

# Run-time firmware integrity verification: what if you can't trust your network card?

International Symposium on Recent Advances in Intrusion Detection  
Menlo Park, September 2011

Loïc Dufлот, Yves-Alexis Perez, Benjamin Morin

ANSSI

French Network and Information Security Agency  
firstname.lastname@ssi.gouv.fr



# Embedded devices

- ▶ Embedded systems are increasingly prevalent in computers
  - ▶ Network cards, hard drive controllers, chipsets, basebands, etc.
  - ▶ Some have high processing capabilities (“smart” devices)
  - ▶ They act as black-box execution environments
- ▶ They constitute a potential threat for the platforms’ security
  - ▶ They have access to sensitive information
  - ▶ They generally lack the protections available on standard processors (e.g., an MMU)
  - ▶ They potentially run with high privileges w.r.t the main operating system

## Attacks against embedded devices

- ▶ Vulnerabilities were found in several embedded software and firmware in the past few years :

Basebands	Weinmann [13]
Network controllers	Duflot and Perez [4], Triulzi [12], Delugré [3]
Keyboard controllers	Chen [2], Gazet [6]
Chipsets	Ortega and Sacco [9]

- ▶ Defending a system against such attacks is difficult because firmware are running out of the scope of the operating system.
- ▶ Existing IDSEs have probably overlooked these attacks
  - ▶ they mainly monitor the operating system and applications
  - ▶ those attacks are still quite new

## Example from our own proof-of-concept attack

- ▶ In [4], we demonstrated how it is possible for an attacker to take full control of a computer
  - ▶ by exploiting a vulnerability in the network adapter, and
  - ▶ adding a back-door in the OS kernel using DMA accesses ;
  - ▶ the back-door opens a reverse shell when the kernel processes an ICMP message with a particular type.
- ▶ Our proof-of-concept attack was based on a real world vulnerability
  - ▶ the vulnerability lied in the ASF remote administration function of the network adapter of the target machine.
  - ▶ it was unconditionally exploitable when the ASF function was activated to any attacker that would be able to send UDP packets to the machine.

## Problems & existing solutions

Compromised device?

### Consequences

- system
  - ▶ full OS compromise
  - ▶ rootkit re-injection at startup

### Counter-measures

- system
  - ▶ I/O MMU (Intel  $Vt-d$  and AMD IOMMU)

## Problems & existing solutions

Compromised device?

### Consequences

- system**
  - ▶ full OS compromise
  - ▶ rootkit re-injection at startup
- platform**
  - ▶ attacks against other devices on the same buses [11]

### Counter-measures

- system**
  - ▶ I/O MMU (Intel *Vt-d* and AMD *IOMMU*)
- platform**
  - ▶ *PCI Express Access Control Services*

## Problems & existing solutions

Compromised device?

### Consequences

- system
  - ▶ full OS compromise
  - ▶ rootkit re-injection at startup
- platform
  - ▶ attacks against other devices on the same buses [11]
- network
  - ▶ silent data leak
  - ▶ stepping stone to attack the whole network silently
  - ▶ won't be blocked by firewalls on vulnerable machines

### Counter-measures

- system
  - ▶ I/O MMU (Intel *Vt-d* and AMD *IOMMU*)
- platform
  - ▶ PCI Express *Access Control Services*
- network
  - ▶ ?

# Firmware integrity checks

## We need to check the firmware's integrity

- ▶ **At load time**
  - ▶ Peripherals' firmware should be measured during a *trusted boot*
  - ▶ This can be achieved by means of a TPM
- ▶ **At runtime**
  - ▶ Our objective is to check that the firmware is running untampered
  - ▶ The operating system acts as the verifier of the network card's execution
  - ▶ We assume that the operating system is trusted



## Existing solutions

### Remote firmware attestation [7, 8] ?

- ▶ Based on a challenge-response protocol
  - ▶ The target computes a checksum over its entire memory
  - ▶ The verifier checks the correctness of the returned checksum

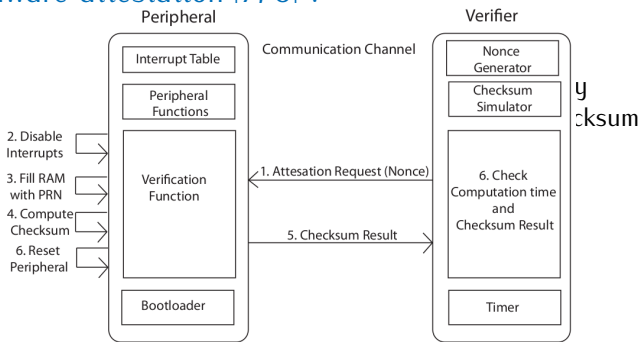
# Existing solutions

## Remote firmware attestation [7, 8] ?

- ▶ Based on

- ▶ The

- ▶ The



## Existing solutions

### Remote firmware attestation [7, 8] ?

- ▶ Based on a challenge-response protocol
  - ▶ The target computes a checksum over its entire memory
  - ▶ The verifier checks the correctness of the returned checksum
- ▶ Remote firmware attestation is difficult [1, 10, 5]
  - ▶ Severe constraints imposed by the checksum function execution
  - ▶ Is it really suitable for a network card?

## Existing solutions

### Remote firmware attestation [7, 8] ?

- ▶ Based on a challenge-response protocol
  - ▶ The target computes a checksum over its entire memory
  - ▶ The verifier checks the correctness of the returned checksum
- ▶ Remote firmware attestation is difficult [1, 10, 5]
  - ▶ Severe constraints imposed by the checksum function execution
  - ▶ Is it really suitable for a network card?

### Software symbiotes ?

- ▶ might be an interesting solution, requires further investigations
- ▶ seems quite intrusive

# What is a network card?

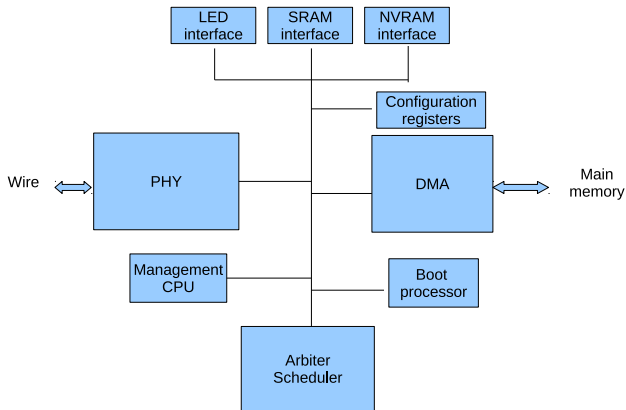
## Quite simple in theory

- ▶ transfer ethernet frames from the host to the wire
- ▶ and vice versa

## Increasingly complex in practice...

- ▶ advanced capabilities (PXE, TSO...)
- ▶ platform administration functions
- ▶ active even when OS is down or absent
- ▶ runs a firmware on embedded CPU

# What is (graphically) a network card?



## Intrusion detection model

- ▶ Our detection method is anomaly-based
  - ▶ The model of normal behavior is based on the NIC's memory layout
  - ▶ The memory profile is built empirically, by means of the observed NIC's memory accesses during "normal" network sessions
  - ▶ Memory areas used to execute code, read and/or write data are distinguished in the model
  - ▶ The card is run in step-by-step (debug) mode during detection
  - ▶ Any memory access that is outside the NIC's memory profile is interpreted as an attempt to divert the firmware's control flow
- ▶ Heuristics used to detect anomalous memory accesses
  - ▶ Step-by-step instruction comparison
  - ▶ Step-by-step instruction address checking
  - ▶ Shadow return stack

## Detection heuristics (1/2)

### Step-by-step instruction comparison

- ▶ Basically consists in checking that the instruction that is to be run is the same as the reference model's one
- ▶ This technique only works if the code is not self modifying (which is the case for the firmware we are considering)

### Step-by-step address checking

- ▶ Basically consists in checking that the instruction pointer value is consistent
- ▶ The network card running code in the heap, in the stack or in the memory scratchpad is indicative of an anomaly



## Detection heuristics (2/2)

### Shadow call stack

- ▶ Basically consists in maintaining a copy of the call stack of the firmware on the host side
- ▶ On an identified CALL-like instruction, the return address is pushed on the shadow stack
- ▶ On an identified RET-like instruction, the return address is checked against the saved one

### Other heuristics (not implemented)

- ▶ Another heuristic could consist in searching for anything that meets the statistical profile of executable code in data area
- ▶ This detection technique is prone to false positives

## Network Adapter Verification and Integrity checking Solution

### We designed our verification framework, NAVIS

- ▶ based on card instrumentation
- ▶ implements detection heuristics
- ▶ prevents control flow modifications

### We used:

- ▶ Broadcom BCM5754 and BCM5755M network cards
- ▶ ASF firmware

### Preventing confusion:

From now on:

**CPU** is the main CPU on the motherboard, running the OS

**MIPS** is the management processor on the network controller

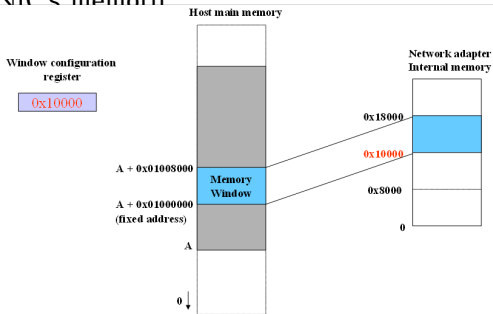
## Instrumenting the card

- ▶ Accessing NIC's memory
  - ▶ The NIC's internal memory is mapped in the main memory
  - ▶ This mechanism provides access to the firmware running on the NIC

# Instrumenting the card

▶ Accessing NIC's memory

- ▶ The
- ▶ This



memory  
window on the NIC

## Instrumenting the card

- ▶ Accessing NIC's memory
  - ▶ The NIC's internal memory is mapped in the main memory
  - ▶ This mechanism provides access to the firmware running on the NIC
- ▶ Identifying memory areas
  - ▶ Broadcom docs and drivers reveal that firmware files have three areas ([text](#), [data](#), [rodata](#))
  - ▶ But they do not provide the mappings for these area
  - ▶ The mappings can be identified by tracking the MIPS activity

## Instrumenting the card

- ▶ Accessing NIC's memory
  - ▶ The NIC's internal memory is mapped in the main memory
  - ▶ This mechanism provides access to the firmware running on the NIC
- ▶ Identifying memory areas
  - ▶ Broadcom docs and drivers reveal that firmware files have three areas ([text](#), [data](#), [rodata](#))
  - ▶ But they do not provide the mappings for these area
  - ▶ The mappings can be identified by tracking the MIPS activity
- ▶ NIC's internal registers
  - ▶ [state](#), [control](#) and [breakpoint](#) registers are accessible from the host

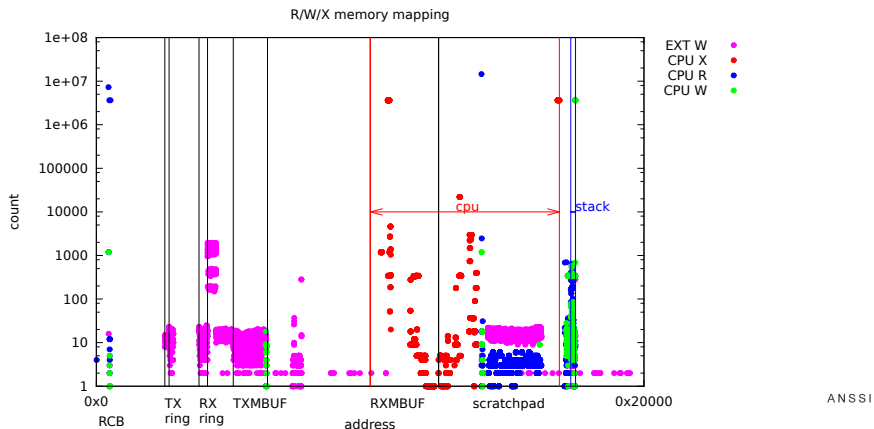
## Instrumenting the card

- ▶ Accessing NIC's memory
  - ▶ The NIC's internal memory is mapped in the main memory
  - ▶ This mechanism provides access to the firmware running on the NIC
- ▶ Identifying memory areas
  - ▶ Broadcom docs and drivers reveal that firmware files have three areas ([text](#), [data](#), [rodata](#))
  - ▶ But they do not provide the mappings for these area
  - ▶ The mappings can be identified by tracking the MIPS activity
- ▶ NIC's internal registers
  - ▶ [state](#), [control](#) and [breakpoint](#) registers are accessible from the host

This allows us to [monitor](#) the activity of the firmware, [detect](#) anomalous behaviours and [stop](#) the adapter when a problem is detected

# Network controller memory map

code exec *instructions executed by the MIPS*  
 MIPS writes *addresses written by the MIPS (SH, SB, SW)*  
 MIPS reads *addresses read by the MIPS (LH/LHU, LB/LBU, LW)*  
 other writes *network packets written to the card memory by DMA from host and by PHY from the wire*

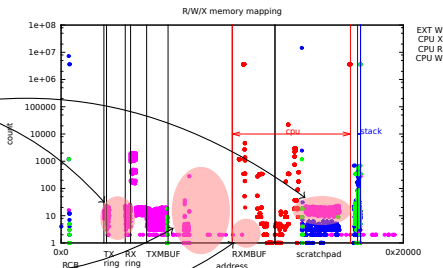




# Memory map analysis

## External writes

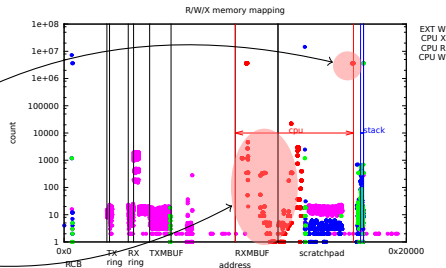
- ▶ in RX/TX rings and in the *scratchpad* area (traffic from and to the main host)
- ▶ in unmapped areas...
- ▶ inside the MIPS code area!



# Memory map analysis

## MIPS execution

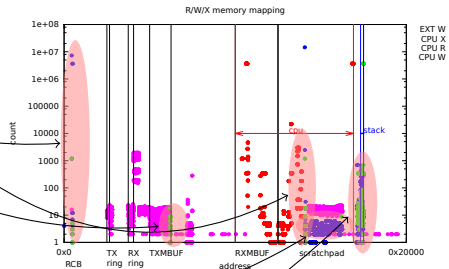
- ▶ mostly at the beginning of scratchpad plus few instructions at the end



# Memory map analysis

## MIPS reads/writes

- ▶ in the Rings Control Blocks and TXMBUF (presumably ASF traffic)
- ▶ in the *scratchpad*, with four different areas:
  - ▶ just after the main code area
  - ▶ in the packets area (presumably ASF traffic)
  - ▶ just after the MIPS code area
  - ▶ after the MIPS code area, just before *scrachpad* end (the stack)



# Challenges in monitoring the MIPS

- ▶ Control flow instructions
  - ▶ No CALL/RET instructions on MIPS architecture
  - ▶ Fortunately, the firmware code is rather simple
  - ▶ Function calls can reliably be inferred from jump instructions (JAL) and a specific register (R31) used to store return addresses
- ▶ Interrupts triggered by the network adapter
  - ▶ Cause unexpected changes in the control flow (looks like an attack)
  - ▶ The firmware uses a single interrupt handler, starting at a fixed address
  - ▶ Return from the interrupt vector is done through register R27
  - ▶ Dealing with interrupts is manageable by monitoring this behaviour

## Summary

- we know
  - ▶ where the MIPS reads and writes data
  - ▶ where the MIPS executes code
  - ▶ how to find function calls and RET
- issues
  - ▶ there are external writes to the MIPS code area
  - ▶ there are writes in **unmapped** areas
  - ▶ interrupts make it harder to follow control flow
- remarks
  - ▶ no way to enforce *rodata*
  - ▶ no such thing as NX on those MIPS processor
  - ▶ no segmentation/pagination

### This allows to detect any unexpected change in the control flow

- ▶ When a return value is modified on the stack
- ▶ But data on the stack, heap and scratchpad can still be modified by the attacker.

## It works!

We tried the various proof-of-concept attacks we created as part of previous researches

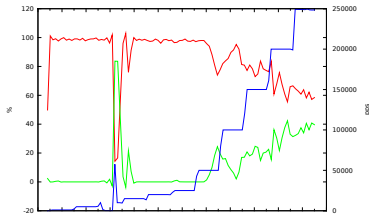
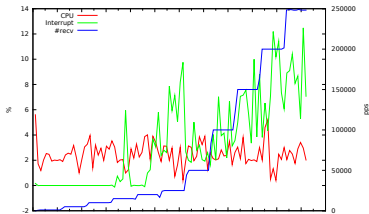
- ▶ they are all detected
- ▶ an alert is reported in NAVIS
- ▶ the network controller is stopped
- ▶ one can either reset the card and start fresh,
- ▶ or try to investigate in the debugger

Since we detect control flow modification, other attacks should be detected as well

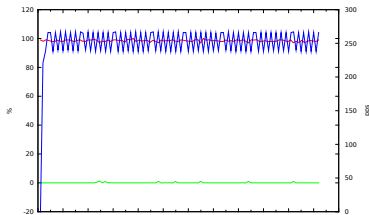
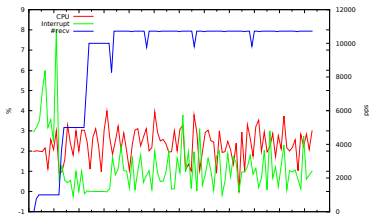
# Performance

- ▶ The monitoring is expected to have a negative impact on the NIC's performance
  - ▶ The MIPS CPU is run in step-by-step mode
  - ▶ Various tests are done at each MIPS cycle (*bounds, call stack, etc.*)
  - ▶ So each MIPS cycle requires many CPU cycles
- ▶ Still, performance is not that poor
  - ▶ We still manage to achieve gigabit speed...
  - ▶ ... at the cost of 100% CPU usage on one core (active loop and context switches)
- ▶ However, what is really important is
  - ▶ packets rate and packet loss
  - ▶ when firmware processes packets
  - ▶ when NAVIS checks are active (ASF traffic)

## ▶ UDP/9



## ▶ UDP/623



Without Navis

With Navis



## Limitations of the approach

- ▶ The detection model is highly adapter-specific:
  - ▶ we tested on BCM5754 and BCM5755M adapters
  - ▶ it could be adapted to other Broadcom adapters (provided they use the same kind of firmwares)
  - ▶ our concept can be ported to other devices as long as similar debug capabilities are present
- ▶ Can an attacker prevent us to control the NIC?
- ▶ We cannot prevent arbitrary writes in code area (since standard behavior seems to allow it)
- ▶ High processing cost

## Conclusion

We proposed NAVIS, a firmware integrity attestation framework:

- ▶ firmware integrity attestation is a (very) hard problem
- ▶ our proof of concept is highly firmware and adapter specific

NAVIS can detect and prevent most low level attacks on NIC firmware

- ▶ But it requires the OS to be trusted.
- ▶ And protecting the OS stays the highest priority.

If the embedded devices implement more functions, could they have more protections too?

## Questions & Answers

Thank you for your attention

Do you have any question?

No network cards were harmed in the making of this paper

# Bibliography I

- [1] Claude Castelluccia, Aurélien Francillon, Daniele Perito, and Claudio Soriente. *On the difficulty of software-based attestation of embedded devices*. In *Proceedings of 16th ACM Conference on Computer and Communications Security*, November 2009.
- [2] K. Chen. *Reversing and exploiting an apple firmware update*. BlackHat, 2009.
- [3] Guillaume Delugré. *Closer to metal : Reverse engineering the broadcom netextreme's firmware*. Hack.lu, 2010.
- [4] Loïc Duflot and Yves-Alexis Perez. *Can you still trust your network card?* CanSecWest, 2010.
- [5] Aurélien Francillon, Claude Castelluccia, Daniele Perito, and Claudio Soriente. *Comments on "refutation of on the difficulty of software based attestation of embedded devices"*. -, 2010.
- [6] Alexandre Gazet. *Sticky fingers & kbc custom shop*. -, 2011.
- [7] Yanlin Li, Jonathan M. McCune, and Adrian Perrig. *SBAP: Software-Based Attestation for Peripherals*. In *Proceedings of the 3rd International Conference on Trust and Trustworthy Computing (Trust 2010)*, June 2010.
- [8] Yanlin Li, Jonathan M. McCune, and Adrian Perrig. *VIPER: Verifying the integrity of peripheral's firmware*. In *Proceedings of CCS'11*, 2011.
- [9] Alfredo Ortega and Anibal Sacco. *Deactivate the rootkit*. BlackHat, 2009.

## Bibliography II

- [10] Adrian Perrig and Leendert Van Doorn. *Refutation of “on the difficulty of software based attestation of embedded devices”*. -, 2010.
- [11] Fernand L. Sang, Eric Lacombe, Vincent Nicomette, and Yves Deswarte. *Exploiting an I/O MMU vulnerability*. In *MALWARE '10: 5th International Conference on Malicious and Unwanted Software*, pages 7–14, 2010.
- [12] Arrigo Triulzi. *Taking NIC backdoors to the next level*. CanSecWest, 2010.
- [13] Ralf-Philipp Weinmann. *All Your Baseband Are Belong To Us*. CCC, 2010.