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

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

HACK.LU 2010 - Luxembourg
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...
- So here comes the need for reverse engineering

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
### 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

1. Overview of the NIC architecture
2. Instrumenting the network card...
3. ...and developing a new firmware
### 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
Overview of the NIC architecture
Instrumenting the network card...
...and developing a new firmware

The targeted device
Overview of the NIC architecture

Instrumenting the network card... and developing a new firmware

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

Internal volatile memory

- Internal memory data.
  - Firmware image
  - Temporary buffers
  - RX/TX network frames
  - ...

PCI physical view

- Registers
  - Window base offset register
- Memory window

0x0000

0x8000
Different kinds of memory

**EEPROM**
- Manufacturer’s information, MAC address, ...
- Firmware images
- **Non-documentated** 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-documentated
Plan

1. Overview of the NIC architecture
2. Instrumenting the network card... 
3. ...and developing a new firmware
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

1. Overview of the NIC architecture

2. Instrumenting the network card...
   - Accessing the device's internal memory
   - Getting to debug firmware code

3. ...and developing a new firmware
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

1. Overview of the NIC architecture

2. Instrumenting the network card...
   - Accessing the device's internal memory
   - Getting to debug firmware code

3. ...and developing a new firmware
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
Overview of the NIC architecture
Instrumenting the network card...
...and developing a new firmware

Accessing the device's internal memory
Getting to debug firmware code

Architecture de InVitroDbg

Network card

PCI Bus

Linux Loadable kernel module
- Provide R/W access to device internal memory

read/write

Modified-Qemu user space
- ELF with embedded firmware
- Modified signal handler
  - Catch memory faults
  - Perform I/O requests

GDB Stub

Cross-compiled GDB for MIPS

G. Delugré
Closer to metal: Reverse engineering the Broadcom NetExtreme's firmware
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 disassembly framework
- Real-time IDA-like graphical interface for debugging
## Debuggers comparison

### InVitro
- Firmware code executed in userland
- No injection in the device memory
- A lot of transactions on the PCI bus
- Fake memory view from the PCI bus

### 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
Overview of the NIC architecture
Instrumenting the network card... and developing a new firmware

Accessing the device's internal memory
Getting to debug firmware code

Execution flow trace

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

1. Overview of the NIC architecture
2. Instrumenting the network card...
3. ...and developing a new firmware
Creating a new firmware: what for?

Multiple purposes

- Provides an open-source alternative to proprietary firmware
- Creates a rootkit firmware resident in the NIC

Code testing

- We control the firmware image in volatile memory
- We control the MIPS CPU state
- Thus we can quickly inject and test code in memory
- **How to get our code loaded during the device bootstrap?**
### Plan

1. **Overview of the NIC architecture**

2. **Instrumenting the network card...**

3. **...and developing a new firmware**
   - Reversing the EEPROM format
   - Description of the bootstrap process
   - Building your own firmware
Reversing the EEPROM format

Non-documented format

- EEPROM contains non-volatile data
- Data is r/w accessible using specific device registers
- Format is **not documented** by Broadcom spec. sheets

Contents

- Discovered by firmware code and memory analysis
- It contains
  - A bootstrap header
  - Device metadata (manufacturer’s id, device revision...)
  - Device configuration (MAC, voltages, ...)
  - A set of firmware images (bootstrap code, default image, PXE...)

G. Delugré
Closer to metal: Reverse engineering the Broadcom NetExtreme’s firmware
Plan

1. Overview of the NIC architecture
2. Instrumenting the network card...
3. ...and developing a new firmware
   - Reversing the EEPROM format
   - Description of the bootstrap process
   - Building your own firmware
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 address 0x4000_0000

So?

- This memory zone is **execute-only** (not read/write), probably a ROM
- Hack: An non-documentated 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

CPU entry point
Non-writable memory
Load Phase1 bootcode

CPU ROM

Phase1 Bootcode

Stored in EEPROM
Device initialization
Load Phase2 bootcode

Stored in EEPROM
Finalize device configuration
Can load other firmwares

Phase2 Bootcode

PXE/RPL
ASF
IPMI
**Description of the bootstrap process**

No trusted bootstrap sequence!

**Bootstrap**

Every time the source power is plugged-in, or a PCI reset 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**
   - Is the default firmware executed
   - Configures networking (Ethernet link, MAC, ...)
   - Can load another firmware if requested
Plan

1. Overview of the NIC architecture
2. Instrumenting the network card...
3. ... and developing a new firmware
   - Reversing the EEPROM format
   - Description of the bootstrap process
   - Building your own firmware
Developing your own firmware

Building our own firmware

All we need is

- A cross-compiled binutils for MIPS
- ld-scripting to map the firmware at 0x10000
- We can start developing our firmware in C
- Inject our firmware in the EEPROM

Memory mapping

- Memory view from the CPU is documented in the specs
- Volatile memory is accessible from address 0
- Memory greater than 0xC000_0000 maps into device registers
Developing our own firmware

**Size requirements**

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

**Firmware structure**

- Initialize the stack ($sp = 0x1c000$)
- Configure the device for working (way far beyond this talk)
- Perform custom malicious/fun actions from the NIC!
Examples of customized firmware

Remote firmware debugger
- Remote debugging using the Ethernet link
- Would offer debugging even if the machine is shut down

Rootkit capabilities
- Rootkit (still in development)
- Take over the network
  - Packet interception/forge by the firmware
  - Embedding an IP/UDP stack and a light DHCP client
  - → Stealthy communication (OS never aware)
- Corrupt physical memory
  - Reuse DMA capabilities over PCI to corrupt system RAM
  - Write access OK, read access still unstable
- The device and the OS driver still have to work properly!
### 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 highly stealthy rootkit embedded in the NIC
Questions?