, we have to get the device's base virtual address then just access it like any other memory

A kernel proxy between the NIC and userland

- The module provides primitives for reading and writing inside the NIC (registers, volatile, EEPROM)
- It exposes them to userland by creating a virtual char device
- Processes can then use `open`, `read`, `write`, `seek` syscalls

Extracting the firmware code

Firmware dump

- We can dump the executed firmware code from userland
- Based at address `0x10000` in volatile memory (refering to the specs)
- We can directly disassemble MIPS code, obviously it is not encrypted, nor obfuscated

Static analysis

- Static disassembly analysis already made possible
- **We will focus on how to dynamically analyze the executed code**

Plan

Going further

Plan

- Using this kernel proxy, we can easily dump and modify the device's memory from userland
- Now we have to control what is executed on the NIC, the firmware code

Two firmware debuggers

InVitroDbg is a firmware emulator based on a modified Qemu

InVivoDbg is a real firmware debugger to control code executed on the NIC

Both use the kernel proxy to interact with the NIC.

InVitroDbg

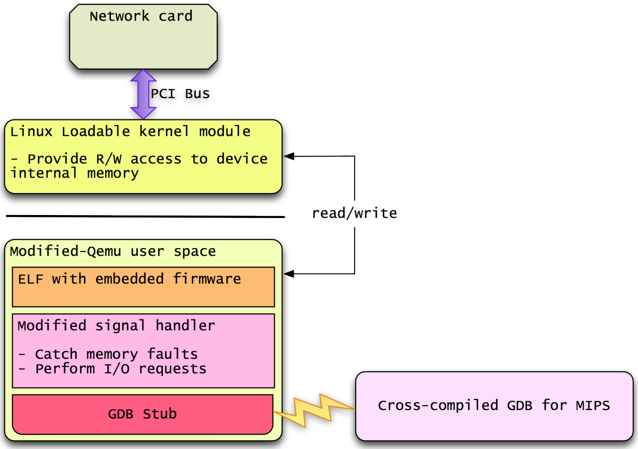
A firmware emulator

- Emulates the NIC MIPS CPU
- Interacts with the physical NIC memory

Mechanism

- Based on a modified Qemu
- Firmware code embedded in a userland ELF executable
- Code segment mapped at the firmware base address
- Catches memory faults and redirects accesses to the real device
- Debugging made possible using the GDB stub of Qemu

Architecture de InVitroDbg



InVitroDbg

InVitro

- Firmware code executed in userland
- No injection in the device memory
- Architecture can be reused for other devices
- A **lot** of transactions on the PCI bus
- Fake memory view from the PCI bus

InVivoDbg

Firmware debugger

- Firmware code really executed on the NIC
- Controlling the CPU using dedicated registers

Mechanism

- CPU control with NIC registers: `halt`, `resume`, `hbp`
- CPU registers found in non-documented NIC registers
- Debugger core written in Ruby
- Integrated with the Metasm dissassembly framework
- Real-time IDA-like graphical interface for debugging

InVivoDbg

InVivo

- IDA-like GUI
- Easily extensible with Ruby scripts
- Few PCI transactions required
- Real memory view from the NIC CPU

Extending InVivoDbg

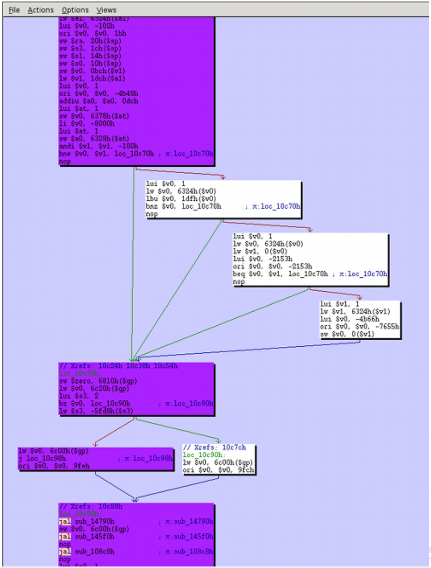
Execution flow tracing

- Reuse the Metasm plugin BinTrace (A. Gazet & Y. Guillot)
- Log every basic block executed
- Save a trace which can be visualized offline
- Support differential analysis of different traces

Interest

- Quickly visualize the default execution path of the code
- Monitor the effect of various stimuli (received packet, driver communication. . .) on execution

Execution flow trace



Extending InVivoDbg

Memory access tracing

- Step-by-step firmware code
- Log each memory access (`lw`, `sw`, `lh`, `sh`, `lb`, `sb`)
- Save the generated trace
- Replay the trace

Interest

- Does not rely on firmware code analysis
- Extracts the very core behavior of the firmware
- Logs every register access tells us what the firmware is actually doing, e.g. how it configures the device

Memory access trace

```
0x109c8: READ at address 0xc0000400
0x109f0: WRITE 0x00000012 at address 0xc000045c
0x109f8: WRITE 0x00000006 at address 0xc0000468
0x10a00: WRITE 0x00010000 at address 0xc0006800
0x10a08: WRITE 0x00000001 at address 0xc0005ce0
0x10a0c: WRITE 0x00000001 at address 0xc0005cc0
0x10a14: WRITE 0x00000001 at address 0xc0005cb0
```

Plan

Creating a new firmware: what for?

Multiple purposes

- Provides an open-source alternative to proprietary firmware
- Creates a rootkit firmware resident in the NIC
- Turns a network card into a physical memory dumper (forensics)

How to get code execution?

- Writing the firmware in memory and redirecting $pc
- **Writing the firmware in EEPROM so that it runs at bootstrap**
- We can then use the previous debuggers to debug our own code!

Plan

Understanding the EEPROM layout

EEPROM

- Contains non-volatile data
- Memory layout is **not documented** by Broadcom
- Layout uncovered by analyzing firmware code

Memory structure

- Bootstrap header
- Device metadata (revision, manufacturer's id)
- Device configuration (MAC address, power, PCI config, ...)
- **Firmware images**
- Each structure is followed by a CRC32

Description of the bootstrap process

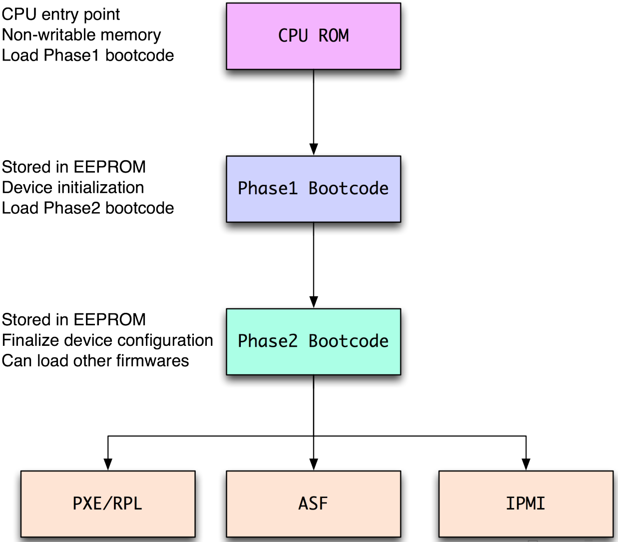
Firmware bootstrap

- How is the firmware loaded from EEPROM to volatile memory ?
- Method: reset the device and stop the CPU as quick as possible!
- Result: CPU executes code at unknown address 0x4000_0000

So?

- This memory zone is **execute-only** (not read/write), probably a ROM
- Hack: An non-documented device register holds the current dword pointed by $pc
- **We can dump the ROM** by modifying $pc and polling this register!

Description of the bootstrap process



Description of the bootstrap process

No trusted bootstrap sequence!

Bootstrap

Every time the source power is plugged-in, or a PCI reset signal is issued, or the reset register is set:

1. CPU starts on a **boot ROM**
   - Initializes EEPROM access
   - Loads bootstrap firmware in memory from EEPROM
2. Execution of the **bootstrap firmware**
   - Configures the core of the device (power, clocks...)
   - Loads a second-stage firmware from EEPROM
3. Execution of the **second-stage firmware**
   - Sets up networking (Ethernet link, MAC, ...)
   - Can load another firmware if requested
   - Tells the driver the device is ready

Developing your own firmware

Coding environment

All we need is

- A cross-compiled `binutils` for MIPS
- We can start developing our firmware in C
- Inject our firmware in the EEPROM

CPU memory mapping

- Volatile memory is accessible from address `0`
- Memory greater than `0xC000_0000` maps into device registers

Developing your own firmware

Size requirements

- Code can reside between `0x10000` and `0x1c000`
- 48 KB memory shared by code, stack, and incoming packet buffers

Firmware initialization

- Initialize the stack pointer
- Configure the device for working (PHY/MAC init)
- Then you can add whatever feature you wish

Plan

Network connectivity

Networking capability

- It is active on the network even if the machine is shut down
- It can listen for incoming packets and forge new packets
- But first it needs to detect network configuration (our own IP address, router address, DNS. . .)

Dynamic network configuration detection

- Embeds a very light DHCP client
- If no DHCP, tries to catch DNS packets
  - contain router MAC, DNS server IP and our own IP
- Everything can be sent using a fake MAC address

Direct Memory Access

DMA

- PCI supports *Direct Memory Access*
- The NIC transfers frames from/to physical memory with DMA
- Arbitrary DMAs $\Rightarrow$ compromise the OS memory

How to do arbitrary DMA

1. Modify the physical address where packets are read/written
2. Modify the packet contents in the device memory on-the-fly
3. Force the device to operate a network operation (recv/send)
4. An arbitrary read/write to physical memory is then triggered

Actually **MUCH** more complicated in practice, but this is the idea

Counter-measures

Counter-measures

- Rootkit is active before the system boot
  - → Use a trusted boot technology, like Intel TXT
- Rootkit can corrupt kernel code
  - → Use an IOMMU technology, like Intel VT-d
- Qubes seems to make use of these features
- Also check Loic Duflot & Y-A. Perez talk about runtime firmware integrity verification (CSW 2011)

Plan

Forensics

Using the NIC for forensics purpose

1. The target system is up and running
2. The NIC is hotplugged on a free PCI slot
3. The device is powered up and the firmware starts
4. The whole physical memory is dumped over the Gigabit link

Device base address

- Our device has no base address (normally assigned by BIOS)
- We cannot safely retrieve the PCI-bridge physical address
- Hopefully we don't need one, all DMA transactions are initiated by the NIC

Forensics

Getting DMA to work

- OS will not crash if we prevent any interrupts to spawn
- The firmware has to configure the NIC as would do the driver
- We need to write structures in memory for DMAs to work...
  - ...but we cannot taint physical memory (forensics)
  - ...and we cannot use the NIC memory (no base address)
- So I use the VGA framebuffer as a temporary memory zone
  - It has a fixed base address (`0xa0000`)
  - Just a few pixels needed
  - Safe as long as nothing moves above these pixels

This is still a work in progress, no operational demo yet

Conclusion

In a nutshell...

- Reverse engineering of a proprietary firmware for security purpose
  - Made possible with a few free open-source tools (Qemu, Ruby, Metasm, binutils, ...)
  - Real-time firmware debugging!
  - But depends on targeted device (here Broadcom NICs)
- No firmware signature/encryption in Broadcom Ethernet NICs
- One can build and load its own firmware
  - To offer an open-source alternative for the community
  - To build a stealthy rootkit embedded in the NIC
  - To turn a NIC into a high-speed physical memory dumper

Thank you for your attention!

Questions?