Ghost in the Wireless, iwlwifi edition

Nicolas Iooss and Gabriel Campana
nicolas.iooss@ledger.fr
gabriel.campana@ledger.fr

Ledger Donjon

Abstract. Wi-Fi replaced Ethernet and became the main network protocol on laptops for the last few years. Software implementations of the Wi-Fi protocol naturally became the targets of attackers, and vulnerabilities found in Wi-Fi drivers were exploited to gain control of the OS, remotely and without any user interaction. However, not much research has been published on Wi-Fi firmware, outside of Broadcom models. This article presents the internals of an Intel Wi-Fi chip. This study, mostly conducted through reverse engineering, led to the discovery of vulnerabilities such as arbitrary code execution on the chip and secure boot bypass, which were reported to the manufacturer.

1 Introduction

1.1 How we met the Intel Wi-Fi chip

One day in January 2021, Gabriel tried to browse a web application hosted by his laptop using his smartphone. This operation seems simple, but that day, it made his laptop disconnect from the Wi-Fi network, and this was reproducible. As this was quite annoying, he opened his kernel log (listing 1).

```
 1  iwlwifi 0000:01:00.0: Start IWL Error Log Dump:
 2  iwlwifi 0000:01:00.0: Status: 0x00000100, count: 6
 3  iwlwifi 0000:01:00.0: Loaded firmware version: 34.0.1
 4  iwlwifi 0000:01:00.0: 0x00000038 | BAD_COMMAND
 5  ...
 6  iwlwifi 0000:01:00.0: Start IWL Error Log Dump:
 7  iwlwifi 0000:01:00.0: Status: 0x00000100, count: 7
 8  iwlwifi 0000:01:00.0: 0x00000070 | ADVANCED_SYSASSERT
 9  ...
10  iwlwifi 0000:01:00.0: 0x004F01A7 | last host cmd
11  ieee80211 phy0: Hardware restart was requested
```

Listing 1. Messages appearing in Linux kernel log while requesting a web page

The failed assertion (line 8) indicated an issue in the firmware of the Wi-Fi chip. This issue was easy to reproduce and only occurred when both the smartphone and the laptop were connected to the same Wi-Fi access point. Why is this happening? Can it be exploited, for example to run arbitrary code on the Wi-Fi chip?

This event started an adventure in the internals of Intel Wi-Fi chips. As the interactions between a kernel module and a hardware component can be very complex, the first step was to better understand the Linux kernel module driving the chip. This work quickly led to the code actually loaded on the chip. Nicolas then joined the adventure and developed some tooling, as using IDA disassembler felt too rudimentary. Analyzing the code led to the discovery of a simple vulnerability enabling arbitrary code execution on the Wi-Fi chip.

As the chip was quite old, we also experimented on a more recent laptop, with a more recent Wi-Fi chip. The differences between the chips are presented in figure 1. We did not find the same vulnerability on this chip, and both chips included a mechanism preventing modified firmware from being loaded (by verifying a digital signature). So at first we did not have any way to run arbitrary code on this newer chip.

	First chip	Second chip
Hardware device	Intel Dual Band Wireless AC 8260	Intel Wireless-AC 9560 160MHz
Launch date	Q2 2015	Q4 2017
Firmware file	`iwlwifi-8000C-34.ucode`	`iwlwifi-9000-pu-b0-jf-b0-46.ucode`
Firmware version	`34.0.1`	`46.6f9f215c.0`

Intel website resources: `https://www.intel.com/content/www/us/en/products/sku/86068/intel-dual-band-wirelessac-8260/specifications.html` and `https://www.intel.com/content/www/us/en/products/sku/99446/intel-wirelessac-9560/specifications.html`

Fig. 1. Differences between the two studied Wi-Fi chips

Both Wi-Fi chips expose a rich interface to the Linux kernel. Using it, we managed to dump the code which actually verifies the firmware signature. Analyzing this code quickly led to the discovery of a simple signature verification bypass on the first studied chip. Unfortunately this bypass did not work on the newer chip, even though the root cause of the issue did not appear to be fixed. After some weeks, we found a way to bypass the signature verification on the newer Wi-Fi chip too.

Being able to run arbitrary code on the chip enabled us to gain a more precise understanding of its working. For example, the Wi-Fi firmware is too large to fit in the memory of the chip and a mechanism is implemented to store code and data in the main system memory. This is what Intel calls the *Paging Memory* in the source code of the Linux kernel module. The content of this memory has to be authenticated in some way, to prevent an attacker on the main operating system from modifying it. In practice, the firmware seems to use a hardware-assisted universal message authentication code to ensure the integrity of each page in this Paging Memory. The details of this mechanism do not seem to be publicly documented anywhere, even though they are key to ensure the security of the chip.

1.2 State of the art and contributions

The first public remote exploits against Wi-Fi were presented in 2007 [9]. The exploited vulnerabilities were found in Linux kernel modules thanks to fuzzing. These modules being open-source and their code quality quite low, multiple vulnerabilities were found in the Wi-Fi kernel modules of major network cards manufacturers. Public analysis of Wi-Fi firmware wasn't a thing at that time, probably because the attack surface of kernel modules was sufficient for attackers to gain access to a remote computer.

In 2010 [8], the reverse engineering of an Ethernet network card firmware led to the discovery of vulnerabilities in the ASF protocol implementation. The researchers successfully gained control of this network card, remotely.

In 2012 [4], the firmware of an Ethernet Broadcom chip was reverse engineered and modified to include a debugger and eventually a backdoor. Broadcom's Ethernet and Wi-Fi firmware aren't encrypted or signed and can thus be patched, allowing dynamic analysis. Public datasheets also help analysis [3] [7]. Vulnerabilities in Broadcom's Wi-Fi chipsets were found and exploited in 2017 [2].

In this article, we'll present the internals of Intel Wi-Fi chips, gained through the reverse engineering of the associated firmware. While the firmware source code isn't available, the Linux kernel module interacting with these PCI chips is open source and is of great help. Links to the Linux kernel sources are specific to the version 5.11 in order to have permalinks.

The main contributions of this article are:

— The publication of an Intel Wi-Fi firmware parsing tool,
— Reverse engineering of Intel Wi-Fi firmware,
— Internals of these firmware,
— Exploitation of vulnerabilities in the secure-boot mechanisms,
— Publication of on-chip instrumentation, tracing and debugging tools.

2 Finding the firmware code

2.1 Discovering iwlwifi

When studying a hardware component such as the Intel Wi-Fi chip, one of the first things to do is to identify which one it is: its model name, revision number, etc. On a laptop which was used to perform experiments, the kernel log indicated the presence of an Intel Wireless-AC 9560 chip handled by `iwlwifi`, the Linux kernel module for Intel Wireless Wi-Fi (listing 2).

```
iwlwifi 0000:00:14.3: Detected Intel(R) Wireless-AC 9560 160MHz,
    REV=0x318
```

Listing 2. Extract of kernel log showing information about the Wi-Fi chip

In practice, four kernel modules are used to implement the Wi-Fi feature with this chip, in Linux 5.11:

— `iwlwifi`[1] handles the hardware interface (through the PCIe bus) with the chip.

— `iwlmvm`[2] implements some higher-level interface to the firmware of chips using MVM (which seems to be an acronym for multi-virtual MAC).

— `mac80211`[3] implements a IEEE 802.11 (Wi-Fi) networking stack in Linux.

— `cfg80211`[4] provides a configuration interface to user-space programs.

The modules `iwlwifi` and `iwlmvm` support many versions of Intel Wi-Fi chips. To identify which version is used, these modules use the PCI device ID. The studied chip uses a PCI device ID `9df0` (listing 3), which is mapped to a structure named `iwl9560_trans_cfg` in `iwlwifi`.[5]

```
$ lspci -nn -s 00:14.3
00:14.3 Network controller [0280]: Intel Corporation Cannon Point-LP
     CNVi [Wireless-AC] [8086:9df0] (rev 30)
```

Listing 3. Requesting the PCI device ID using lspci

[1] `https://elixir.bootlin.com/linux/v5.11/source/drivers/net/wireless/intel/iwlwifi`

[2] `https://elixir.bootlin.com/linux/v5.11/source/drivers/net/wireless/intel/iwlwifi/mvm`

[3] `https://elixir.bootlin.com/linux/v5.11/source/net/mac80211`

[4] `https://elixir.bootlin.com/linux/v5.11/source/net/wireless`

[5] `https://elixir.bootlin.com/linux/v5.11/source/drivers/net/wireless/intel/iwlwifi/pcie/drv.c#L463`

To communicate with the chip, `iwlwifi` configures the first Base Address Register (BAR) of the PCIe interface, using functions `pcim_iomap_regions_request_all` and `pcim_iomap_table`.[6] This is a standard way of communicating with a PCIe chip using Memory-Mapped Input/Output (MMIO). After configuring this interface, the kernel module uses it to retrieve some hardware revision information. Then, at some point, the function `iwl_request_firmware`[7] tries to load a file named `iwlwifi-9000-pu-b0-jf-b0-{API}.ucode`[8] where `{API}` is a number identifying the interface version of the firmware. At the time of the study, the Linux firmware repository[9] contained 6 such files, with numbers between 33 and 46. To study the correct firmware, it was necessary to find out which one was actually loaded. And this information was actually written in the kernel log (listing 4)!

```
iwlwifi 0000:00:14.3: loaded firmware version 46.6f9f215c.0
    9000-pu-b0-jf-b0-46.ucode op_mode iwlmvm
```

Listing 4. Extract of kernel log showing the chosen firmware file

2.2 Dissecting the firmware file

In the hardware world, some devices receive their firmware directly, as an opaque blob, without much analysis from the operating system. The studied Intel Wi-Fi chips are not like these devices. Instead, their firmware files are first decoded by `iwlwifi` and only some parts are actually sent to the chips.

In the kernel module, the function which parses the firmware file is named `iwl_parse_tlv_firmware`.[10] It parses a header followed by a series of Type-Length-Value entries (TLV) containing much information.

The firmware we studied in the experiments is available on `https://git.kernel.org/pub/scm/linux/kernel/git/firmware/linux-firmware.git/tree/iwlwifi-9000-pu-b0-jf-b0-46.ucode?`

[6] `https://elixir.bootlin.com/linux/v5.11/source/drivers/net/wireless/intel/iwlwifi/pcie/trans.c#L3455`

[7] `https://elixir.bootlin.com/linux/v5.11/source/drivers/net/wireless/intel/iwlwifi/iwl-drv.c#L160`

[8] `https://elixir.bootlin.com/linux/v5.11/source/drivers/net/wireless/intel/iwlwifi/cfg/9000.c#L29`

[9] `https://git.kernel.org/pub/scm/linux/kernel/git/firmware/linux-firmware.git/tree/?h=20211216`

[10] `https://elixir.bootlin.com/linux/v5.11/source/drivers/net/wireless/intel/iwlwifi/iwl-drv.c#L554`

h=20210511&id=4f549062619750e76f3155fc50b5c0f6529eed8a. This web page gives the ASCII representation of the firmware, which starts with the header containing a version string IWL.release/core43::6f9f215c.

After the header, each entry of the file starts with a type which is an item of enum iwl_ucode_tlv_type.[11] The actual code which is loaded on the chip is contained in entries with type IWL_UCODE_TLV_SEC_RT and IWL_UCODE_TLV_SEC_INIT (and a few other ones not described here). Each such entry defines a memory *section* (hence the _SEC_ in the name) of the loaded firmware and starts with a 32-bit load address (in Little Endian bit order) followed by the content.

For example, in the studied firmware file, the bytes at offset 0x2f4 are 13000000 bc020000 00404000 06000000 a1000000. This defines a TLV entry of type 0x13=IWL_UCODE_TLV_SEC_RT with 0x2bc bytes. This type enables to decode the remaining bytes as the definition of a firmware section at the address 0x00404000 which starts with the bytes 06000000 a1000000.

Plugging everything together leads to finding the sections presented in listing 5.

```
SEC_RT     00404000..004042b8 (0x2b8=696 bytes)
SEC_RT     00800000..00818000 (0x18000=98304 bytes)
SEC_RT     00000000..00038000 (0x38000=229376 bytes)
SEC_RT     00456000..0048d874 (0x37874=227444 bytes)
SEC_INIT   00404000..004042c8 (0x2c8=712 bytes)
SEC_INIT   00800000..008179c0 (0x179c0=96704 bytes)
SEC_INIT   00000000..00024ee8 (0x24ee8=151272 bytes)
SEC_INIT   00456000..00471d04 (0x1bd04=113924 bytes)
SEC_INIT   00410000..00417100 (0x7100=28928 bytes)
SEC_RT     ffffcccc..ffffccd0 (0x4=4 bytes)
SEC_RT     00405000..004052b8 (0x2b8=696 bytes)
...
```

Listing 5. Raw decoding of the sections in the firmware file

This listing contains some strange entries. For example, some SEC_INIT sections (used at initialization time) seem to be inserted between two sets of SET_RT sections (used for runtime) and the entry for ffffcccc seems off. The iwlwifi kernel module contains a macro which defines this last value as a separator between CPU1 and CPU2 (listing 6).[12] Indeed the studied W-Fi chip contains two processors named *UMAC* and *LMAC*! In literature, *MAC* usually means *Medium Access Controller* and is a layer of

[11] https://elixir.bootlin.com/linux/v5.11/source/drivers/net/wireless/intel/iwlwifi/fw/file.h#L47

[12] https://elixir.bootlin.com/linux/v5.11/source/drivers/net/wireless/intel/iwlwifi/fw/file.h#L461

a network stack. According to Wi-Fi-related documents,[13] it seems *UMAC* means *Upper MAC* while *LMAC* means *Lower MAC*. These documents also give an overview of how these abstraction layers seem to be stacked in Intel Wi-Fi chips (see listing 7).

```
#define CPU1_CPU2_SEPARATOR_SECTION 0xFFFFCCCC
#define PAGING_SEPARATOR_SECTION    0xAAAABBBB
```

Listing 6. Definitions of section separators

```
------------------------------------------+-------------------
  UMAC (Upper Medium Access Controller) | Host Interfaces
------------------------------------------+-------------------
           LMAC (Lower Medium Access Controller)
--------------------------------------------------------------
                   PHY (Physical layer)
--------------------------------------------------------------
                       Wi-Fi Antenna
--------------------------------------------------------------
```

Listing 7. Stack of layers in the Wi-Fi chip (the host communicates with both UMAC and LMAC)

`iwlwifi` also defines the notion of *Paging Memory*. The sections in this *Paging Memory* are loaded using an interface different from the other sections and described later in this article (cf. section 4.1).

All this knowledge gives a better understanding on how the sections are grouped in the firmware file (listing 8).

```
Runtime code for CPU 1 (LMAC):
    SEC_RT    00404000..004042b8 (0x2b8=696 bytes)
    SEC_RT    00800000..00818000 (0x18000=98304 bytes)
    SEC_RT    00000000..00038000 (0x38000=229376 bytes)
    SEC_RT    00456000..0048d874 (0x37874=227444 bytes)

Initialization code for CPU 1 (LMAC):
    SEC_INIT  00404000..004042c8 (0x2c8=712 bytes)
    SEC_INIT  00800000..008179c0 (0x179c0=96704 bytes)
    SEC_INIT  00000000..00024ee8 (0x24ee8=151272 bytes)
    SEC_INIT  00456000..00471d04 (0x1bd04=113924 bytes)
    SEC_INIT  00410000..00417100 (0x7100=28928 bytes)

Runtime code for CPU 2 (UMAC):
    SEC_RT  CPU1_CPU2_SEPARATOR_SECTION ("cc cc ff ff 00 00 00 00")
    SEC_RT    00405000..004052b8 (0x2b8=696 bytes)
    SEC_RT    c0080000..c0090000 (0x10000=65536 bytes)
    SEC_RT    c0880000..c0888000 (0x8000=32768 bytes)
    SEC_RT    80448000..80455ad4 (0xdad4=56020 bytes)
```

[13] https://www.design-reuse.com/articles/39101/reusable-mac-design-for-various-wireless-connectivity-protocols.html

```

Paging code for CPU 2 (UMAC):
    SEC_RT   PAGING_SEPARATOR_SECTION ("bb bb aa aa 00 00 00 00")
    SEC_RT     00000000..00000298 (0x298=664 bytes)
    SEC_RT     01000000..0103b000 (0x3b000=241664 bytes)

Initialization code for CPU 2 (UMAC):
    SEC_RT   CPU1_CPU2_SEPARATOR_SECTION ("cc cc ff ff 00 00 00 00")
    SEC_INIT   00405000..004052b8 (0x2b8=696 bytes)
    SEC_INIT   c0080000..c0090000 (0x10000=65536 bytes)
    SEC_INIT   c0880000..c0888000 (0x8000=32768 bytes)
    SEC_INIT   80448000..80455ad4 (0xdad4=56020 bytes)
```

Listing 8. Decoding of the sections in the firmware file, grouped by kind

2.3 Mapping the memory layout

There are some oddities in the list of the firmware sections presented in listing 8. One of them is that some addresses start with `80` or `c0` instead of `00`. Again, the Linux source code greatly helps to understand what is going on: it defines `FW_ADDR_CACHE_CONTROL` to `0xC0000000`[14] and uses this value to mask the high bits out of some addresses.

During the study we first used these addresses as-is. At some point we stumbled upon the ARC700 Memory Management Unit (MMU) and found in its reference manual [1]:

> The build configuration register `DATA_UNCACHED` (`0x6A`) describes the Data Uncached region. Memory operations that access this region will always be uncached. Instruction fetches that access the same region will, however, be cached as this region relates to data only.
>
> This region, which is only present in builds with an MMU, is fixed to the upper 1 GB of the memory map. As the upper 2 GB of the memory is the un-translated memory region, the Data Uncached region is consequently both uncached and un-translated. This makes this region suitable for e.g. peripherals. Note that this region is active even if the MMU is disabled.

Addresses starting with `c0` are located in the upper 1 GB of the chip memory and are therefore uncached and un-translated references to the memory located at the address given by the remaining bits. And addresses starting with `80`, located in the upper 2 GB of the memory, can be cached

[14] `https://elixir.bootlin.com/linux/v5.11/source/drivers/net/wireless/intel/iwlwifi/iwl-drv.c#L552`

but are never translated by the MMU. For example, the section loaded at address `c0080000` is in fact loaded at physical address `00080000` and uses high bits in order to bypass the MMU translation. This is illustrated in figure 2.

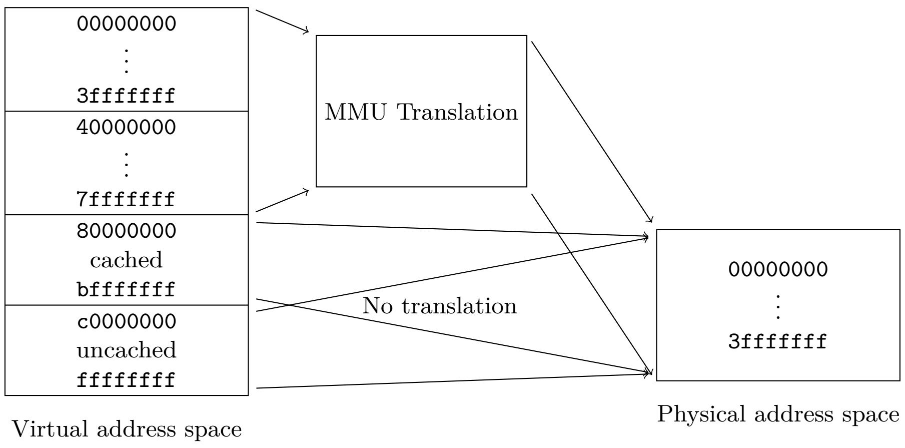

Fig. 2. Virtual and physical address spaces of ARC700 microcontrollers

Moreover `iwlwifi`'s code contains references to the address of two Data Close Coupled Memories (DCCM) and a Static RAM Memory (SMEM).[15] This enables writing a map of the memory layout used by the Wi-Fi chip, presented in figure 3. This figure includes some components which are presented later in this document.

2.4 Verifying the signature

Is it possible to run arbitrary code on the Wi-Fi chip by modifying the firmware file? Now that the layout of the file has been presented, it is possible to try modifying any byte in a section. Doing so triggers a failure reported by `iwlwifi` and prevents the loaded firmware from starting (listing 9).

```
iwlwifi 0000:00:14.3: SecBoot CPU1 Status: 0x3030003,
    CPU2 Status: 0x0
```

Listing 9. Error message seen in the kernel log with a modified firmware

[15] https://elixir.bootlin.com/linux/v5.11/source/drivers/net/wireless/intel/iwlwifi/cfg/9000.c#L21

`00000000..00100000`	Executable memory (maximum 1 MB)
`00000000..00038000`	Code used by CPU 1 (LMAC)
`00060000..000611ca`	Loader code which enforces Secure Boot
`00061e00..00061f00`	Loader Secure Boot RSA public key
`00080000..00090000`	Code used by CPU 2 (UMAC)
`00400000..00490000`	SRAM (Static RAM, 576 KB)
`00401000..00403000`	Loader data, including its stack
`00404000..004042c8`	Code Signature Section for CPU 1 (LMAC)
`00405000..004052b8`	Code Signature Section for CPU 2 (UMAC)
`00410000..00417100`	Code used by CPU 1 Initialization (LMAC)
`00422000..00448000`	Pages used by CPU 2 (UMAC)
`00448000..00455ad4`	Code and data used by CPU 2 (UMAC)
`00456000..0048d874`	Code and data used by CPU 1 (LMAC)
`0048f000..00490000`	Sensitive data used by CPU 2 (UMAC, external read access is denied)
`00800000..00818000`	DCCM (Data Close Coupled Memory, 96 KB) (data used by CPU 1, LMAC)
`00816000..00817000`	Stack for LMAC CPU (4096 bytes)
`00880000..00888000`	DCCM 2 (32 KB) (data used by CPU 2, UMAC)
`00886014..00886334`	Stack for task `IDLE` (800 bytes)
`00886334..00886d34`	Stack for task `MAIN` (2560 bytes)
`00886d34..00887734`	Stack for task `BACKGROUND` (2560 bytes)
`00887734..00887ffc`	Stack for interrupt handlers (2248 bytes)
`00a00000..00b00000`	Hardware Registers (for peripherals)
`00a03088..00a0308c`	Feature flags, including debug mode
`00a04c00..00a04c84`	Access bits for memory regions
`00a24800..00a24b00`	RSA2048 coprocessor
`00a25000..00a25060`	SHA256 coprocessor
`00a38000..00a40000`	NVM (Non-Volatile Memory)

Fig. 3. Map of the physical memory layout used by the studied Wi-Fi chip

In the error message, `SecBoot` likely means *Secure Boot*, a technology used to ensure that only authorized code can run on a platform. How is the firmware authenticated? Usually there is some kind of signature, which is verified against a public key.

Looking at the sections from listing 8 again, they can be grouped in five parts where each starts with a small section, located at address `0x00404000` for the LMAC CPU, at `0x00405000` for the UMAC CPU and at `0x00000000` for the paging memory. This section is not parsed by `iwlwifi` but it is small enough to be able to guess its layout:

— `0x30` bytes: header, including the build date at offset `0x14` (for example the bytes `28 01 21 20` encode the date 2021-01-28)
— `0x50` bytes: zeros (probably some padding)

- `0x100` bytes: RSA-2048 modulus, in Little Endian
- 4 bytes: RSA exponent, always `0x10001`
- `0x100` bytes: RSA-2048 signature, in Little Endian
- 4 bytes: number of other sections of the group, in Little Endian
- For other sections of the group: `0x10` bytes containing four 32-bit Little Endian integers `{7, size + 8, address, size}`

The signature is a RSA PKCS#1 v1.5 signature using SHA256 on the content of every section, including the small first one without the signature field. This confirms that the code loaded on the chip is actually signed.

By the way, even though `iwlwifi` does not parse the small section, it includes some references to something named *CSS*. The meaning of this acronym is not documented but it likely is *Code Signature Section*.

This section contains the public key used to verify the signature. Compared to usual secure boot implementations, this is normal. Indeed, some chips only contain a fingerprint of the public key, for example in their fuses, and verify that the given public key matches this fingerprint. In this case the public key has to be provided. But some chips could forget to check the public key, which would enable attackers to easily bypass the authentication. With the studied Intel Wi-Fi chip, modifying the firmware and re-signing it with a custom key did not work (and triggered the same error as in listing 9).

2.5 Extracting the firmware code

The previous parts detailed the content of a firmware file, the layout of the memory and the way the code was authenticated. This knowledge is more than enough to extract the code which actually runs on the chip. A last question remains before beginning to analyze it: which Instruction Set Architecture (ISA) is the code using? A few years ago a tool named `cpu_rec.py` was published exactly for this kind of need [5]. It guessed that the code used the ARCompact instruction set. This instruction set was supported by IDA Pro disassembler and the generated assembly code seemed to be meaningful.

Moreover, when downloading the Intel Windows drivers,[16] the archives contain a text file `express_logic_threadx.txt` describing license amendments for Express Logic ThreadX (listing 10). This file indicates that wireless connectivity solutions developed by Intel could use ARC 605, ARC7 and ARC6, which belong to the ARCompact family.

[16] `https://www.intel.com/content/www/us/en/download/18231/intel-proset-wireless-software-and-drivers-for-it-admins.html` (accessed on 2022-01-17)

```
Express Logic ThreadX License Amendment / Addendum Summary
[...]
1/9/2008
    Adds ARC 605
[...]
7/11/1012
    Modifications made by this amendment apply only to Intel group
        that develops wireless connectivity solutions
    Adds ARC7
[...]
6/16/2013
    Retroactively replaces ARM7 (Amendment 4) with ARC6
```

Listing 10. Extract of `express_logic_threadx.txt`

To better understand the logic of the firmware, support for these instruction sets was added to Ghidra. This work was already presented at SSTIC 2021 [6].

3 Vulnerability Research

3.1 Executing arbitrary code

Talking to the Wi-Fi chip through debugfs The previous parts focused on static analysis, using files and source code. When analyzing a system, it is useful to also have some way to query its state, debug some code, etc. For Intel's Wi-Fi chip, `iwlwifi` and `iwlmvm` modules expose many files in the debug filesystem. For example, `iwlmvm/fw_ver` contains information about the firmware which was loaded (listing 11).

```
$ DBGFS=/sys/kernel/debug/iwlwifi/0000:00:14.3
$ cat $DBGFS/iwlmvm/fw_ver
FW prefix: iwlwifi-9000-pu-b0-jf-b0-
FW: release/core43::6f9f215c
Device: Intel(R) Wireless-AC 9560 160MHz
Bus: pci
```

Listing 11. Reading the firmware version from Linux debugfs

Among these files, `iwlmvm/mem` enables reading the memory of the Wi-Fi chip (listing 12)!

```
$ dd if=$DBGFS/iwlmvm/mem bs=1 count=128 |xxd
00000000: 2020 800f 0000 4000 2020 800f 0300 e474    ....@.  .....t
00000010: 2020 800f 0300 3837 2020 800f 0000 c819    ....87  ......
00000020: 6920 0000 6920 4000 6920 0000 6920 4000  i ..i @.i ..i @.
00000030: 2020 800f 4700 14b6 6920 0000 6920 4000    ..G...i ..i @.
00000040: 6920 0000 4a20 0000 4a21 0000 4a22 0000  i ..J ..J!..J"..
00000050: 4a23 0000 4a24 0000 4a25 0000 4a26 0000  J#..J$..J%..J&..
00000060: 4a27 0000 4a20 0010 4a21 0010 4a22 0010  J'..J ..J!..J"..
```

```
00000070: 4a23 0010 4a24 0010 4a25 0010 4a26 0010  J#..J$..J%..J&..
```

Listing 12. Reading the beginning of the chip memory

The kernel module also implements write operations with `iwlmvm/mem` but they do not seem to work. During the study we discovered that some Wi-Fi chips could be booted in *debug mode*, where writing to `iwlmvm/mem` would work fine. However, we only had access to Wi-Fi chips in *production mode*, where writing the memory was forbidden.

The debug filesystem also provides another way to read the chip memory with a file named `iwlmvm/sram`. This interface provided by this file only allows reading data from the chip, not writing to it.

Back to the debug filesystem, another file interested us, `iwlmvm/prph_reg`. The Wi-Fi chip contains many peripheral registers (sometimes called *hardware registers*) located at addresses `0x00a*****` and this file enabled reading them. Such registers would usually contain state information, but in the case of the studied Wi-Fi chip, they also included the current Program Counter (`pc`) of the processors! The address of these interesting registers are defined in Linux[17] (listing 13). Even though three `pc` registers are defined, only the first two contain non-zero values on the studied Wi-Fi chip (listing 14): one for the UMAC processor and another for the LMAC processor, which this document described previously (in section 2.2).

```
#define UREG_UMAC_CURRENT_PC    0xa05c18
#define UREG_LMAC1_CURRENT_PC   0xa05c1c
#define UREG_LMAC2_CURRENT_PC   0xa05c20
```

Listing 13. Definitions of program counter registers in Linux

```
$ echo 0xa05c18 > $DBGFS/iwlmvm/prph_reg
$ cat $DBGFS/iwlmvm/prph_reg
Reg 0xa05c18: (0xc0084f40)

$ echo 0xa05c1c > $DBGFS/iwlmvm/prph_reg
$ cat $DBGFS/iwlmvm/prph_reg
Reg 0xa05c1c: (0xb552)

$ echo 0xa05c20 > $DBGFS/iwlmvm/prph_reg
$ cat $DBGFS/iwlmvm/prph_reg
Reg 0xa05c20: (0x0)
```

Listing 14. Reading the values of program counter registers

[17] `https://elixir.bootlin.com/linux/v5.11/source/drivers/net/wireless/intel/iwlwifi/iwl-prph.h#L373`

Talking to the Wi-Fi chip through PCIe The previous section described very useful files in the Linux debug filesystem. How are they actually implemented? More precisely, how is the operating system (Linux) able to read the memory and the peripheral registers of the Wi-Fi chip? Answering these questions is important to understand the security boundaries and how running arbitrary code on the chip is prevented.

Reading `iwlmvm/prph_reg` makes the Linux kernel execute the function `iwl_trans_pcie_read_prph`.[18] A simplified implementation of this function is presented in listing 15.

```
 1 // drivers/net/wireless/intel/iwlwifi/iwl-csr.h
 2 /*
 3  * HBUS (Host-side Bus)
 4  *
 5  * HBUS registers are mapped directly into PCI bus space, but are
 6  * used to indirectly access device's internal memory or registers
 7  * that may be powered-down.
 8  */
 9 #define HBUS_BASE   (0x400)
10 
11 /*
12  * Registers for accessing device's internal peripheral registers
13  * (e.g. SCD, BSM, etc.).  First write to address register,
14  * then read from or write to data register to complete the job.
15  * Bit usage for address registers (read or write):
16  *  0-15:  register address (offset) within device
17  * 24-25:  (# bytes - 1) to read or write (e.g. 3 for dword)
18  */
19 #define HBUS_TARG_PRPH_WADDR    (HBUS_BASE+0x044)
20 #define HBUS_TARG_PRPH_RADDR    (HBUS_BASE+0x048)
21 #define HBUS_TARG_PRPH_WDAT     (HBUS_BASE+0x04c)
22 #define HBUS_TARG_PRPH_RDAT     (HBUS_BASE+0x050)
23 
24 // drivers/net/wireless/intel/iwlwifi/pcie/trans.c
25 u32 iwl_trans_pcie_read_prph(struct iwl_trans *trans, u32 reg) {
26  // Here, 0x03000000 means "read 3+1 = 4 bytes"
27  reg = 0x03000000 | (reg & 0x000FFFFF);
28 
29  // hw_base address mapping the MMIO space of the PCIe endpoint
30  writel(reg, trans->trans_specific->hw_base + HBUS_TARG_PRPH_RADDR);
31  return readl(trans->trans_specific->hw_base + HBUS_TARG_PRPH_RDAT);
32 }
```

Listing 15. Implementation of iwl_trans_pcie_read_prph

In short, `iwl_trans_pcie_read_prph` writes a normalized register index to some offset of the MMIO space (line 30 of listing 15) and reads back a 32-bit value from another offset (line 31). These offsets are documented as being part of a *Host-side Bus* interface (HBUS) and the underlying

[18] `https://elixir.bootlin.com/linux/v5.11/source/drivers/net/wireless/intel/iwlwifi/pcie/trans.c#L1833`

implementation seems to be directly in hardware (it does not involve the firmware). This impression is strengthened by the fact that this interface can be used to read the program counters of the chip processors. Doing so shows values which change so much that this indicates that neither the UMAC or the LMAC processor is executing code to process host requests to read peripheral register values. This interface is described in figure 4.

`iwlwifi` also defines offsets (macros `HBUS_TARG_MEM_RADDR`, `HBUS_TARG_MEM_RDAT`, etc.) and functions (`iwl_trans_pcie_read_mem` and `iwl_trans_pcie_write_mem`) to access the chip memory. Of course these functions cannot be used to write to arbitrary memory locations at runtime but their use by functions such as `iwl_trans_pcie_txq_enable` indicates that some regions of the firmware are indeed writable from Linux.

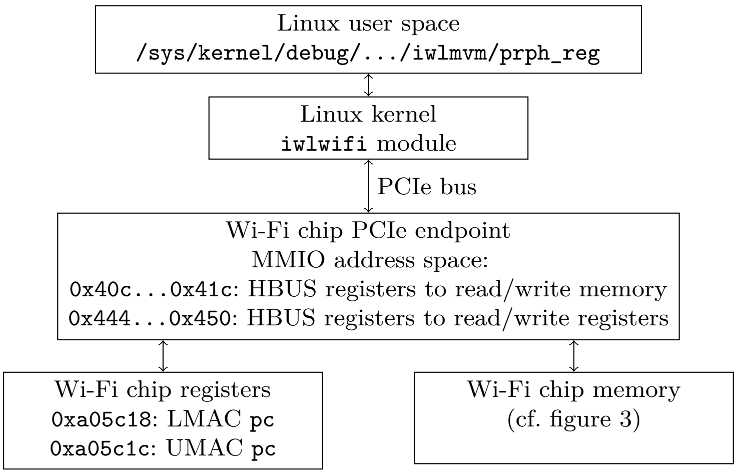

Fig. 4. Interaction between Linux debug filesystem and the Wi-Fi chip

Nevertheless, `iwlmvm/mem` in the debug filesystem does not use this interface. Instead the implementation of the read operation (in function `iwl_dbgfs_mem_read`[19]) boils down to calling `iwl_mvm_send_cmd(mvm, &hcmd);` with a *host command* in the parameter `hcmd`. This function calls `iwl_trans_pcie_send_hcmd` to enqueue a command in queues that the Wi-Fi chip reads using Direct Memory Access (DMA). This interface is shared with every command that the Linux kernel sends to the chip

[19] https://elixir.bootlin.com/linux/v5.11/source/drivers/net/wireless/intel/iwlwifi/mvm/debugfs.c#L1799

(for example to request scanning access points, to configure some radio properties, etc.) and we can expect that messages sent through it are processed by the firmware.

When `iwlwifi` and `iwlmvm` prepare a command for the Wi-Fi chip, they use a structure named `iwl_host_cmd`[20] where they fill the command ID and parameters. The identifiers consist of two bytes, defining a group of commands (enum `iwl_mvm_command_groups`[21]) and a command inside a group. For example, the command used to read memory is:

- group `DEBUG_GROUP = 0xf`,
- command `LMAC_RD_WR = 0` or `UMAC_RD_WR = 1`, to read memory from the LMAC or the UMAC processor.

This identifier is packed into a 4-byte structure `iwl_cmd_header`[22] before being sent to the chip. With this information, it should be possible to find the code processing such commands in the firmware.

Arbitrary Code Execution The host manages the chip through a set of commands mentioned previously. The command IDs as well as the associated request and response structures are declared in the kernel module source code.

The firmware implementation of these commands was reverse-engineered, allowing us to find undocumented commands. One of these commands (of ID `0xf1`) receives host data in 2 steps:

1. A first structure made of a size and a flag (`struct input { size_t count; int flag; }`) is received. The size field is actually the expected size of the next received data.
2. Data is then read directly on the stack, leading to a stack overflow if the size specified in the first command is larger than the size of the stack buffer.

In order to trigger the vulnerability, we based our exploit on `ftrace-hook`. It allows sending arbitrary commands to the chip by hijacking a single function from the Linux module: `iwl_mvm_send_cmd()`. The exploit works in 2 steps:

[20] https://elixir.bootlin.com/linux/v5.11/source/drivers/net/wireless/intel/iwlwifi/iwl-trans.h#L207
[21] https://elixir.bootlin.com/linux/v5.11/source/drivers/net/wireless/intel/iwlwifi/fw/api/commands.h#L32
[22] https://elixir.bootlin.com/linux/v5.11/source/drivers/net/wireless/intel/iwlwifi/fw/api/cmdhdr.h#L65

1. A shellcode is first put somewhere at a fixed address in the heap of the firmware using legit commands. Reverse engineering allowed us to discover a few commands which copy large amounts of data from the host to the heap without alteration, for later use. Optionally, the debugfs mechanism can be used to ensure that the shellcode is indeed written to the expected address.
2. The vulnerability is then triggered: the stack overflow vulnerability allows the attacker to take control of `pc` and redirect the execution to the shellcode previously put in the heap.

We developed a shellcode which enables the global *debug mode* flag. This flag is notably checked by the firmware `iwlmvm/mem` implementation to tell whether write access is allowed, which eventually allows us to read and write memory using this convenient debugfs mechanism.

This stack overflow vulnerability was successfully exploited in the firmware version `34.0.1`. This vulnerability doesn't exist anymore in the firmware version `46.6f9f215c.0`.

3.2 Secure Boot and bypassing it

Locating the Loader The previous sections presented how we interacted with Intel Wi-Fi chips from Linux and how the code is loaded from firmware files. During the study we wondered whether the verification of the authenticity of the code is implemented in hardware or in some code running on the LMAC or the UMAC processors. Indeed it is common for microcontrollers to have a *Boot ROM* with code which authenticates the loaded firmware before running it. If an Intel Wi-Fi chip had such code, how could we find it?

Actually on the studied chip, this is easy:
- the Linux kernel module can read the memory of the chip,
- and the module can also read the program counter registers (`pc`) of the chip processors.

We patched `iwlwifi` to dump parts of the memory and to record the `pc` values right before the firmware was loaded. We found out that most memory regions contain random data which change at every boot, except two areas:
- one between addresses `0x00402e80` and `0x00402fff`,
- one between addresses `0x00060000` and `0x00061eff`.

The second area contains valid ARCompact instructions and the recorded `pc` values alternate between `0x0006107e`, `0x00061092`, `0x00061098` and a few other addresses. So we knew we dumped some interesting code. Moreover the first instructions of this area include `mov sp, 0x00403000`, defining the *stack pointer* to the top of the first area.

The dumped code is quite small (4554 bytes) and, surprisingly, it does not include any implementation of RSA or SHA256 algorithms. How could it verify the firmware signature?

Studying more closely the data we got shows that at address `0x00061e00` is located the same RSA2048 public key as in the firmware file. This key is used by a function at `0x00060fa8`. After more analysis we found out that the dumped code uses this key with some hardware registers in the following sequence:

— Write `1` and `0` to the peripheral register located at `0x00a24b08`.
— Write `3` to `0x00a24b00`.
— Write the 256 bytes of the public key to `0x00a24900`, `0x00a24901`, etc.
— Write the 256 bytes of the firmware signature to `0x00a24800`, `0x00a24801`, etc.
— Write `1` to `0x00a2506c` and `0x00a25064`.
— Wait for the lowest bit of peripheral register located at `0x00a24b04` to become zero.
— Read the decrypted RSA signature from `0x00a24a00`.
— Write `1` to `0x00a20804`.

This code probably drives a coprocessor which decrypts RSA2048 signatures in PKCS#1 v1.5 format. Other peripheral registers are used in a similar way, to compute the SHA256 digest of the firmware being loaded. Such coprocessors are usually called *cryptographic accelerators* and it is normal to see one on a Wi-Fi chip, which could offload some cryptographic operations to dedicated hardware.

This new knowledge of the cryptoprocessor enabled looking for code referencing its addresses in the firmware. And indeed the UMAC code uses the cryptoprocessor in a similar way to verify some signatures, for example when processing `FW_PAGING_BLOCK_CMD` commands.

Bypassing Secure Boot Linux loads a firmware on the Wi-Fi chip by sending its sections. We previously described (in section 2.4) that it is not possible to directly modify the content of these sections. By reverse-engineering the code of the loader, we found the code which computed a SHA256 digest over all the sections. The loader needs to implement this to verify a RSA-2048 signature embedded in the first section (using a cryptoprocessor).

This code does not wait for the full firmware to be received before computing its digest, but updates the SHA256 state after each section is received. Does it mean that an attacker can modify a section after it has been verified? We patched the Linux kernel in order to send a section twice: once with the original content, and a second time with some modifications. This failed. The firmware started successfully but the modifications were ignored. Digging further, we discovered that the loader modifies some hardware registers of the chip after receiving a section. We suppose this locked some memory pages to make them no longer writable from Linux.

In short, when the firmware loader starts, Linux is allowed to write to most of the memory of the chip, and the memory progressively becomes read-only while the firmware is loaded. But the memory does not solely contain the firmware: it also contains the loader! And trying to write to the loader data actually works!!

More precisely, when we call Linux's function `iwl_trans_pcie_write_mem` to write some data at `0x00402e80` before loading the firmware, we manage to read the new data back (using `iwl_trans_pcie_read_mem`). The stack of the loader is located at this address, so it is possible to overwrite some return address to make the loader execute our code (which can be written using the normal firmware loading interface). The attack therefore consists in writing a modified firmware to the memory of the chip, replacing a return address with zero in the stack of the loader, and notifying the loader that the firmware is loaded. This works fine on the first Wi-Fi chip studied (Intel Dual Band Wireless AC 8260), but not on the second one (Intel Wireless-AC 9560 160MHz).

On the second chip, we observe that the value we read back after modifying the stack is successfully modified, but the loader seems to ignore it. Another thing was strange: despite the loader using some global variables in memory, we do not see these variables change when reading their values. We suppose this is caused by a caching mechanism: the content of the stack is used from a cache memory of the Wi-Fi chip. As the read/write access from the Linux driver modifies the physical memory directly without invalidating the cache, the chip ignores these modifications.

To fix the attack, we modified the firmware image in order to force cached data to be flushed to the memory. One way to achieve this consists in increasing the number of sections which are loaded by the chip. This number is actually present in the first section transmitted to the chip (the one which contains the signature). By declaring that the firmware contains 196 sections (listing 16), the behavior of the chip changes:

- When trying to load this firmware directly, the chip refuses to boot and a `SecBoot` message appears in the kernel log. This is expected, because the modified section is included in the signed data.
- When trying to load this firmware while overwriting a code address on the stack, the chip successfully boots.

```python
import struct

old_section = get_first_section("iwlwifi-9000-pu-b0-jf-b0-46.ucode")
new_section = (
    old_section[:0x284] +  # Header with RSA signature
    # Define 196 fake sections at address 0 with size 0.
    struct.pack("<I", 196) +
    struct.pack("<IIII", 7, 8, 0, 0) * 196
)
```

Listing 16. Extract of a Python script which modifies the first section

More precisely we identified in the dumped stack, at `0x00402fc0`, the code address `0x00060f7a`. This address is right after a function call,[23] in the code of the firmware (listing 17).

```
00060f70  f1 c0        push_s blink
00060f72  66 0c 8f ff  bl     FUN_000603d4  (initialize things)
00060f76  e6 0b 8f ff  bl     FUN_00060358  (compute SHA256)
  (the value at 0x00402fc0 is here)
00060f7a  7e 0d 8f ff  bl     FUN_000604f4  (verify RSA signature)
00060f7e  d1 c0        pop_s  blink
00060f80  e0 7e        j_s    blink
```

Listing 17. Attacked function of the Wi-Fi chip loader (ARCompact assembly)

We perform the attack by modifying the function `iwl_pcie_load_cpu_sections_8000`[24] (in the `iwlwifi` kernel module) to write zero to `0x00402fc0` (listing 18). This actually bypasses the call to the function which verifies the RSA signature and directly starts the loaded firmware.

```c
iwl_trans_grab_nic_access(trans);
unsigned int iterations;
for (iterations = 0; iterations < 70000; iterations++) {
    iwl_write32(trans, HBUS_TARG_MEM_WADDR, 0x00402fc0);
```

[23] In ARCompact, instruction `bl` peforms a *branch with link* operation, used to call a function.

[24] `https://elixir.bootlin.com/linux/v5.11/source/drivers/net/wireless/intel/iwlwifi/pcie/trans.c#L719`

```
    iwl_write32(trans, HBUS_TARG_MEM_WDAT, 0);
}
iwl_trans_release_nic_access(trans);
```

Listing 18. Loop added to iwlwifi to bypass the signature verification

Being able to load arbitrary code on a Wi-Fi chip greatly helps analyzing how it works. In the remaining parts of this article, we will present some experiments enabled by this access.

4 Use Cases and Practical Applications

4.1 Understanding the Paging Memory

Going beyond physical memory The studied firmware file defined a section at address `0x01000000` with 241664 bytes (cf. listing 8 in section 2.2). Contrary to the other sections, this one is not loaded directly in the memory of the chip. Instead, `iwlwifi` allocates specific buffers in the main memory and transmits their physical addresses to the chip, using a `FW_PAGING_BLOCK_CMD` command in function `iwl_send_paging_cmd`.[25] This means that this code is loaded once the LMAC and the UMAC processors have already been started. At this point, we wondered: where is this code stored in the Wi-Fi chip? How is it authenticated?

The second question is simple to answer: the implementation of the `FW_PAGING_BLOCK_CMD` command in the UMAC code (at address `0x80452184`) reads all the pages using DMA transfers and verify a RSA2048-SHA256 signature provided by a Code Signature Section. However, all DMA transfers target the same 4096-byte page on the memory of the chip, at `0x00447000`. So the data is not actually kept by the chip.

The host physical addresses of the blocks are saved in a structure `iwl_fw_paging_cmd`[26] at address `0xc0885774`. We retrieve the content of the structure from the chip using the debug filesystem (cf. section 3.1) and decode it according to the structure definition (listing 19).

```
struct iwl_fw_paging_cmd at 0xc0885774:
* flags = 0x303: 0x200=secured, 0x100=enabled, 3 pages in last block
* block_size = 15 (0x8000 = 32768 bytes/block, 8 pages/block)
* block_num = 8
Block addresses:
  Host phys 0x10b976000 = Code Signature Section
```

[25] `https://elixir.bootlin.com/linux/v5.11/source/drivers/net/wireless/intel/iwlwifi/fw/paging.c#L232`

[26] `https://elixir.bootlin.com/linux/v5.11/source/drivers/net/wireless/intel/iwlwifi/fw/api/paging.h#L22`

```
    Host phys 0x10b9f0000 = Paging mem 0x01000000
    Host phys 0x10b9f8000 = Paging mem 0x01008000
    Host phys 0x10ba00000 = Paging mem 0x01010000
    Host phys 0x10ba08000 = Paging mem 0x01018000
    Host phys 0x10ba10000 = Paging mem 0x01020000
    Host phys 0x10ba18000 = Paging mem 0x01028000
    Host phys 0x10ba20000 = Paging mem 0x01030000
    Host phys 0x10ba28000 = Paging mem 0x01038000
```

Listing 19. Extracting the configuration of the paging memory, from the chip

If the chip does not keep all pages when processing the `FW_PAGING_BLOCK_CMD` command, how is it able to use this memory? By accessing memory through the debug filesystem, we confirm that the memory located at addresses `0x01000000`, `0x01008000`, etc. is indeed readable and writable. The answer is: by using the Memory Management Unit!

Indeed the UMAC processor defined handlers for the exception vectors `TLBMissI` and `TLBMissD` (at addresses `0xc0080108` and `0xc0080110`) which occur when a memory access fails. These handlers integrate a complex state machine which loads the requested memory page from the host using DMA, in a memory area between `0x00422000` and `0x00447fff`. To confirm that the analysis is correct, we read the global variables used by this state machine, which include an array at `0x804508b8`. For example, in an experiment this array starts with the bytes `ff ff 10 ff 0b ff`. Every byte is related to a virtual memory page.

— The first byte is `0xff`, meaning that the first page (at `0x01000000`) is not currently mapped by the chip.
— The second byte was `0xff`, meaning that the page at `0x01001000` is not mapped.
— The third byte, `0x10`, means that the page at `0x01002000` is mapped at physical address `0x00422000 + 0x10*0x1000 = 0x00432000` of the chip. This is confirmed by reading the data stored at this address directly.
— etc.

The Wi-Fi chip has space for 38 4KB-pages and the firmware defines 59 pages so it is impossible to load all of them simultaneously. Moreover these regions contain global variables which are updated by the firmware. How does the firmware keep the modified bytes when some room is needed to load a newly requested page? By sending another DMA request to write the modified bytes to the host memory. And indeed, using chipsec to read the host physical memory, we observe that the buffer allocated for this Paging memory is modified.

In short, the code running on the UMAC processor uses its MMU to extend its memory capacity, by relying on DMA transfers with the host memory to store the data which do not fit.

Protecting the integrity of the Paging Memory Once we understood the mechanism of the Paging Memory, we tried an obvious attack: we modified a byte in the host memory and made the Wi-Fi chip request it by issuing a command to read memory. This failed (the UMAC reported a `NMI_INTERRUPT_UMAC_FATAL` error and `iwlwifi` restarted the chip), and we did not understand why. How is the integrity of the Paging Memory guaranteed?

The function which handles command `FW_PAGING_BLOCK_CMD` performs some operations that we first overlooked:

— It writes the address `0x8048f400` in the peripheral register `0x00a0482c` and `0x1000` in `0x00a0480c`.

— Before receiving a page (to verify the signature), it writes the physical address of the received page in `0x00a04808`, the index of the virtual page in `0x00a04804`, and `1` in `0x00a04800`.

— After receiving a page, it waits for some bits in the peripheral register `0x00a04800` to become set.

These registers are also used near the code which performs DMA requests. Maybe they are used to compute some digest of the data? Where would these digests be stored? Maybe at the first address which is used, `0x8048f400` (which is the physical address `0x0048f400`). Surprisingly, the content at this location is not readable using the debug commands used by `iwlmvm/mem`. This limitation is due to a check which forbade reading any data between `0x0048f000` and `0x0048fffff`. Fortunately we are not stopped by this, as we are able to load a modified firmware without this restriction.

After more experiments, we discover that `0x0048f400` holds a table of 32-bit checksums for each 4 KB page of the Paging Memory. The checksum of the first page (whose virtual address is `0x01000000`) is located at `0x0048f400`, the checksum of the second one at `0x0048f404`, etc. In an experiment, we obtain that:

— the checksum of a page with 4096 zeros is `00 00 00 00`,

— A page with 4095 zeros and `01` has checksum `11 ac d8 7f`

— A page with 4095 zeros and `02` has checksum `22 58 b1 ff`

— A page with 4095 zeros and `03` has checksum `33 f4 69 80`

— A page with 4095 zeros and `04` has checksum `c9 b0 62 ff`

These values are not so random: they are linear with the input! By XOR-ing the results of the lines with `01` and `02`, we obtain the result written in the line with `03`. Also taking the bytes of the line with `01` and shifting them left one bit gives the result of the line with `02`, with a bit moved from `ac` to `b1`. Continuing this trail, we found out that the computation involved a 32-bit Linear Feedback Shift Register (LFSR) on the input bytes considered as a sequence of 32-bit Little Endian integers, with polynomials `0x10000008d`. But it is not only an LFSR, as values change every time the chip is reset.

More experiments reduce the algorithm to the Python function presented in listing 20. Discussions within our awesome team made us understand we were watching a scheme named *Universal Message Authentication Code*, and our implementation actually matches the example written on Wikipedia.[27]

```python
def checksum(page, secret_key):
    # Return the checksum of a 4096-byte page with a 1024-int key
    result = 0
    for index_32bit_word in range(1024):
        page_bytes = page[index_32bit_word*4:index_32bit_word*4+4]
        page_value = int.from_bytes(page_bytes, "little")

        sec = secret_key[index_32bit_word]
        for bit_pos in range(32):
            if page_value & (1 << bit_pos):
                result ^= sec

            # Linear Feedback Shift Register with 0x10000008d
            if sec & 0x80000000:
                sec = ((sec & 0x7fffffff) << 1) ^ 0x8d
            else:
                sec = sec << 1
    return result
```

Listing 20. Python implementation of the checksum algorithm used to ensure the integrity of the Paging Memory

This algorithm is quite weak in this case: in our study we were able to request the checksums for pages containing bytes `01 00...00`, `00 00 00 00 01 00...00`, etc., which directly leaks the 1024 integers used in the secret key. With this key, it is simple to modify a page in a way which does not modify the checksum.

In short, the integrity of the Paging Memory, which prevents the Linux kernel from modifying its content, is guaranteed by a 32-bit checksum algorithm, a secret key generated each time the chip boots and the impossibility to read the stored checksums (we achieved this by compromising

[27] `https://en.wikipedia.org/wiki/UMAC#Example` (accessed on 2022-01-17)

the integrity of the firmware beforehand). So we did not discover a vulnerability there, but a way to leverage future arbitrary-read vulnerabilities into arbitrary code execution on the Wi-Fi chip.

4.2 Instrumentation, Tooling and Fuzzing

Debugger A debugger has been developed to make the dynamic analysis of some pieces of firmware code easier.

A shellcode is first written in a part of uninitialized firmware memory. The first instruction of the debugged code is modified to redirect the firmware execution to the shellcode. The shellcode waits in a loop for custom commands from the host to:

- read and write LMAC and UMAC CPU registers,
- read and write from/to memory,
- resume the execution of the firmware.

In order to make debugging faster, an experiment has been conducted with QEMU to redirect the execution of the debugged code in QEMU, and forward the memory and register accesses to the debugger. Slight modifications of QEMU's core are required to allow QEMU's plugin system to write to memory.

Nevertheless, a few issues are encountered:

- Firmware timers are triggered at regular intervals, disturbing debugging. Disabling these timers leads to unexpected side effects.
- *Extension Core Registers* are modified by the hardware even if executed instructions don't reference them.
- A few ARC700 instructions must be fixed or added to QEMU.

Traces Once secure boot is disabled and unsigned firmware can be loaded, the firmware can be patched to change the behavior of some functions. In order to facilitate firmware analysis, a *tracing* mechanism was developed to tell dynamically which functions are executed.

The list of all firmware functions is retrieved thanks to a custom Ghidra script. These functions are patched to replace the first prologue instruction (`push_s blink`) with the instruction `trap_s 0`. The code of the associated interrupt handler is replaced to store the address of the instruction which triggered the interrupt, in a buffer shared with the host.

This mechanism allows to gather every function executed by the firmware, but it's slightly more complicated on the UMAC processor:

- The instruction `trap_s 0` triggers an *unrecoverable machine check exception.* An invalid instruction seems to trigger a different interrupt handler, but it can also be replaced to store the faulty instruction.

— Some functions can't be instrumented because triggering an interrupt during their execution seems to lead to a *machine check exception*, probably because of a double fault.

On-Chip Fuzzing In order to find vulnerabilities, the code of the firmware has been modified to hook some functions related to Wi-Fi packets parsing and fuzz randomly input parameters. While it indeed leads to crashes, these functions use hardware registers which make crashes bound to the state of the card. Crashes are thus difficult to reproduce. Moreover, some checks on packet validity seem to be done by the hardware, before packets are handled by the firmware. These crashes can't be reproduced through remote frame injection.

4.3 Initial crash analysis

Further analysis showed that the initial bug that led to this study isn't exploitable. It's a crash of the LMAC CPU because the firmware doesn't expect to receive *TDLS Setup Request* commands from the host, while the device seems to support TDLS (Tunnel Direct Link Setup, listing 21).

```
$ iw phy | grep -i tdls
                * tdls_mgmt
                * tdls_oper
        Device supports TDLS channel switching
```

Listing 21. Querying TDLS support on the first studied chip

Several users reported this crash on the Kernel Bug Tracker[28] and the bug is actually fixed since firmware update 36.[29] As explained by the maintainer in this comment:

> *Anyway, the new firmware has the fix: we don't advertise TDLS anymore.*

It's worth noting that even if a firmware update is available, some Linux distributions don't include it. For instance, this crash can reliably be triggered remotely with a single Wi-Fi packet targeting an up-to-date Ubuntu 18.04, leading to the reboot of the Wi-Fi firmware.

[28] `https://bugzilla.kernel.org/show_bug.cgi?id=203775`

[29] `https://git.kernel.org/pub/scm/linux/kernel/git/firmware/linux-firmware.git/commit/?id=5157165f22041346b3a82e12ba072d456777fdf2`

5 Conclusion

This journey studying Intel Wi-Fi chips was incredible. We did not expect to bypass the secure boot mechanism of the chip, and this achievement opened the door to many new possibilities. Most importantly, we can now instrument the firmware to better understand some undocumented parts.

While this document is quite large, it does not include some work which was also done: studying how the WoWLAN (Wake-on-Wireless Local Area Network) feature is implemented, how ThreadX operating system is used by the UMAC code, how the chip really communicates with the host using DMA, how fragmented Wi-Fi frames are parsed, how the LMAC configures a MPU (Memory Protection Unit), etc. In the future we will likely continue looking for vulnerabilities in the Wi-Fi radio interface. Future work can also include how the Wi-Fi part of the chip interacts with the Bluetooth part. Indeed, all studied chips also provide a Bluetooth interface which seems to require some coordination with the Wi-Fi firmware to operate. Another area of interest could be the interaction between the Wi-Fi chip and Intel CSME (Converged Security and Management Engine) for AMT (Active Management Technology): the `iwlwifi` module was modified in Linux 5.17-rc1 (released in January 2022) to document how this works.[30]

We would like to thank our employer Ledger for letting us work on this exciting topic, Intel developers for providing useful documentation in `iwlwifi` and Microsoft for publishing the ThreadX source code.[31]

Finally, we hope that the publication of this article will lay the groundwork for helping other researchers to dive into that topic.

A Appendix: glossary

— BAR: Base Address Register
— CSS: (probably) Code Signature Section (a firmware section which contains metadata about other sections, including a signature)
— DMA: Direct Memory Access (a way to transmit data between two devices without running code on a processor)
— DCCM: Data Close Coupled Memory (some kind of memory)
— LMAC: Lower Medium Access Controller

[30] `https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=2da4366f9e2c44afedec4acad65a99a3c7da1a35`
[31] `https://github.com/azure-rtos/threadx/`

— MMIO: Memory-Mapped Input Output
— SRAM: Static Random Access Memory (some kind of memory)
— UMAC: Upper Medium Access Controller

References

1. ARC. Arc 700 memory management unit reference, 2008. `http://me.bios.io/images/7/73/ARC700_MemoryManagementUnit_Reference.pdf`.
2. Gal Beniamini. Over the air: Exploiting broadcom's wi-fi stack (part 1), 2017. `https://googleprojectzero.blogspot.com/2017/04/over-air-exploiting-broadcoms-wi-fi_4.html`.
3. Andrés Blanco and Matías Eissler. One firmware to monitor 'em all, 2012. `http://archive.hack.lu/2012/Hacklu-2012-one-firmware-Andres-Blanco-Matias-Eissler.pdf`.
4. Guillaume Delugré. How to develop a rootkit for broadcom netextreme network cards. RECON, July 2011. `https://recon.cx/2011/schedule/events/120.en.html`.
5. Louis Granboulan. cpu_rec.py, un outil statistique pour la reconnaissance d'architectures binaires exotiques. SSTIC, June 2017. `https://www.sstic.org/2017/presentation/cpu_rec/`.
6. Nicolas Iooss. Analyzing arcompact firmware with ghidra. SSTIC, June 2021. `https://www.sstic.org/2021/presentation/analyzing_arcompact_firmware_with_ghidra/`.
7. Yuval Ofir Omri Ildis and Ruby Feinstein. Wardriving from your pocket, 2013. `https://recon.cx/2013/slides/Recon2013-Omri%20Ildis%2C%20Yuval%20Ofir%20and%20Ruby%20Feinstein-Wardriving%20from%20your%20pocket.pdf`.
8. Yves-Alexis Perez, Loïc Duflot, Olivier Levillain, and Guillaume Valadon. Quelques éléments en matière de sécurité des cartes réseau. SSTIC, June 2010. `https://www.sstic.org/2010/presentation/Peut_on_faire_confiance_aux_cartes_reseau/`.
9. Julien Tinnès and Laurent Butti. Recherche de vulnérabilités dans les drivers 802.11 par techniques de fuzzing. SSTIC, June 2007. `https://www.sstic.org/2007/presentation/Recherche_de_vulnerabilites_dans_les_drivers_par_techniques_de_fuzzing/`.