What if you can't trust your network card?

Loïc Duflot, Yves-Alexis Perez, and Benjamin Morin

ANSSI
French Network and Information Security Agency
51 boulevard de la Tour Maubourg, 75007 Paris
firstname.lastname@ssi.gouv.fr

Abstract. In the last few years, many different attacks against computing platform targeting hardware or low level firmware have been published. Such attacks are generally quite hard to detect and to defend against as they target components that are out of the scope of the operating system and may not have been taken into account in the security policy enforced on the platform. In this paper, we study the case of remote attacks against network adapters. In our case study, we assume that the target adapter is running a flawed firmware that an attacker may subvert remotely by sending packets on the network to the adapter. We study possible detection techniques and their efficiency. We show that, depending on the architecture of the adapter and the interface provided by the NIC to the host operating system, building an efficient detection framework is possible. We explain the choices we made when designing such a framework that we called NAVIS and give details on our proof of concept implementation.

Keywords: firmware, NIC, network adapter, runtime verification

1 Introduction

In [8], we demonstrated how it is possible for an attacker to take full control of a computer by exploiting a vulnerability in the network adapter[1]. This proof of concept shows how it is possible for an attacker to take full control of the adapter and to add a backdoor in the OS kernel using DMA accesses. The vulnerability was unconditionally exploitable when the ASF function was enabled on the network card to any attacker that would be able to send UDP packets to the victim.

While preventing the network card from tampering with the operating system is possible using existing mechanisms, having a compromised network card remains a real problem, not only because the network card is a critical component from the security perspective, but also because a compromised device can be used to compromise surrounding peripherals on the computer.

Possible countermeasures were considered in [8], but none of them seemed really convincing. The best way to prevent a network card from being compromised would probably consist in formally verifying that the code running in the

[1] See http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2010-0104.

firmware is correct. Considering that network adapters' firmware code is increasingly complex and generally proprietary, the prevention problem is brought down to a *detection* problem. In this paper[2], we propose a pragmatic approach to detect network card corruptions, where the monitor is located inside the operating system. As much as we know, the kind of attacks we are trying to detect has not been the subject of many papers in the intrusion detection community. Still, these attacks represent a real threat considering the privilege level an attacker might gain in successfully exploiting the underlying vulnerabilities. Moreover, we believe that studying a *detection* approach (as opposed to a *prevention* one) is relevant, as the vulnerabilities reside in a component which is not completely under user control.

Our contribution is twofold. First, we raise the community's awareness of the threats associated with widespread devices by illustrating the effectiveness of an attack against a network device. Second, we present a solution to this problem in the form of an anomaly detection system called NAVIS (Network Adapter Verification and Integrity checking Solution). This solution is based on several detection paradigms and aims at instantly blocking attacks against firmware embedded on the target network device. Our goal is to block attacks corresponding to a modification of the control flow of the embedded device, while maintaining good performance and virtually avoid false positives. As an illustration of the efficiency of the NAVIS framework, we focus on a particular network adapter and developed a proof of concept implementation of our detection system.

The paper is organised as follows. In section 2, we present existing mechanisms to assess firmware integrity. Section 3 summarizes our previous attack on a network card and its implications on the security of a system. Then, we present the assumptions for our work, on which we build our firmware corruption detection system. Our prototype implementation of the monitor is described in section 5. Section 6 illustrates the effectiveness of our approach and presents experimental results. Before concluding and evoking future work, we discuss the limitations of our approach.

2 Problem Statement and Related Work

2.1 Attacks against firmware

In the last few years, several researchers have examined the security of firmware and embedded software in various devices, such as basebands [27], network cards [25,7], keyboard controllers [6] or chipsets [24].

These attacks might enable an intruder to take full control of the component and use it as a stepping stone to run other attacks against the OS (through DMA attacks) or other peripherals. Even without bouncing on the component, the attack itself might be interesting to eavesdrop data (keylogger on the keyboard controller) or perform man in the middle stealthily (on the network card).

[2] Our results have been presented in the CanSecWest 2011 conference [9].

2.2 Countermeasures

Defending a system against such attacks is difficult as firmware are running out of the scope of the operating system and potentially have a wide access on other systems resources (like the PCI bus) and there is not much control over what they actually do.

Patching is the most obvious countermeasure. However, one can only patch known vulnerabilities, and patching firmware is even harder than patching applications on an operating system. Moreover, adapters often start running resident firmware in ROM before dynamically loading a newer firmware. This resident firmware cannot be patched, so there might be a window of opportunity before a new, fixed firmware can be safely loaded.

As we will see later, IOMMUs can help protect the system, but it is not 100% efficient as it might not protect other peripherals, as shown by Sang et al. [21]. Besides, IOMMU does not protect the affected subsystem, whose corruption can be critical, especially in the case of a network card (as previously mentioned, it may lead to e.g., passive eavesdropping).

Many vulnerability mitigation techniques have been proposed in the literature for defending against arbitrary code execution attacks ; these include address space layout randomization (ASLR) [22], canaries [13], W⊕X principle (a.k.a NX bit), data tainting. However, some of them can independently be circumvented by attackers. W⊕X techniques for can instance be circumvented using Return oriented Programming (ROP) and canaries will fail to be efficient against ROP without returns [5], [23]. But most importantly, these defense techniques are impractical in the case of firmware because these systems generally lack the required basic features since they run on hardware-constrained devices with embedded CPUs like MIPS.

Our approach basically consists in verifying the integrity of the firmware of a network card at runtime in order to detect malicious control flow alterations. Generally speaking, run-time integrity verification consists in checking that an untrusted target is running untampered. In the remainder of this section, we focus on two kinds of protection approaches against arbitrary code execution attacks, namely CFI (Control Flow Integrity) and Remote firmware attestation.

2.3 Control flow integrity

Classical Control Flow Integrity (CFI) [1] security policy dictates that software must follow a path of a Control-Flow Graph determined ahead of time. The CFG can be determined by analysis (source code analysis, binary analysis or execution profiling).

In its objectives, our intrusion detection approach is similar to CFI, applied to a firmware, as proposed by Francillon et al. [11]. We control access to memory regions, which can be seen as a form of Software Memory Access Control (SMAC), and we use a shadow call stack to achieve detection. Our monitor uses an execution profile of the network card, which can be seen as a very coarse and primitive form of access control policy. The profile is built ahead of time and is

derived from an inspection of multiple executions of the firmware. It is used by the monitor at runtime to detect abnormal executions.

However, our approach differs from CFI in its design. First, we do not rewrite the code of the firmware. Second, we do not have a fine grained model to dynamically ensure that the control flow remains within an expected control flow (i.e., Control Flow Graph).

Similar to CFI, software guards [4,10] use program rewriting techniques in order to insert code elements in a host program. These elements may perform arbitrary tasks at runtime to protect the host program against illegitimate modifications (e.g., self-checksumming). They have primarily been used to implement software cracking protections, but software guards could be used to implement temper-resistance features inside firmwares.

2.4 Remote firmware attestation

Runtime integrity verification can be achieved with software-based remote attestation [15]. The verification is performed by a trusted verifier during the execution of the target. In our case, the target would be the network adapter and the verifier would be the operating system.

Remote device attestation is based on a classical challenge-response protocol, where the verifier first sends a random nonce n to the target. The target then computes a checksum over its entire memory using n as seed[3] and returns the checksum to the verifier. The verifier then checks the correctness of the result.

The target 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

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

As discussed by several authors [3,17,12], remote firmware attestation is difficult. First, a malware could keep a (compressed) copy of the legitimate firmware code in memory and redirect memory reads to compute the correct checksum. For this reason, checksum computation time must be predictable and near-optimal in order to detect checksum computation overheads caused by memory redirects. Also, the verifier must know the exact hardware configuration of the target. Second, data memory must be reset into a predictable state before attestation with pseudo-random values because otherwise, data memory is unpredictable and may contain malware code.

In [15], remote firmware attestation has been implemented on Apple Aluminum Keyboard firmware, which is a rather simple device. Still, attestation takes up to two seconds, during which the peripheral is unresponsive. This leads us to the following question : is remote firmware attestation adequate for complex devices such as network adapters? Indeed, the checksum function imposes

[3] The nonce is used as a seed to prevent replay attacks.

severe constraints : it requires to reset the memory of the device and block all interrupts, which can be time consuming for the device. Moreover, the assumption that the device cannot communicate with a third-party machine during computation may not hold (especially for a network adapter...). As a summary, we doubt whether firmware attestation is currently suited for devices with harsh time constraints.

2.5 Other IDS-oriented protections

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 [18]), by using an embedded microcontroller in the chipset (DeepWatch [2]), or by embedding the verifier in System Mode Management (HyperGuard [20], HyperCheck [26]). However, these mechanisms are primarily 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., [26]), which is "problematic" in our case.

3 Exploiting network adapters firmware vulnerabilities

In [8], we demonstrated how it is possible for an attacker to subvert the execution of a network adapter by exploiting a software fault in its firmware code and then gain control over the operating system.

Network adapters have become complex objects. Indeed, they are not only used to process network frames and transfer them between the wire and the operating system anymore. They are also used as *out-of-band* low-cost management devices. Their position in the hardware stack (i.e., between the operating system and the network) has led manufacturers to develop new remote administration functions like ASF (Alert Standard Format), IPMI (Intelligent Platform Management Interface) or AMT (Active Management Technology), which allow network adapters to communicate with a command and control node. Moreover, those administration functions are active even with a broken, powered-off or even absent operating system, which means that they have a very privileged position on the motherboard and have access to other components (like System Management Bus (SMBus), PCI bus or ACPI).

The administration functions are not handled completely in hardware but rather using a management CPU included on adapters, which runs an embedded firmware and performs various tasks (network frames handling, authentication, interactions with the platform, etc.). The CPU inspects network frames before sending them to the OS and, when the adapter is the final destination, process the whole packets to perform the administrative tasks.

The vulnerability that was exploited in [8] lied in the authentication part of the ASF firmware of some *Broadcom NetXtreme* adapters. When ASF was

enabled, the adapter was vulnerable to remote code execution before any authentication was performed, meaning that an attacker could run any code on the embedded CPU. On the card itself it was possible to examine each and every packet (from and to the OS), to send packets to a remote machine for later inspection or to reconfigure the card itself (a proof of concept changing MAC addresses and LED configuration was done). Attacking the platform was also possible, for example by forcing an ACPI restart through the SMBus.

Using a DMA attack, it was possible to compromise the running kernel and insert a backdoor in it. In our attack, the backdoor basically consisted in opening a reverse shell when certain type of ICMP packet were processed by the host.

Other attacks are conceivable, which do not require to fully compromise the host operating system (e.g., SSLstrip-like attacks, ARP and DNS caches poisoning, packet drops, etc.), which is why it is not sufficient to protect the host from a compromised network card. We need to be able to detect network card corruption.

4 Detecting network adapter firmware corruption

This section describes the principle of the NAVIS network adapter integrity checker. NAVIS is a kind of anomaly detection system which checks memory accesses performed by the NIC processor against a model of expected behaviour based on its memory layout profile. Any memory access that is outside the NIC memory profile is interpreted as an attempt to divert the firmware control flow. Of course, profiling the memory layout of the network card is a prerequisite to try to detect attacks. In the remainder of this section, we first present our basic assumptions for our detection system before describing the memory profiling approach. The anomaly detection heuristics are described in the last part of this section. The details of implementation, the practical obstacles, and how they are circumvented are described in the next section.

4.1 Assumptions

Our objective is to detect an adapter firmware corruption at runtime from the host operating system. Therefore, we need to assume that the operating system is trusted (i.e., that it cannot be compromised by the controller), as it plays the role of the verifier. We also assume that the firmware is not compromised in the initial state of the system, i.e., we have to check the controller firmware's integrity at system startup. We believe that these two assumptions are realistic by using standard mechanisms that equip current computers.

Firmware load-time integrity can be enforced using a TPM (Trusted Platform Module) [14]. A TPM is a secure cryptographic chip present on most x86 platforms, whose primary goal is to allow the operating system to verify the integrity of the platform. Specific software (including embedded software) can

be measured by the operating system using the TPM to detect unexpected configuration changes. Peripherals' firmware should be part of the components that are measured during the *trusted boot pathway*. After a (trusted) kernel is booted, the network driver will force a firmware reload, using a trusted file on the system (integrity checked via TPM calls) and the reset the embedded CPU.

As pointed out by Rutkowska [19], using so-called *Dynamic Root of Trusts* can even solve race conditions at boot time. We consider such techniques to provide an efficient solution to the problem of integrity verification of embedded software at load-time. As a result, we do not study such a problem in this paper.

Operating system's runtime integrity can be enforced by means of an IOMMU mechanism. Once the system is booted in a trusted state (thanks to a TPM and the *dynamic root of trusts*), an IOMMU protects it from DMA attacks initiated from the devices by only allowing them access to a specific (and private) area of the main memory. Any attempt to access memory outside that area fails and triggers an alert on the system.

Other types of attacks against the operating system (either direct or through userland applications) are outside the scope of this paper.

4.2 Model of the network adapter

Figure 1 sketches the typical architecture of a network card. The PHY is responsible for sending and receiving signals on the wire and performing physical and logical conversions. The SRAM is the volatile memory area where packets are temporarily stored before being sent to the operating system by means of the DMA controller of the card. The *management CPU* is an on-chip processor which operates independently of all architectural blocks and is intended to run a custom firmware that can be used for custom frame processing. Many different firmware types exist, e.g., management firmware (for ASF, IPMI or AMT) or accelerators like TSO (TCP segmentation Offloading).

Model of the memory layout As NAVIS monitors NIC memory accesses, we now focus on the memory of the network card.

In theory, the architecture of a network adapter should be quite simple. Like most embedded systems, NICs are based on a Von Neumann memory architecture, where executable code and data are located in a single address space. The software which makes up a firmware is usually executed as a monolithic application. As a result, firmware generally lacks memory protections that are commonly found on custom systems (such as a memory management unit, randomization or NX features) because they do not require memory protection between different applications or isolation between kernelland and userland.

In fact, one may argue that the integration of additional features in network adapters (see section 3) should make these protections a requirement. However, apart from the fact that it would probably degrade the NIC performances, having

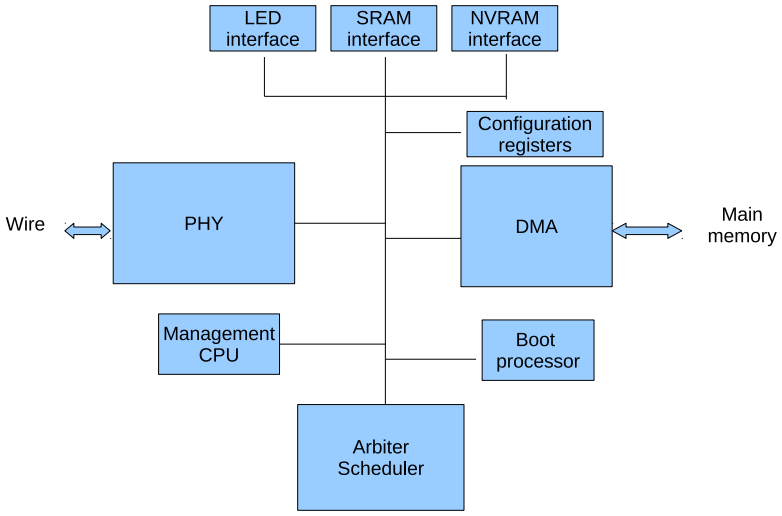


Fig. 1: Architecture of network adapter

a more sophisticated adapter in a computer would give rise to other questions regarding the security model of the overall system.

To sum up, our approach is based on a flat memory model that combines both code and data, on top of which we enforce access restrictions and control flow integrity verifications.

Next, the memory layout model must distinguish precisely those memory areas that are used to execute code, to read and write data, and specify which areas are in read-only mode. In the case of the network card, data read and write operations can be performed by three components of the card : the management CPU, the DMA controller and the PHY.

The DMA controller and the PHY are used to transfer packets between the host and the wire, which are stored in a specific place in the card memory. Some area is reserved too for storing the structures used to synchronize DMA transfers between the host and the card (mainly pointers to the packets themselves).

The management CPU uses some memory for the code it executes, for the read-only data shipped with the firmware and for the various structures usually needed (like room for a stack and heap). As it usually needs to process some packets (e.g management packets for ASF or TCP packets for TSO), it can read and write on the memory area used for storing packets. The management CPU also has access to the sending area because it might need to send packets.

Building the reference memory layout One of the obstacles that came up in building the reference memory layout of the network card used in our experiments is that the purposes of the various memory areas are not public.

Therefore, we have built the reference memory layout of the network card empirically, by monitoring the NIC activity during typical network sessions : large HTTP download, SSH sessions and legitimate ASF traffic (session open, a few "query" commands and session close). The data obtained is a good representation of the network controller activity. Details on the memory reference model acquisition are given in section 5.3.

Figure 3 (p.14) shows the memory map of the card used in our experiments. Of course, this memory map is highly card-specific, but our acquisition procedure can applied to other card models, provided that the cards can be tightly controlled by the host.

4.3 Detection heuristics

Based on the memory model presented previously, NAVIS uses three complementary detection heuristics to detect network controller firmware corruptions. The first two aim at enforcing access restrictions on memory areas. The third one is used to detect potential control flow integrity violations and uses a *shadow return stack*.

During the initialization phase, NAVIS records a *golden model* of the firmware, which serves as a reference for the subsequent verifications. As a reminder, we assume that the golden model is authentic (see 4.1). NAVIS then acts as a debugger to keep track of the NIC CPU operations and update its internal model of the NIC status. The following verifications a performed at each state transition.

Step-by-step instruction address checking: Based on the memory layout model, NAVIS checks the consistency of the instruction pointer at each execution step. If the instruction pointer points to a memory area that corresponds to the heap, the stack or the scratchpad, then a code injection attack followed by a control flow redirection probably occurred.

Step-by-step instruction comparison: In addition to the previous verification, NAVIS also checks that there is a match between the instruction that is to be run by the CPU and the one that should be run according to the golden model. A mismatch is indicative of a code injection in the NIC memory, in which case the NIC is stopped.

Of course, this heuristic is valid only if the code is not self modifying. This assumption does not seem excessive : despite their increasing complexity, one does not expect network cards to require the execution of self-modifying code for their legitimate processing.

This assumption might need to be revisited at some point. Management firmware already include software like web and application servers, it might be

possible that in the future java-based applications become available, where code would be written in memory before beeing executed, and thus there would be no *golden model* for that part. Anything using *just in time* execution would fail the assumption.

Shadow stack : In order to detect malicious control flow alterations, we maintain a simplified copy of the call stack of the firmware on the verifier side, called the *shadow stack*. The shadow call stack is used to verify that a function call returns to the callsite most recently used for invoking the function. Of course, the shadow call stack must be maintained in a protected memory, so that the attacker cannot modify it. In our case, the shadow call stack is maintained on the host side, in userland, which is assumed to be trusted.

The shadow stack is updated every time a `CALL`-like or a `RET`-like instruction is executed by the firmware as follows:

- on a `CALL` instruction, the return address is pushed on the shadow stack;
- on a `RET` instruction, the target return address is matched against the one that was previously saved on the shadow stack; a difference between the two addresses is the sign of an anomaly.

The concept of a shadow call stack is not original by itself, but its implementation turns out to be complex on a concrete network adapter whose firmware architecture is not known (see section 5.5 for details). The main challenges actually reside in the identification of CALL and RET instructions and in the presence of interrupts triggered by components of the NIC. These interrupts are susceptible to disrupt the control flow of the firmware which is monitored.

This approach is similar in its principle to the Instruction-Based Memory Access Control mechanism proposed by Francillon [11], except that we do not have to implement the monitor *inside* the firmware. This is possible because the former has physical access to the latter, and because we assume that the network card cannot subvert the operating system. In a way, our settings are less constraining than his, but they are also the only viable solution considering that we do not modify the underlying NIC hardware.

Step-by-step instruction address checking may seem superfluous, considering that the attack types it detects are included in those that are detected by the shadow stack. However, step-by-step instruction address checking may prove useful in practice when the specificities of a given network adapter make the implementation of shadow stack protection inaccurate (in particular, dealing with on-board interrupts is a difficult task, see 5.5). We chose to use all three techniques considering that our implementation of the shadow stack technique might not be perfect (because of specificities of the network adapter). The shadow stack is also the slowest method so it makes sense to enable it only when it is really needed.

Other heuristics : Another way to detect code injection attacks could consist in scanning the memory in search of values whose statistical distribution matches

that of executable code in memory areas that are supposed to contain data only (heap, stack and scratchpad). Such data locations are used to store ethernet packets and there is no reason why data stored there should meet the statistical profile of binary instructions.

We mention this type of detection criterion here, but it has not been implemented. Indeed, due to its statistical nature, this approach is more error prone than the previous ones, and its benefits are uncertain. Also, scanning the whole packet area every time a packet arrives would be time consuming and would degrade the performances of NAVIS.

5 Implementation of NAVIS

In the remainder, we consider the case of the *Broadcom NetXtreme* network adapter. Those adapters can be found on various type of machines but are generally integrated on mainboard of desktop and laptops sold by HP and Dell. The variants used in this study are mobile versions of the 575x series.

5.1 Quick description of the Broadcom NetXtreme network adapters

Broadcom provides a complete set of specifications of their network adapters for open source driver development which we used as a basis for our work.

The network card follows the model shown in Fig. 1. The management firmware is run by a MIPS CPU which has access to the various components and especially the whole memory area.

The memory layout is described in Broadcom documentation though a lot of space is either undocumented or explicitly marked as *unmapped*. Depending on the documentation version, read access to *unmapped* areas returns *all zeros* or *unexpected data* while write access *are dropped internally* or *have no effect*. In practice, useful data can sometime be found on unmapped areas.

The host communicates with the card through different ways. The driver can configure it using MMIO address space (including DMA configuration) and then sends and receives data through DMA reads and writes in a reserved address space setup initially. The data structures used to communicate with the cards are called *rings* since they are circular buffers. Several such rings are used for sending and receiving packets, both in the card memory and in the main host memory. The rings contain pointers (in a structure called *buffer descriptor*) to the packet, and the ring is controlled by a structure named *ring control block*. These structures are located in various places in the card memory.

The firmware uses area allocated from the card memory space. It needs room for storing the code as well as the various data structures (heap, stack etc.).

5.2 Low level interface to the device

We first need to be able to reach the network card (and especially the embedded CPU and the firmware) from the operating system to allow NAVIS to perform various verifications to ensure firmware integrity.

Such an interface was implemented to analyse the vulnerability presented in section 3, as well as to craft an external debugger for the network adapter's embedded MIPS CPU that executes the firmware. The same interface is reused to analyse the standard behaviour of the firmware and monitor the CPU activity in real time from the host and detect strange or unusual behaviours.

From our previous study, we know that many interesting components of the network card are directly accessible to the host, like registers and internal memory. Everything is accessible in the MMIO region dedicated to interactions between the network card and the driver.

Among the registers that are directly accessible from the host:

- the program counter indicates what is the next instruction which will be fetched and executed by the embedded CPU,
- 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.

Access to internal memory is achieved by using a memory window (Fig. 2). This mechanism provides direct access to the firmware running on the adapter: reading an address in the card memory means writing the base address to the relevant register and reading at the correct offset in the MMIO address space.

5.3 Memory profiler

Identifying code and data area: The documentation and driver code show that firmware files have three areas (text (code), data, and read-only data), but the exact mappings into the card memory are not specified, so we first need to identify them.

Thanks to the low-level interface to the NIC, the following operations of the embedded CPU are monitored:

- code execution: instructions executed by the CPU,
- CPU write operations: addresses written by the CPU (`SB`, `SH`, `SW`[4]),
- CPU read operations: addresses read by the CPU (`LB`/`LBU`, `LH`/`LHU`, `LW`[5]),
- other write operations : network packets written to the card memory by DMA from host and by PHY from the wire.

[4] *store byte, halfword, word*

[5] *load byte, byte upper, half word, half word upper, word .*

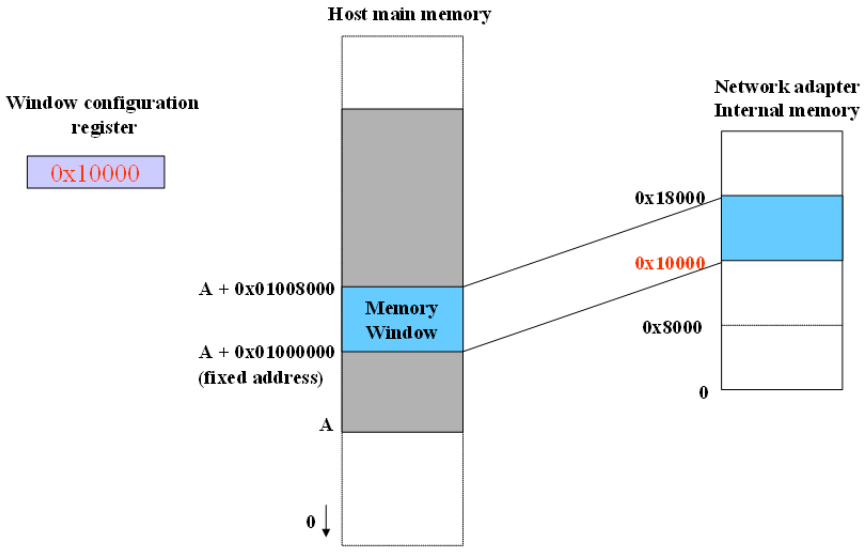


Fig. 2: Memory window

By monitoring these events we can map the CPU activity. The mapping will be highly adapter and firmware specific, but the same analysis could be performed for other combinations.

We made a record of that activity during a somehow standard network session: large HTTP download, SSH sessions and legitimate ASF traffic (i.e., session open, a few "query" commands for the system state and session close). The data obtained is a good representation of the network controller activity since the host sends and receives various traffic and the network controller receives, processes and sends ASF packets, performs authentication and session management, and communicates with the platform for collecting information about the system state.

5.4 Memory map analysis

According to the memory map (Fig. 3), we know where the CPU reads and writes data: first in the structures used for replying to ASF traffic (the ring control blocks, the transmit ring and the TXMBUF area, where packets are stored before sending), then in the *scratchpad* (a generic writeable area, where received packets are stored for handling), and finally the CPU stack and heap. We also know where the CPU executes code (in a space taken from the RXMBUF and scratchpad area where the firmware is stored), with a main area and a secondary area just before the stack.

We also note that there are external writes to the CPU code area and writes in areas noted as *unmapped* in the documentation. Other external writes include

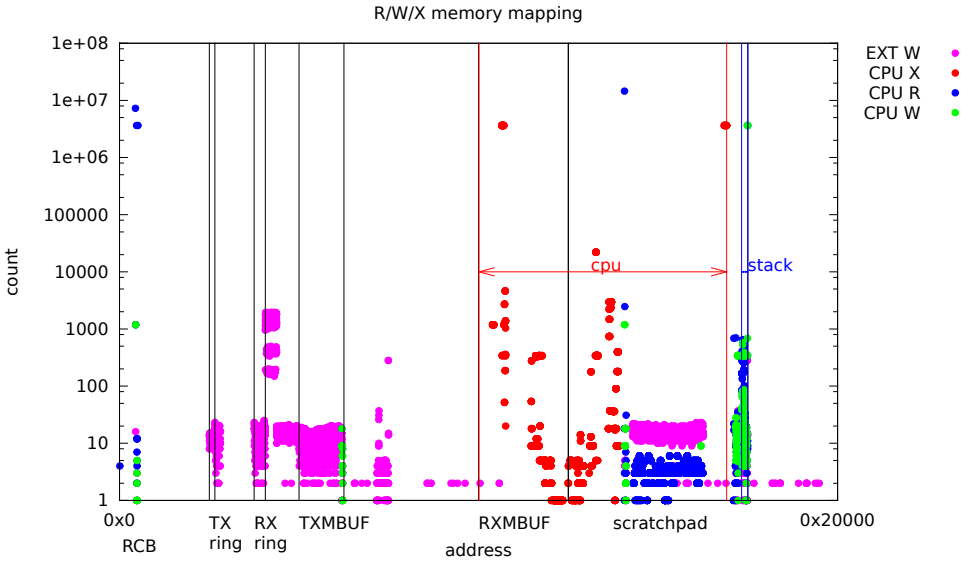


Fig. 3: Memory map

network packets from and to the host, located in the `RX` and `TX` rings and network packets to the adapter (ASF traffic), stored in the *scratchpad*.

Finally, it's important to note that there is no way to enforce *rodata* and that there is no segmentation/pagination mechanisms.

5.5 Implementation of the detection heuristics

Step-by-step instruction address checking: The instruction checks are easy but highly specific. The analysis should therefore been done for each NIC model and each firmware version.

The first heuristic checks the program counter against code bounds. The recorded model defines some areas where code is expected to be located, and code execution outside these areas by the CPU is indicative of an attack. This verification is more complex to implement than the next one because it was not possible to find a unique code area on this specific card model and firmware version combination. Thus, multiple checks must be done because there are one main area and several sub-areas. An incomplete model (i.e, one which does not cover the whole ranges of operations for the analysed firmware) may lead to false positives and false negatives.

Step-by-step instruction comparison: The second heuristic compares the content of the memory area pointed to by the program counter (which is the

address of the next instruction to be executed) and compares it to the recorded *golden model*. A mismatch between the two indicates that the code has been overwritten and that an attack is ongoing (since we assume that the code is not self-modifying). In this case, the monitor halts the embedded CPU.

Implementation of the shadow stack Maintaining a shadow stack on the host is complex because we need to identify function `CALL`s and `RET`s. Unfortunately, the firmware runs on a MIPS architecture, and there are no such instructions in MIPS assembly language.

On MIPS architecture with 32 internal general purpose registers, `r29` is usually used as a stack pointer and `r31` to hold a return value while `r0` is always zero. Other registers are used for general operations.

MIPS CPU only have jump and branch instructions. For instance, `BEQ` is *branch on equal*, `JAL` is *jump and link* (jump to immediate address and store return address in `r31`), and `JR r` is *jump register* (jump to address stored in register `r`).

Fortunately, 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`.

In theory, locating function `CALL`s and `RET`s is not difficult. However, we have to manage interrupts, which can be triggered in the network adapters asynchronously. Some of them can be predicted (by looking at the MIPS CPU status registers), but it is difficult to predict the exact CPU cycle when the interrupt will be triggered. Interrupts cause unexpected changes in the control flow of the network adapter and can cancel instructions (because of the MIPS delay slot). Therefore, we need to take interrupts into account to implement our shadow stack.

In the firmware we are looking at, there is only one interrupt handler starting at a fixed address (interrupt vector), and return from the handler is done through `JR r27`. As a result, identifying interrupts is possible : we need to detect unexpected jumps to the interrupt vector and check that the program will go back using `JR r27`. However, interrupts sometimes cause errors on the shadow stack: the MIPS delay slot is ignored on interrupt, so we need to take that into account. Indeed, if an interrupt is taken when a `CALL` instruction (or a `RET`) instruction is in the delay slot, the CPU will indeed perform as if running this instruction (causing a modification of the shadow stack) when in fact this instruction is ignored (as if replaced by a NOP in the CPU pipeline). As a consequence, each time our framework detects an interrupt, we check whether the last instruction that was supposed to be run was a `CALL` or a `RET` instruction. If it is the case, that means that our shadow stack is incorrect and we have to correct it.

6 Experimental results

6.1 Effectiveness of the detection

Needless to say that the kinds of attacks we are trying to detect are extremely specific. Therefore, it would not make sense to check the effectiveness of our tool against, e.g. the DARPA evaluation dataset.

Also, our intrusion detection system basically consists in finding evidences of code injection and control flow redirects in the memory of the network card using simple heuristics, so our detector cannot actually be tuned. Therefore, using ROC curves (receiver operating characteristic curves) to test it would not be relevant either [16] (plotting the true-positive rate of detection against the corresponding false-positive rate of error implies a degree of freedom in the settings of the detector).

One way to evaluate the effectiveness of our intrusion detection system experimentally may consist in testing it against a set of various attacks (e.g., stack overflow, return-oriented programming) and/or vulnerabilities of the same type. However, implementing variants of arbitrary code execution attacks is time-consuming, especially on exotic and undocumented architectures. Moreover, as our detection approach only relies on the measurable *effects* of the attacks on the monitored system (not on attack signatures), merely applying code obfuscation techniques do not seem to be relevant.

As a summary, we can essentially speculate on the detection effectiveness from a theoretical point of view.

6.2 Experimental Settings

As a consequence, we chose a very simple experimental setting.

For our experiment, we used a Dell D530 laptop using a 5755M Broadcom NetXtreme adapter running a firmware vulnerable to the different kinds of attacks we presented in [8]. The laptop is running Debian Squeeze with our NAVIS detection framework.

In one setting of the experiment, the target PC is directly connected to the internet through the adapter we are monitoring and we manually simulate standard user actions (FTP downloads, web browsing etc.). At the same time, we allowed automatic processes to access resources on the web several days in a row. In a second setting we directly connect the adapter to a PC emulating an attacker sending attack packets that will try to exploit vulnerabilities in the adapter. Three different types of payload are used for the experiments.

In our first experiment, none of the packets associated with regular traffic did trigger any alert from NAVIS. On the contrary, all three different kind of attacks using ASF traffic were successfully detected by NAVIS.

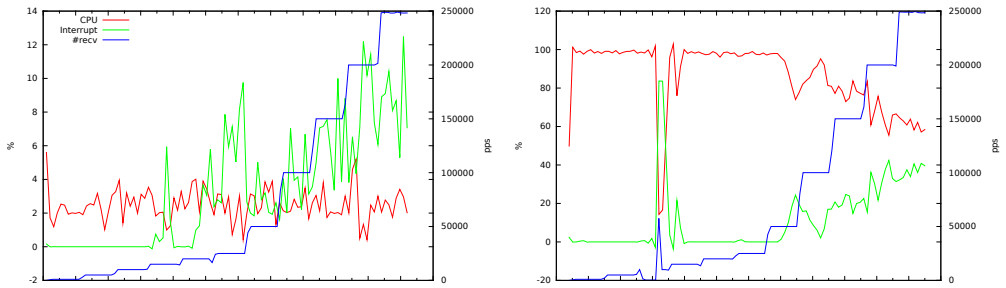
6.3 Performance

We were expecting that our detection framework would drastically decrease the performances of the machine we are monitoring. Indeed, we run the MIPS CPU

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. As a result, NAVIS uses 100% CPU for one core even when the adapter is not processing traffic. Indeed, when the MIPS processor is idle (because there is no ASF traffic at all) it loops on an waiting procedure which means the host CPU still analyses the various steps.

The network adapter speed itself is not impaired by the detection technique. Even after activating NAVIS, we still achieve gigabit speed. This comes from the fact that the firmware we are monitoring only processes special kind of UDP packets (ASF packets) so the fact that this firmware is running in step by step mode does not have any kind of impact on regular traffic.

The testbed is composed of the Dell D530 laptop (IP 192.0.2.1), a gigabit switch and a second machine with a gigabit ethernet card (IP 192.0.2.2). The test is run using `pktgen` (a packet generator included in the Linux kernel), while `dstat` (a statistics collecting tool) is run on the receiving machine (the D530 one) to monitor CPU usage along with network statistics (mainly packet rate). The test is done in two parts, first on a standard installation (Fig. 4a) then with (Fig. 4b) NAVIS running. Generated traffic is sent and received on UDP port 9 and packet size is 256 and the source machine sends traffic at rates from 1000 to 250 000 packets per second.



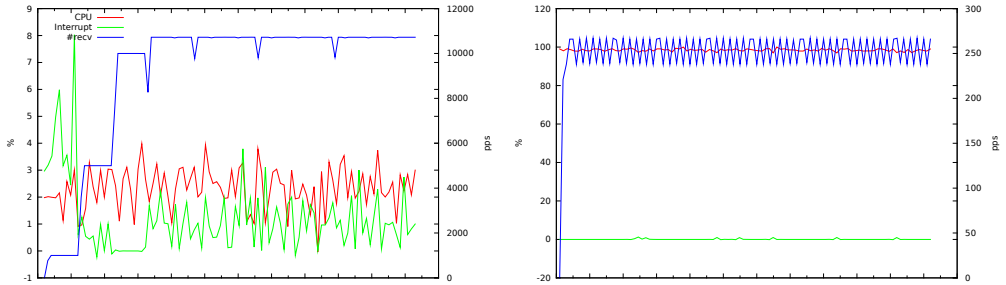
(a) NAVIS stopped (b) NAVIS running

Fig. 4: CPU usage and packet rate (UDP port 9)

It's pretty clear that NAVIS does not really prevent the network to reach full speed on this test, as both packet rate curves have the same shape when send rate augments and they both reach 250 000 packets per second. At low packet rates, the 100% CPU usage is mostly the active loop of the debugger. When packet rate rises, software interrupts from system calls are starting to become significant. The packet generator isn't able to generate more traffic but it seems likely that NAVIS could handle more packets before slowing down the traffic.

Performances might not be that good with firmware needing to process every network packets. A good test for that case is to send UDP packets on port 623 (`ASF/RMCP` port) to the D530. In that case the PHY will detect the packet needs to be handled by the firmware, which needs to check if the datagram is ASF traffic or not before relaying it to the host.

So we run the same test, this time sending datagrams to UDP port 623.



(a) NAVIS stopped (b) NAVIS running

Fig. 5: CPU usage and packet rate (UDP port 623)

Even when NAVIS is not running (Fig. 5a), we have issues sending datagrams to the network card. Processing done by the firmware to check if the packet is ASF or not is slowing down the whole packet processing, meaning the PHY queues are full and ethernet frames are dropped when packet rate is above 11000.

When running the same tests with NAVIS, we can achieve speeds around 24Mb/s, but packet rate drops dramatically and barely exceeds 250 packets per second (Fig. 5b). The speed issues aren't related to all the context switches from the system calls (since interrupts are mostly at 0%) but are due to the time spent in processing the various memory accesses to the card.

It might be worth implementing the *verification* part of NAVIS inside the kernel and optimize all the PCI accesses in order to improve the packet processing rate of the whole installation.

7 Limitations of the approach

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

This approach 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. One could imagine that an attacker would be able to craft an attack only by being able to modify data areas. These kind of attacks would not be detected by NAVIS.

Moreover, the fact that the firmware we are considering is quite simple makes it easier for us to verify its integrity. For instance, the following characteristics simplify the analysis:

- the firmware is not using any kind of indirection for `CALL` operations (there are no function pointers). Function adresses are hardcoded and can be easily identified by disassembling `CALL` instructions;
- no paging mechanism is involved. Addresses in the firmware are physical addresses and therefore our framework does not need to perform any kind of address translation;
- the firmware is running on the embedded CPU as a single thread.

8 Conclusion and Future work

In this paper we studied the difficult problem of firmware integrity attestation or verification. We looked at the problem from a theoretical point of view and showed that depending on the interface of the device we are considering and the nature of the firmware, monitoring was possible. In our setting, the host operating system acts as an external verifier running a framework called NAVIS that constantly analyses the behaviour of the embedded firmware and stops the device whenever an unexpected behaviour is detected. We developed a proof of concept for a popular model of network adapter and showed that our proof-of-concept was indeed efficient against attacks (even 0-day ones). The proof-of-concept is highly specific to the adapter but shows that firmware integrity verification can be achieved in practice.

Future work on this topic involves studying alternate detection mechanisms such as on the fly virtualisation and control by an hypervisor of embedded firmware.

References

1. Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow integrity principles, implementations, and applications. *ACM Transactions on Information and System Security*, 13, November 2009.
2. Yuriy Bulygin and David Samyde. Chipset based approach to detect virtualization malware. BlackHat, 2008.
3. 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.
4. Hoi Chang and Mikhail J. Atallah. Protecting software code by guards. In *ACM Workshop on Security and Privacy in Digital Rights Management*, 2001. ACM Workshop on Security and Privacy in Digital Rights Management, Philadelphia, Pennsylvania, November 2001.
5. Stephen Checkoway, Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, Hovav Shacham, and Marcel Winandy. Return-oriented programming without returns. In *Proceedings of the 17th ACM conference on Computer and communications security*, CCS '10, pages 559–572. ACM, 2010.

6. K. Chen. Reversing and exploiting an apple firmware update. BlackHat, 2009.
7. Guillaume Delugré. Closer to metal : Reverse ingineering the broadcom netextreme's firmware. Hack.lu, 2010.
8. Loïc Duflot and Yves-Alexis Perez. Can you still trust your network card? CanSecWest, 2010.
9. Loïc Duflot, Yves-Alexis Perez, and Benjamin Morin. Run-time firmware integrity verification : what if you can't trust your network card? CanSecWest, 2011.
10. Ùlfar Erlingsson, Martìn Abadi, Michael Vrable, Mihai Budiu, and George C. Necula. Xfi: Software guards for system address spaces. In *Symposium on Operating System Design and Implementation (OSDI)*, volume 4637, pages 75–88, 2006.
11. Aurélien Francillon. *Attacking an Protecting Constrained Embedded Systems from Control Flow Attacks*. PhD thesis, Institut Polytechnique de Grenoble, 2009.
12. Aurélien Francillon, Claude Castelluccia, Daniele Perito, and Claudio Soriente. Comments on "refutation of on the difficulty of software based attestation of embedded devices". -, 2010.
13. Mike Frantzen and Mike Shuey. Stackghost: Hardware facilitated stack protection. In *Proceedings of the 10th conference on USENIX Security Symposium - Volume 10*, SSYM'01, pages 5–5. USENIX Association, 2001.
14. Trusted Computing Group. The trusted platform module.
15. 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.
16. R. A. Maxion and R. R. Roberts. Proper use of roc curves in intrusion/anomaly detection. Technical report, School of Computing Science, University of Newcastle upon Tyne, 2004.
17. Adrian Perrig and Leendert Van Doorn. Refutation of "on the difficulty of software based attestation of embedded devices". -, 2010.
18. 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.
19. Joanna Rutkowska. Remotely attacking network cards (or why do we need vt-d and txt), 2010.
20. Joanna Rutkowska and Rafal Wojtczuk. Preventing and detecting xen hypervisor subversions. BlackHat, 2008.
21. 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.
22. Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. On the effectiveness of address-space randomization. In *Proceedings of the 11th ACM conference on Computer and communications security*, CCS '04, pages 298–307. ACM, 2004.
23. Saravanan Sinnadurai, Qin Zhao, and Weng fai Wong. Transparent runtime shadow stack: Protection against malicious return address modifications.
24. Alexander Tereshkin and Rafal Wojtczuk. Introducing ring -3 rootkits. BlackHat, 2009.
25. Arrigo Triulzi. Taking NIC backdoors to the next level. CanSecWest, 2010.
26. 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.
27. Ralf-Philipp Weinmann. All Your Baseband Are Belong To Us. CCC, 2010.