h550 project - Security assessment of an AP

Esteban Aguililla Klein

January 22, 2025

Contents

1 Abstract **2**

2 Introduction **2**

3 Scope **2**
3.1 Lab setup . . . 3
3.1.1 Equipment . . . 3

4 Reconnaissance **4**
4.1 Physical . . . 4
4.2 Bootloader . . . 6
4.2.1 Boot log . . . 6
4.2.2 U-boot . . . 6
4.3 Console . . . 7
4.4 Network . . . 7

5 Exploitation **7**
5.1 Dumping the firmware . . . 7
5.1.1 Through the bootloader . . . 7
5.1.2 Through the console . . . 8
5.2 CVE-2014-0659 . . . 8
5.3 Reverse Engineering . . . 9

1 Abstract

Closed source firmware cannot be trusted. Therefore, we will show how to hack an access point and search for exploitable vulnerabilities from the hardware to the software. Doing, so we will take a look at how U-boot and BusyBox shells can be used and leveraged to our advantage. Afterwards, we will showcase a level 10[1] known vulnerability (which is in fact a backdoor placed by the constructor) that allows remote code execution and how it was found.

2 Introduction

Nowadays most IOT devices contain some flavor of a vulnerability, backdoor or critical bug. However, those problems in some cases are never fixed as IoT devices are dependent on their vendors for firmware update due their code being proprietary (in most cases). Furthermore, some users might forget (or not know) that their device's software is not updated and by extension, vulnerable. This paper takes the approach of a guide on how to perform basic hardware hacking on everyday embedded devices. What should be taken away from this work is the process which aims to be as generic as possible and not the results as the chosen target is now discontinued and irrelevant. Therefore, we will start by setting the scope of our analysis then, do reconnaissance under those constraints followed by the exploitation of our findings.

3 Scope

The device is an access point from Cisco released in October 2008, aimed at small businesses. This products has been discontinued in 2019 and its sales ended back in 2014.

(a) Product picture (b) Bottom (c) Inside

Figure 1: Pictures of of the WAP4410N

The device can be powered up using a barrel adapter or through PoE which hints at its purpose being ease of batch deployment. On the network side,

[1] https://nvd.nist.gov/vuln/detail/CVE-2014-0659

it support 802.11n, 802.11g and 802.11b. With the time period in mind, for 802.11b and g, the only supported encryption protocol was WEP (64/128 bits in this case). Also, firmware updates were handled through the web porta : there is no I/O on the device. On figure 1c, we can see a cable going outside of the device, this is the UART access which was enabled by default.

3.1 Lab setup

The testing setup is made of the WAP4410N, a router and a computer : for simplicity sake, the router is out of scope from this analysis and all its security features are disabled. The computer has a wireless connection to the AP.

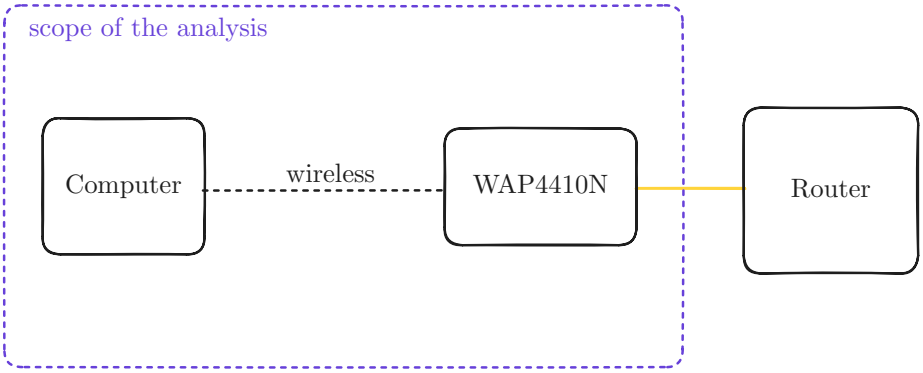


Figure 2: Lab setup

3.1.1 Equipment

The following table contains all the software and hardware required for basic hardware hacking.

software	hardware
Wireshark, nmap GNU Screen, xxd, dd, binwalk squashfs-utils radare2, ghidra	FT232 board USB-A UART adapter Multimeter Dupont Cables

Figure 3: List of used software/hardware

At first, we intended to use a SPI programmer (CH341a) to dump the firmware more easily using flashrom. However, as we will see in a later section, it was not feasible in this context without specialized hardware tools. On the software side, there are no real limitations as most of the state of the art tools are free and open source (with the exception of IDA).

4 Reconnaissance

This section was inspired by one of the steps cyber kill chain which will vaguely be followed throughout this paper. This is the most important part of any hardware hacking project : the more information about the target we have the easier finding vulnerabilities will become.

4.1 Physical

Teardown The device is held together by four 8-points screws under rubber feets which indicates that it wasn't meant to be opened (nor repaired . . .). Once inside, most of the device is empty and everything is available without further need to dismantle it.

IC reconnaissance At this point we can start looking for interesting ICs like flash memory or processors :

- reference : MX29LV640DBTC-90G
- format : 48 TSOP
- manufacturer : Macronix
- size : 8MB

- family : Atheros 9100
- reference : Atheros9132
- manufacturer : Qualcomm
- architecture : MIPS 24K V7.4 32bits
- endianness : big endian
- bogomips : 265.21

The MIPS architecture is very common on embedded devices and very easy to reverse engineer its assembly. However, this makes it impossible for us to use frida as MIPS is only available in the pro version. Then, regarding the NAND flash, as hinted before, reading a 48 TSOP IC requires specialised and expensive hardware. Therefore, using an SPI programmer to dump it is out of the question.

Finding the UART This is the most important step of the physical reconnaissance : the UART allows us to communicate with the device from a computer. To find it, we have to usually look for 3 to 4 pins next to each other.

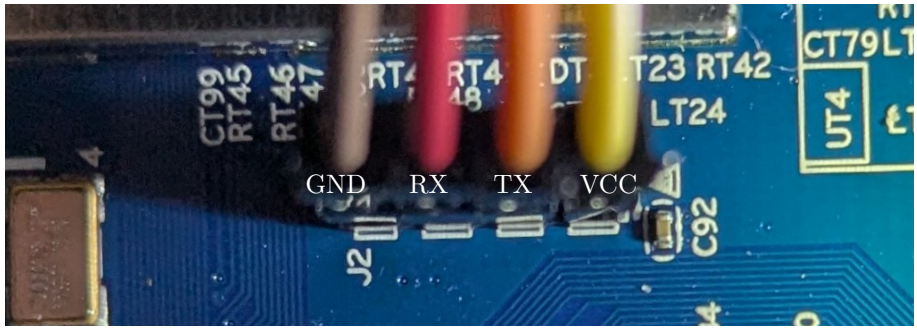


Figure 4: UART connected through Dupont cables with labeled ports

On the device, there are 4 male header pins that might be the UART. We can check using a multimeter. First, we must find the ground, to do so, we can look at any IC datasheet and find a ground pin then, we can try to find a grounded metal to connect to it instead of a pin by checking for continuity. After having found a ground connection we can start taking measurements of our candidate,

pin	R_{VSS}	V	info
1	$8.6k\Omega$	$\approx 3.3V$	VCC
2	$\infty k\Omega$	$\approx 0V - VCC$	TX
3	$\infty k\Omega$	$\approx 0V$	RX
4	0 k Ω	$0V$	GND

Figure 5: Multimeter measurements

The voltage goes from 0V to VCC, at startup because of the boot log outputs a lot of data. The RX has a voltage of 0 as its used to receive data and doesn't send any. The next step is to try connecting the UART to the computer using our TTL,

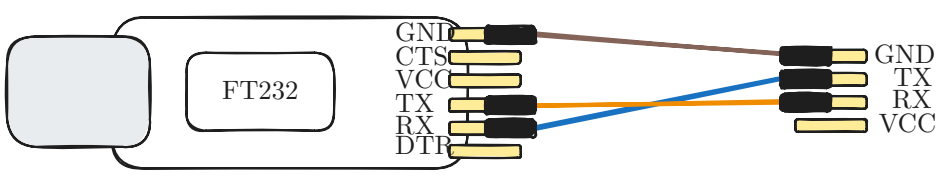


Figure 6: UART to USB-A

Regarding the baudrate, we can try 115200 as its the most common value,

```
screen /dev/ttyUSB0 115200
```

This is indeed the correct baudrate however, if that wasn't the case we would have had to find it by trial and error or by using a logic analyzer.

4.2 Bootloader

4.2.1 Boot log

First, we will take a look at the boot log that is displayed before getting access to any console as analyzing it provides a lot of information about the hardware, software, ... One such information is the bootloader : U-Boot 1.1.4 which is a very common and open source bootloader.

```
arg 1: console=ttyS0,115200
arg 2: root=31:02
arg 3: rootfstype=jffs2
arg 4: init=/sbin/init
arg 5: mtdparts=ar9100-nor0:256k(u-boot),
        64k(u-boot-env),
        6464k(rootfs),
        1280k(uImage),
        64k(nvram),
        64k(calibration)
arg 6: mem=32M
```

From the capture snippet above we see that the rootfs uses jffs2 however, this is not the case.

```
Kernel command line: console=ttyS0,
        115200 root=31:02 rootfstype=squashfs init=/sbin/
            init mtdparts=ar9100-nor0:256k(u-boot),
        64k(u-boot-env),
        6464k(rootfs),
        1280k(uImage),
        64k(nvram),
        64k(calibration)
        mem=32M
```

This snippet comes after the previous ones and tells us that rootfs uses squashfs. This information can be confirmed later using the console and/or by analyzing the dumped firmware. Usually an embedded device has two file systems with overlayfs : the union of two filesystems where the lower is read only (squashfs) and the upper read write (jffs2) for persistency. However, the device only uses squashfs, the configuration files are saved on non volatile memory through a program found in */usr/sbin/nvram*

4.2.2 U-boot

At some point we are prompted to interrupt the auto boot : by doing so, we get the U-boot console which allows us to get more information on the device,

```
ar7100> bdinfo
...
flashstart  = 0xBF000000
```

```
flashsize    = 0x00800000
flashoffset  = 0x0002F690
...
baudrate     = 115200 bps
```

We now know the memory addresses of the start and of the flash storage : this will come handy during the firmware extraction in 5.1.1. There are other useful commands available to us such as tftpboot which allows us to boot over the network by serving an image on our computer. Assuming that the console is protected, this could allow us to get an unrestricted console by providing our own patched image (or using an open source one). Other than that, there are custom commands for testing purposes.

4.3 Console

If we leave the autoboot alone, we get into a root Linux ash shell. There are device specific CLI commands and a reduced busybox. There is a fwversion command which tell us that the firmware version is 2.0.4.2. As we are already root, we can't really do any privilege escalation. Furthermore, as the device only has a squashfs3 file system we can't get any persistency.

4.4 Network

We can run a port scan on the device using nmap,

```
Nmap scan report for wap86eb04 (192.168.1.3)
Host is up (0.012s latency).
Not shown: 65532 closed tcp ports (reset)
PORT        STATE        SERVICE
80/tcp      open         http
443/tcp     open         https
32764/tcp   open    unknown
MAC Address: CC:EF:48:86:EB:04 (Cisco Systems)
```

The http(s) ports are used for the web administrative portals : when using the http web portal the credential are sent through base64. However, this isn't really a problem as the http port will most likely be disabled by any competent IT staff in real use cases

5 Exploitation

In this section we will see how to exploit the device

5.1 Dumping the firmware

5.1.1 Through the bootloader

This is the most "classical" technique used to dump firmware because of its simplicity and lack of technical and material requirements. To perform it, we will leverage the memory display command using the addresse from 4.2.2 and capturing the output. The size is written in decimal and divided by 4 as it reads by blocks of dwords. When the read is done, the output has to be cleaned into a binary format (there is a script in the repo). However, this technique didn't work in our case. The reason, is unclear as the output is not entirely unreadable when using binwalk : some possible reasons might be,

- glitchy cable
- this version of U-Boot has a problem with memory display
- out of bound area on the NAND flash
- endianness problem

5.1.2 Through the console

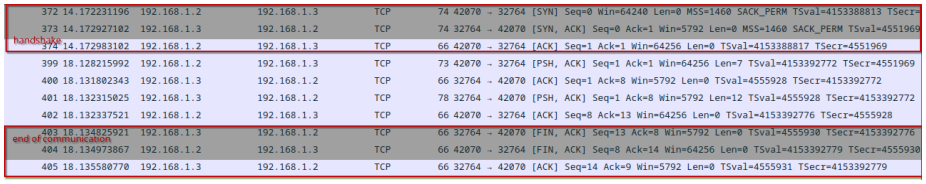
To do so, we have to update the version of busybox : we will download pre-compiled busybox binary and serve it to our device through an ftp server. We download it in the /tmp directory as it the only location in a read only filesystem where we can write a file. From there, we use netcat to upload the mtd partitions : concatenate the partitions together to make the whole firmware image :

```
                  /home/aaaaaa/aaaaaaaaa/aaa/aaaaaaaaaaaa/dump/firmware2.0.4.2.bin
--------------------------------------------------------------------------------------------------
DECIMAL                           HEXADECIMAL                       DESCRIPTION
--------------------------------------------------------------------------------------------------
158992                            0x26D10                           CRC32 polynomial table, big
                                                                    endian
327680                            0x50000                           SquashFS file system, big
                                                                    endian, version: 3.0,
                                                                    compression: unknown, inode
                                                                    count: 794, block size: 65536,
                                                                    image size: 4761817 bytes,
                                                                    created: 2011-05-13 10:54:02
6946816                           0x6A0000                          uImage firmware image, header
                                                                    size: 64 bytes, data size:
                                                                    875547 bytes, compression:
                                                                    gzip, CPU: MIPS32, OS: Linux,
                                                                    image type: OS Kernel Image,
                                                                    load address: 0x80002000,
                                                                    entry point: 0x801C2000,
                                                                    creation time: 2011-05-13
                                                                    10:51:49, image name: "Linux
                                                                    Kernel Image"
8269351                           0x7E2E27                          PEM private key
8270238                           0x7E319E                          PEM certificate
--------------------------------------------------------------------------------------------------
```

with this we can do some reverse engineering however, as the rootfs uses squashfs3 we can't patch binaries on the fly : we need to upload the entire firmware therefore we didn't path any binary.

5.2 CVE-2014-0659

This CVE is a backdoor planted by SerComm. To try it, we ping the 32764 using netcat : this opens a prompt in which after entering some random data and press enter will answer with ScMM, when analyzing the traffic we can notice



372	14.172231196	192.168.1.2	192.168.1.3	TCP	74 42070 → 32764 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 SACK_PERM TSval=4153388813 TSecr=
373	14.172927102	192.168.1.3	192.168.1.2	TCP	74 32764 → 42070 [SYN, ACK] Seq=0 Ack=1 Win=5792 Len=0 MSS=1460 SACK_PERM TSval=4551969
374	14.172983102	192.168.1.2	192.168.1.3	TCP	66 42070 → 32764 [ACK] Seq=1 Ack=1 Win=64256 Len=0 TSval=4153388817 TSecr=4551969
399	18.128215992	192.168.1.2	192.168.1.3	TCP	73 42070 → 32764 [PSH, ACK] Seq=1 Ack=1 Win=64256 Len=7 TSval=4153392772 TSecr=4551969
400	18.131802343	192.168.1.3	192.168.1.2	TCP	66 32764 → 42070 [ACK] Seq=1 Ack=8 Win=5792 Len=0 TSval=4555928 TSecr=4153392772
401	18.132315025	192.168.1.3	192.168.1.2	TCP	78 32764 → 42070 [PSH, ACK] Seq=1 Ack=8 Win=5792 Len=12 TSval=4555928 TSecr=4153392772
402	18.132337521	192.168.1.2	192.168.1.3	TCP	66 42070 → 32764 [ACK] Seq=8 Ack=13 Win=64256 Len=0 TSval=4153392776 TSecr=4555928
403	18.134825921	192.168.1.3	192.168.1.2	TCP	66 32764 → 42070 [FIN, ACK] Seq=13 Ack=8 Win=5792 Len=0 TSval=4555930 TSecr=4153392776
404	18.134973867	192.168.1.2	192.168.1.3	TCP	66 42070 → 32764 [FIN, ACK] Seq=8 Ack=14 Win=64256 Len=0 TSval=4153392779 TSecr=4555930
405	18.135580770	192.168.1.3	192.168.1.2	TCP	66 32764 → 42070 [ACK] Seq=14 Ack=9 Win=5792 Len=0 TSval=4555931 TSecr=4153392779

Figure 7: Capture of the exploit traffic

the data that we sent and as the answer we get, 53 63 4d 4d ff ff ff ff 00 00 00 00 which translates to ScMM. Based on the work of Eloi Vanderbken : you can use this exploit to get remote code execution on the device. Going into some reverse engineering of the vulnerable binary.

5.3 Reverse Engineering

The vulnerability is located int he **scfgmgr** file : we know this because it mas mentioned by the discoverer of the vulnerability. However, to find it naturally, one can look at the process list and analyze all the programs that sound like they don't belong there. The backdoor opens a new child process for each new connection to its socket and starts an alarm of 10s during which you have to send the correct character sequence :

1. Checks if the response starts with the correct code *(ScMM)* and is followed by and identifier (we will call it **id**) and optionally an argument
2. The program matches the **id** and, if it matches with a known **id** it performs the corresponding action *eg. there is an id for restarting the device*
3. The child process terminates

One of those *id*, opens a pipe that calls for **popen** with the user input as argument and returns the output through the socket. In the snippet above,

```
read_ret = (char *)malloc(1);
*read_ret = '\0';
*output = read_ret;
__stream = popen(command,"r");
```

Figure 8: Snippet of vulnerable code

command is a user input and, the resulting stream from the **popen** command

is read into the **read_ret** which is pointed by **output** : a function argument. After, returning from the pipe, **output** is written to the socket.