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

Loïc Duflot, **Yves-Alexis Perez**,
Benjamin Morin

Agence Nationale de la Sécurité des Systèmes d'Information

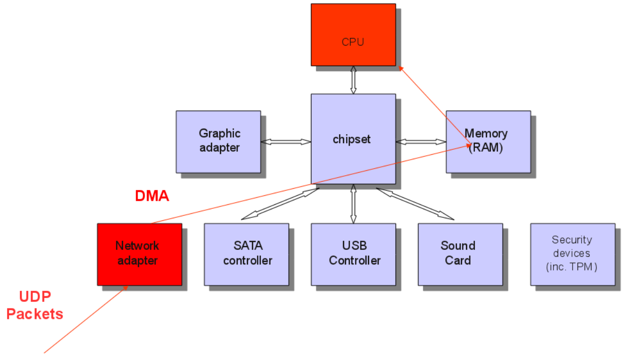
Liberté • Égalité • Fraternité
RÉPUBLIQUE FRANÇAISE
Premier ministre
Agence Nationale de la Sécurité des Systèmes d'Information

Last year...

- 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.
- We showed a live demo, using 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.

ANSSI

This is how it worked



See http://www.ssi.gouv.fr/trustnetworkcard

ANSSI

And...

We presented possible countermeasures, but none of them seemed really convincing:

- patching is the most obvious countermeasure, however
  - you can only patch vulnerabilities you know of,
  - patching applications on an OS isn't simple at all (and not always done), patching firmwares is even harder,
  - firmwares in ROM can't be patched;
- `I/O MMUs` can help, but are not a 100% efficient (see later).

This year we study what an efficient countermeasure would be.

Finding efficient countermeasures is critical

In the last few years, people have been looking at firmware and embedded software:

- Basebands (Weinmann [17])
- Network cards (Triulzi [15], Delugré [4])
- Keyboard controllers (Chen [3], Gazet [6])
- Chipsets (Ortega and Sacco [8])

Defending a system against such attacks is difficult as firmware are running out of the scope of the operating system.

Problem statement

- We have a machine (smartphone, computer, tablet PC) accessing the network through a network adapter.
  - This network adapter is running a firmware
  - We need to check the integrity of the firmware
- Firmware's integrity must be checked
  - at load time
  - during run time

Verification at load-time

- Firmware load-time integrity can be checked using a `TPM`
  - A `TPM` is a secure cryptographic chip present on most platforms, whose primary goal is to assure the integrity of a platform.
  - Specific software (incl. embedded software) can be measured to detect changes to previous configurations.
  - Peripherals' firmware should be part of the components that are measured during the *trusted boot pathway*.
  - Using *Dynamic Root of Trusts* can even solve race conditions at boot time.
- So we can pretty much consider that the problem is solved.

But how can we check the integrity of the platform during its execution?

Verification at run-time

- Run-time integrity verification basically consists in checking that an untrusted *target* is running untampered
- The verification is performed by a trusted *verifier* during the execution of the target
- Can be achieved with *software-based remote attestation*
- In this case,
  - The target would be the network adapter
  - The verifier would be the operating system

Remote firmware attestation [7]

Remote device attestation is based on a challenge-response protocol

1. The verifier sends a random nonce to the target
   - The nonce is used as a seed to prevent replay attacks

Remote firmware attestation [7]

Remote device attestation is based on a challenge-response protocol

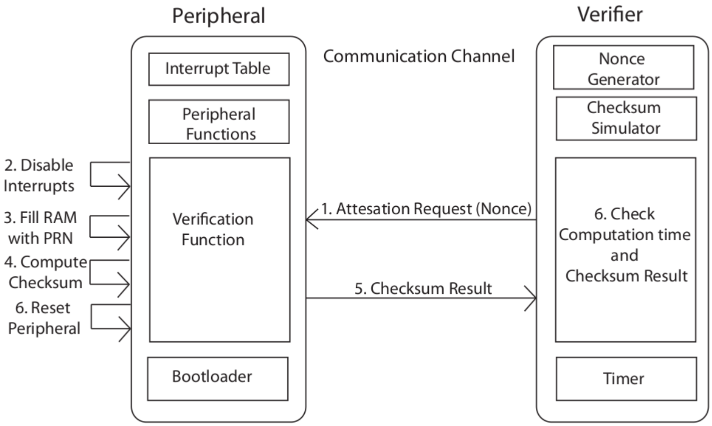
1. The verifier sends a random nonce to the target
2. The target computes a checksum over its entire memory and returns it to the verifier
    - Data and unused code memory is erased with a predictable value
    - Memory is read in a pseudo-random traversal to prevent checksum precomputation
    - All interrupts are disabled during the computation of the checksum
    - The device is reset after the checksum is returned

Remote firmware attestation [7]

Remote device attestation is based on a challenge-response protocol

1. The verifier sends a random nonce to the target
2. The target computes a checksum over its entire memory and returns it to the verifier
3. The verifier checks the correctness of the result
    - The verifier has a copy of the expected target's memory content and compares the checksum returned by the target with its own computation
    - The verifier also checks that the computation time is within fixed bounds

Remote firmware attestation [7]



Remote firmware attestation is difficult [2, 9, 5]

Challenges

- A malware could keep a (compressed) copy of the legitimate firmware code in memory and redirect memory reads to compute the correct checksum

Solutions

- Checksum computation time must be predictable and near-optimal in order to detect checksum computation overheads caused by memory redirects
- The verifier must know the exact target hardware configuration

Remote firmware attestation is difficult [2, 9, 5]

Challenges

- A malware could keep a (compressed) copy of the legitimate firmware code in memory and redirect memory reads to compute the correct checksum
- Data memory is unpredictable and may contain malware code

Solutions

- Checksum computation time must be predictable and near-optimal in order to detect checksum computation overheads caused by memory redirects
- The verifier must know the exact target hardware configuration
- Data memory is reset into a predictable state before attestation with pseudo-random values

ANSSI

Remote firmware attestation is *very* difficult

Is remote firmware attestation adequate for complex devices such as network adapters?

- The assumption that the device cannot communicate with a third-party machine during computation may not hold (especially for a network adapter...)
- The checksum function imposes severe constraints
  - It requires to reset the memory of the device and block all interrupts
  - It can be time consuming for the device

Firmware attestation might not be suited for devices with harsh time constraints

Host-based IPS

- Other approaches have been proposed to monitor the integrity of a system at a low level.
    - By using a dedicated hardware coprocessor to monitor the integrity of the memory (Copilot [10]),
    - By using an embedded microcontroller in the chipset (DeepWatch [1]),
    - By embedding the verifier in System Mode Management (HyperGuard [12], HyperCheck [16]).
- However, these mechanisms are designed to protect the main operating system
    - ... and it is unclear whether they can be used to monitor the integrity of peripherals.
    - ... moreover, some require a trusted network card for remote attestation, e.g. [16]

ANSSI

Assumptions

- First, we need to assume that the host operating system is trusted
- `I/O MMU` can be used to protect the operating system

```
kernel: DMAR:[DMA Write] Request device [08:00.0] fault addr 0
kernel: DMAR:[fault reason 05] PTE Write access is not set
kernel: DRHD: handling fault status reg 2
kernel: DMAR:[DMA Write] Request device [08:00.0] fault addr 1282000
kernel: DRHD: handling fault status reg 2
kernel: DMAR:[fault reason 05] PTE Write access is not set
```

- ... however, a compromised network adapter may still attack other peripherals [13]

So what do we do?

The solution is (unfortunately) likely to be specific to the adapter. The kind of live verifications that we will be able to carry out will depend on the architecture of the controller we are considering.

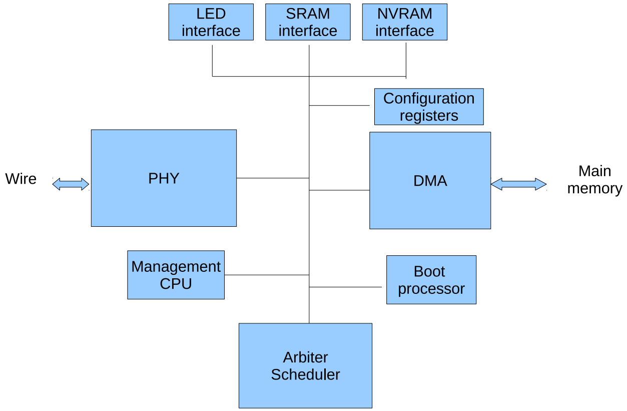
- What kind of interface is available to the host?
- and what does this interface allow us to do?

In the remainder, we consider the case of the Broadcom NetXtreme network adapter.

Case study on Broadcom NetXtreme architecture

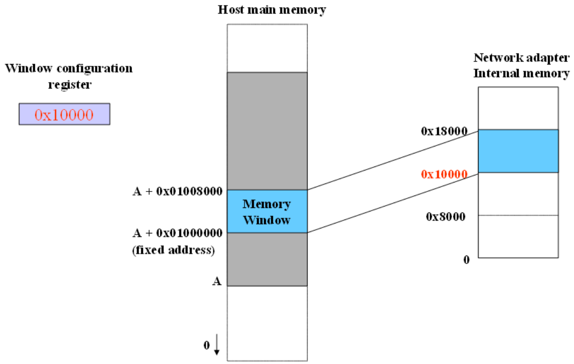
- Broadcom provides a complete set of specifications of their network adapter for open source driver development.
- From last year's study, we know that many interesting components of the network card are directly accessible to the host :
  - registers (control, state, status, breakpoint registers)
  - internal memory
- Everything is accessible in the `MMIO` region dedicated to interactions between the network card and the driver.

Internal architecture of NetXtreme network card



Full access to internal memory

- Access to internal memory is achieved by using a memory window to browse the content of the memory
- This mechanism provides direct access to the firmware running on the adapter



ANSSI

Access to control registers

Among the registers that are accessible from the host:

- state registers indicate whether the embedded CPU is stalled or not (and if so, why),
- control registers allow us to run the embedded CPU of the network adapter step by step,
- breakpoint registers allow us to selectively enable debug conditions associated with addresses.

Running the network adapter in step by step mode

- Last year we used such an interface to craft an external debugger
- However, we could also use the interface to analyse the behaviour of the firmware in real time :
  - to monitor the activity of the firmware from the host, and
  - to detect strange or unusual behaviours.

Search for MIPS code in memory (idea rejected)

- We could periodically search memory areas (stack, scratchpad, heap) for anything that looks like MIPS executable code
  - Conceptually similar to emulation-based shellcode detection [11, 14]
- Such data locations are used to store ethernet packets
- There is no reason why data stored there should meet the statistical profile of MIPS II instructions.
- Depending on the analysis strategy, we might get false positives.

Step-by-step instruction comparison

- Basic code integrity check procedure:
  - at initialisation phase, we record a golden model of the firmware
  - for every single step, we check that the instruction that is to be run is the same as the golden model's one
  - otherwise we stop the network adapter.
- This technique only works if the code is not self modifying (which is the case for the firmware we are considering)

Step-by-step instruction address checking

We can also very easily check at each step if the instruction pointer value is consistent.

- if the instruction pointer says that the network card is running code in the heap, in the stack or in the memory scratchpad, something is wrong
- this requires to be able to identify where code and data are located.

Identifying code and data area

- Broadcom docs and driver code tells us firmwares files have three areas:
  - *text* (code),
  - *data*,
  - *rodata*
- we don't have the mappings into the card memory
- maybe we can find them?

Finding memory mappings

- We can watch the `RX RISC` and monitor:
  - code execution: *instructions executed by the CPU*
  - `CPU` writes: *addresses written by the CPU (`SH`, `SB`, `SW`)*
  - `CPU` reads: *addresses read by the CPU (`LH`/`LHU`, `LB`/`LBU`, `LW`)*
  - other writes: *network packets written to the card memory by `DMA` from host and by `PHY` from the wire*
- By monitoring these events we can map the CPU activity.
- The mapping will be highly `NIC` and firmware specific, but the same analysis could be done for other combinations.

ANSSI

Implementation

- We can make our integrity verifier:
  - check *bounds* for the `CPU`: two areas means four bounds, easy
  - check read and write access (but there will still be plenty of room to put stuff)
- What we can't do:
  - prevent arbitrary writes in code area (since standard behavior seems to allow it)

Maintaining a shadow stack

- Something else that we can do is maintain a shadow *call stack*:
  - the idea is to keep a reconstructed copy of the call stack of the firmware on the host.
  - each time we identify a `CALL`-like instruction, we push the corresponding return address on the shadow stack.
  - each time we identify a `RET`-like instruction, we check that the address where the firmware is trying to return meets the one that we saved on the stack
  - If it is not the case, then something is definitely wrong

ANSSI

Maintaining a shadow stack

- Something else that we can do is maintain a shadow *call stack*:
  - the idea is to keep a reconstructed copy of the call stack of the firmware on the host.
  - each time we identify a `CALL`-like instruction, we push the corresponding return address on the shadow stack.
  - each time we identify a `RET`-like instruction, we check that the address where the firmware is trying to return meets the one that we saved on the stack
  - If it is not the case, then something is definitely wrong
- However, maintaining a shadow stack on the host is not easy
  - We need to identify function `CALL`s and `RET`s
  - The firmware runs on a `MIPS` architecture and there is no `CALL`/`RET` instruction in `MIPS` assembly language

Basics of the MIPS CPU

RISC architecture with 32 internal general purpose registers

```
******** CPU Registers *************************************
$0   = 00000000 $1   = 00010000 $2   = 00000000 $3  = 40000000
$4   = 0001b4b8 $5   = 0001b8e6 $6   = 00000000 $7  = 0001bfc4
$8   = 00000040 $9   = 00000050 $10  = 0001b8bc $11 = 0001bfc0
$12  = 80000000 $13 = 00000001 $14  = 00000000 $15 = ffffffbf
$16  = a4020000 $17 = aaaaaaaa $18  = 00000000 $19 = 0001af48
$20  = 0000ad60 $21 = 018004f1 $22  = 000000fc $23 = 00010000
$24  = ffffffff $25 = 80000000 $26  = 00000b50 $27 = 00011104
$28  = c0000000 $29 = 0001bfd8 $30  = 0001c000 $31 = 000111f8
```

- `r29` is usually used as a stack pointer
- `r31` is usually used to hold a return value
- `r0` must be zero

Branching instructions on a MIPS CPU

- Only Jump and Branch instructions. Some examples:
  - `BEQ` Branch on equal
  - `JAL` Jump and link: jump to immediate address and store return address in `r31`
  - `JR r`: Jump to address stored in register r

Characteristics of the firmware (from experimentations)

- Fortunately for us, the firmware that we are monitoring is pretty simple:
  - function calls are done through the `JAL` instructions
  - There are no function pointers. `JAL` are always performed on absolute values.
  - returns from functions are done through JR 31.
- Locating function call and `RET` is not that hard... Except...

Interrupt management

- Interrupts can be triggered in the network adapters.
  - Interrupt are asynchronous
  - some of them can be predicted (by looking at the MIPS CPU status registers)
  - but it is hard to predict the exact CPU cycle when the interrupt will be triggered.
- Interrupts cause unexpected changes in the control flow of the network adapter.
  - they can cancel instructions (because of the MIPS delay slot)
  - so we need to take them into account

Identifying and dealing with interrupts

- In the firmware we are looking at:
  - there is only one interrupt handler starting at a fixed address (interrupt vector)
  - return from the handler is done through `JR r27`
- So identifying interrupts is not so hard:
  - detect unexpected jumps to the interrupt vector and check that the program will go back using `JR r27`
- But sometimes, interrupts cause errors on the shadow stack:
  - The MIPS delay slot is ignored on interrupt, so we need to take that into account.

Limits

- 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.

Demo

Demonstration of an attack being prevented.

Performance

- Surprisingly enough, performance is not that poor.
  - we run the MIPS in step-by-step mode
  - at each MIPS cycle we do various tests (*bounds, call stack*...)
  - so each MIPS cycle leads to a lot of host CPU cycles
- We still manage to achieve gigabit speed
  - 100% CPU usage on one core
  - the overhead is due to `ASF` processing, not network traffic
- Performances might not be that good with other firmwares which need to touch every network package

Conclusions

- firmware integrity attestation is a hard problem
- proof of concept exists but it's highly firmware and adapter specific

You still need to trust your network card

and protect your OS as much as you can.

ANSSI

Question & answers

?

ANSSI

Bibliography I

[1] Yuriy Bulygin and David Samyde. *Chipset based approach to detect virtualization malware.* BlackHat, 2008.

[2] 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.

[3] K. Chen. *Reversing and exploiting an apple firmware update.* BlackHat, 2009.

[4] Guillaume Delugré. *Closer to metal : Reverse ingineering the broadcom netextreme's firmware.* Hack.lu, 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] Alfredo Ortega and Anibal Sacco. *Deactivate the rootkit.* BlackHat, 2009.

[9] Adrian Perrig and Leendert Van Doorn. *Refutation of "on the difficulty of software based attestation of embedded devices".* –, 2010.

Bibliography II

[10] Nick L. Petroni, Jr. Timothy, Fraser Jesus, Molina William, and A. Arbaugh. *Copilot - a coprocessor-based kernel runtime integrity monitor.* In *Proceedings of the 13th USENIX Security Symposium*, pages 179–194, 2004.

[11] Michalis Polychronakis, Kostas G. Anagnostakis, and Evangelos P. Markatos. *Emulation-based detection of non-self-contained polymorphic shellcode.* In *Recent Advances in Intrusion Detection*, volume 4637 of *Lecture Notes in Computer Science*, pages 87–106. Springer, 2007.

[12] Joanna Rutkowska and Rafa Wojtczuk. *Preventing and detecting xen hypervisor subversions.* BlackHat, 2008.

[13] Fernand L. Sang, Eric Lacombe, Vincent Nicomette, and Yves Deswarte. *Exploiting an I/OMMU vulnerability.* In *MALWARE '10: 5th International Conference on Malicious and Unwanted Software*, pages 7–14, 2010.

[14] Makoto Shimamura and Kenji Kono. *Yataglass: Network-level code emulation for analyzing memory-scanning attacks.* In *Proceedings of the 6th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, DIMVA '09, pages 68–87. Springer, 2009.

[15] Arrigo Triulzi. *Taking NIC backdoors to the next level.* CanSecWest, 2010.

Bibliography III

[16] Jiang Wang, Angelos Stavrou, and Anup Ghosh. *Hypercheck: a hardware-assisted integrity monitor.* In *Proceedings of the 13th international conference on Recent advances in intrusion detection*, RAID'10, pages 158–177. Springer-Verlag, 2010.

[17] Ralf-Philipp Weinmann. *All Your Baseband Are Belong To Us.* CCC, 2010.