

THE FAKE CISCO

Hunting for backdoors in
Counterfeit Cisco devices

Dmitry Janushkevich

F-Secure Consulting, Hardware Security Team

Version 1.0, July 2020

1 INTRODUCTION

Producing counterfeit products is, and always has been, a great business if you don't mind being on the wrong side of the law. There's no need to invest in a costly R&D process, and no need to select the best performing and looking materials; the only criterion is the cost of manufacture. This is why we see many imitations of expensive products on the market, and are likely to continue to see them being made and sold at a fraction of original's price.

Network hardware designed, manufactured, and sold under the Cisco brand is a perfect example of this. Having an excellent reputation because of their great engineering, these products sell at a premium price point. Naturally, this encourages some to try and produce counterfeits as it's a way of making easy money. Stories of such exploits abound in the media: a gang reportedly exporting¹ US\$ 10 million worth of gear to the US, the FBI seizing shipments² of fake hardware, and court rulings being issued³ to stop the manufacturers. What does Cisco do to combat fraud? Actually, a lot. Cisco has a dedicated Brand Protection organization whose purpose is to defend against counterfeit and gray market activities. They partner with customs teams and regional governments all over the world with success. In April 2019, they seized \$626,880 worth of counterfeit Cisco products in one day.⁴ However, despite successful operations Cisco hasn't been able to stop fraud fully. If there's an opportunity to make a fast buck, there'll always be someone willing to take the risk.

In fall 2019, an IT company found some network switches failing after a software upgrade. The company would find out later that they had inadvertently procured suspected counterfeit Cisco equipment. Counterfeit devices quite often work smoothly for a long time, which makes it hard to detect them. In this particular case, the hardware failure initiated a wider investigation to which the F-Secure Hardware Security team was called and asked to analyze the suspected counterfeit Cisco Catalyst 2960-X series⁵ switches. This initiated a research project with the following goals:

- Verify no extra functionality such as "backdoor access" was introduced.
- Understand how and why counterfeit devices bypass the platforms authentication security control.

Naturally, as it's not easy to tell genuine and counterfeit devices apart, to verify whether any kind of "backdoor" functionality existed was also not easy, as it required a considerable amount of technical investigative work. Ultimately, we concluded, with a reasonable level of confidence, that no backdoors had been introduced. Furthermore, we identified the full exploit chain that allowed one of the forged products to function: a previously undocumented vulnerability in a security component which allowed the device's Secure Boot restrictions to be bypassed.

This paper details the process which led to this conclusion and shares the technical knowledge gained during this journey.

¹ <https://www.pcworld.com/article/2920032/uk-gang-arrested-for-exporting-10-million-of-fake-cisco-gear-to-us.html>

² <https://www.infoworld.com/article/2653167/fbi-worried-as-dod-sold-counterfeit-cisco-gear.html>

³ <https://www.sdxcentral.com/articles/news/cisco-wins-latest-battle-in-war-against-chinese-counterfeiters/2019/12/>

⁴ <https://blogs.cisco.com/partner/perform-transform-and-protect>

⁵ <https://www.cisco.com/c/en/us/support/switches/catalyst-2960-x-series-switches/series.html#-tab-documents>

While in this case no "backdoors" were identified, the fact the security functions were bypassed means the security posture of the device was weakened. This could allow attackers who have already gained code execution via a network-based attack, for example, an easier way to gain persistence, and therefore impact the security of the whole organization.

1.1 Acknowledgements

This paper is the result of a huge team effort. The author would like to acknowledge Andrea Barisani's contribution, who was the first point of contact for the team and started the initial investigative work. Thanks also go to Daniele Bianco and Andrej Rosano, who worked on the initial investigation. Furthermore, the author would like to thank Thierry Decroix for numerous edits and reviewing this paper.

1.2 Disclaimer

As this work presents the results of practical research, some of the information that appears may be insufficiently precise or incorrect. Please proceed at your own risk.

1.3 Device details

The following table details the devices the team had access to. The Genuine device was procured from an authorized distributor and the manufacturer confirmed it was genuine.

Device type	Name	SW version
WS-2960X-48TS-L V05	Genuine	c2960x-universalk9-mz.152-2.E7
WS-2960X-48TS-L V01	Counterfeit A	c2960x-universalk9-mz.150-2.EX5 (as provided by the source)
WS-2960X-48TS-L V01	Counterfeit B	c2960x-universalk9-mz.152-4.E7 (upgraded, resulting in breakage)

The devices will be referred to by their names where required.

CONTENTS

- 1 INTRODUCTION..... 1**
 - 1.1 Acknowledgements..... 2
 - 1.2 Disclaimer 2
 - 1.3 Device details..... 2

- 2 ANALYSIS..... 4**
 - 2.1 Symptoms 4
 - 2.2 Exterior differences 4
 - 2.3 Board analysis 5
 - 2.4 Boot log acquisition and analysis 12
 - 2.5 Content extraction from live systems 13
 - 2.6 Direct Flash content extraction 15
 - 2.7 Flash content analysis 16
 - 2.8 Bootloader analysis..... 20
 - 2.9 HBOOT patch analysis 22
 - 2.10 SLIMpro analysis 24

- 3 CONCLUSIONS..... 30**

- 4 ABOUT THE AUTHOR..... 31**

- 5 ABOUT F-SECURE HARDWARE SECURITY TEAM 31**

- 6 APPENDICES..... 32**
 - 6.1 The SoC..... 32
 - 6.2 The MZIP file format..... 35
 - 6.3 The AMCC file format 36
 - 6.4 Software signatures and keys..... 36

2 ANALYSIS

2.1 Symptoms

The biggest indication a 2960X device is a counterfeit is that it becomes inoperable after a software upgrade is performed. This also happened to the victim company and the devices had to be replaced. During negotiating the replacement with the vendor, the company found out they had unknowingly bought counterfeit devices. Moreover, the CISO was brought in to initiate investigations as to whether the company's networks had been compromised.

While the device lost its primary function as a network switch when the software upgrade was installed, it could still be accessed via the console. The following message was then displayed on the console during boot:

```
[Date Time]: %ILET-1-AUTHENTICATION_FAIL: This Switch may not have been manufactured by Cisco or with Cisco's authorization. This product may contain software that was copied in violation of Cisco's license terms. If your use of this product is the cause of a support issue, Cisco may deny operation of the product, support under your warranty or under a Cisco technical support program such as Smartnet. Please contact Cisco's Technical Assistance Center for more information.
```

Reverting the software version did not fix the problem, likely pointing to evidence of data being overwritten during the update process.

2.2 Exterior differences

Because clones and packaging are getting more realistic, many people don't realize they have counterfeit network equipment until it's installed and begins acting strangely. However, it is possible to spot minor differences in the visual appearance of the suspected counterfeits through comparison with a known-genuine device. Presented below are the most prominent differences we found during our investigations.

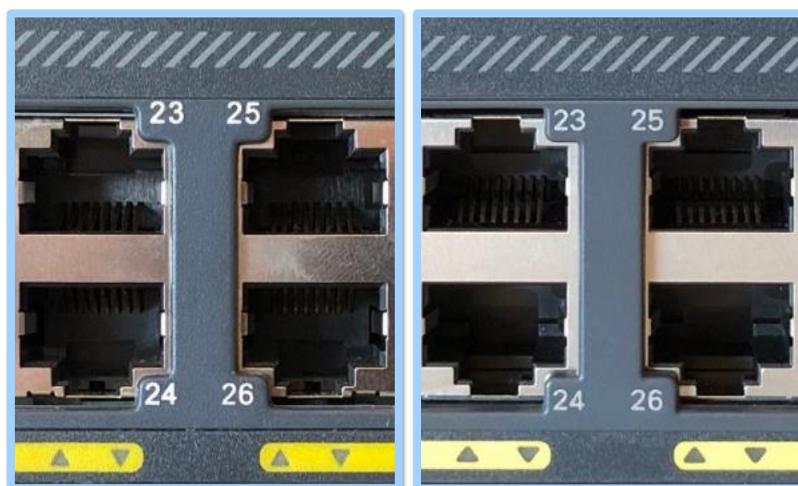


Figure 1. The suspected counterfeit switch (on the left) has port numbers in bright white, while the known genuine device has them in grey. The text itself is misaligned. The triangles indicating different ports are different shapes.



Figure 2. The mode button’s shape is slightly different. The square next to the management port is greenish on the counterfeit switch. On the genuine device, it is bright yellow.

While immediately recognizing such minute differences may be challenging to spot, side-by-side comparison clearly shows that the enclosures of counterfeit units are of a lesser quality.

2.3 Board analysis

The main component of any modern electronics is printed circuit boards (PCBs) carrying electronic components such as integrated circuits, connectors, and passive components. By analyzing these boards, we could spot any differences and similarities between devices of the same family in the hope of gaining insight into what modifications were done by the counterfeiters.

The overall board layout of the three devices was similar, with the Genuine unit and Counterfeit B sharing more similarities in appearance. When observed in detail, however, it was possible to verify modifications for forgery purposes, and the differences are significant.

The absence of a holographic sticker on the counterfeit units was immediately noticeable. While its presence on the Genuine unit was not a guarantee of authenticity, its absence typically indicated a counterfeit.



Figure 3. Legitimate holographic sticker which was absent on both counterfeit units.



Figure 4. Genuine unit, internal view.



Figure 5. Counterfeit A, internal view.



Figure 6. Counterfeit B, internal view.

The Flash part numbers were found to be different, albeit both identifying 1Gbit parallel NOR Flash devices. The Genuine unit had Spansion p/n S29GL01GS11TFIV1 installed, while the counterfeit devices were installed with Micron/Intel p/n JS28F00AM29EWH. This could be attributed to the fact that the Genuine unit was identified as V5 while the counterfeits were V1. It is hard to say without comparing devices of the same version whether this was an indicator. It could also be the result of the manufacturer swapping in cheaper parts.

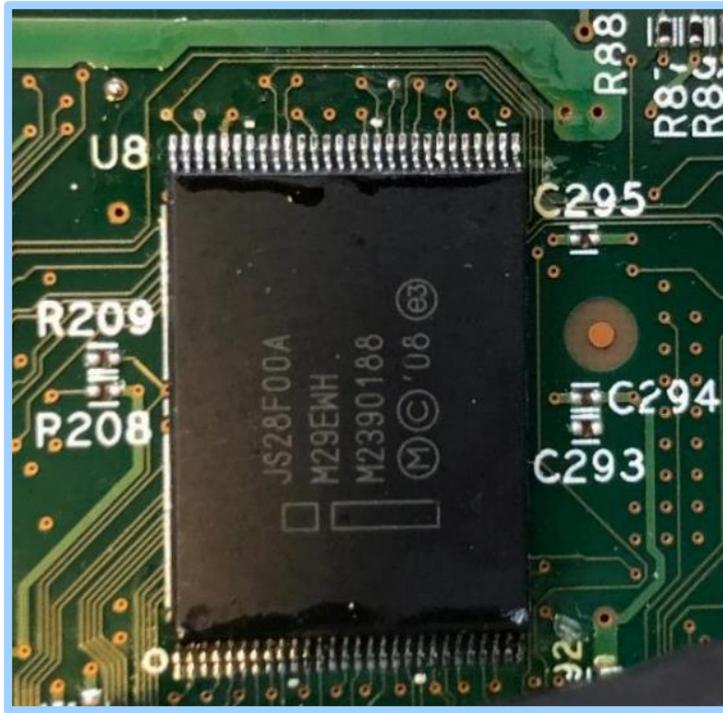


Figure 7. U8, an 1Gbit NOR Flash. PCB rework evidence on the Counterfeit A unit: soldering flux residue on and around the Flash IC is present.

What was more concerning, was the presence of PCB rework traces around the Flash IC on the Counterfeit A unit. While it could be the case the unit was legitimately repaired, no record of such activity was found via available sources. This led us to conclude that, at some point, the Flash chip had been replaced. This may have been part of legitimate repair activities; however, it is not typical for legitimate repair shops to leave flux residues on the board.

2.3.2 COUNTERFEIT B

The Counterfeit B unit was found to have one significant difference when compared to other units: the presence of components with their top marking removed with a laser; the components are U55 and U10006.

Whereas the same U55 component is a serial EEPROM in the TSSOP-8 package in other units, the Counterfeit B unit sported a QFN16 package. This could be a legitimate engineering change when producing a new board revision, but it is unusual for an I²C EEPROM to be manufactured in such a package. Furthermore, there is no realistic reason to remove top marking for such a simple part.

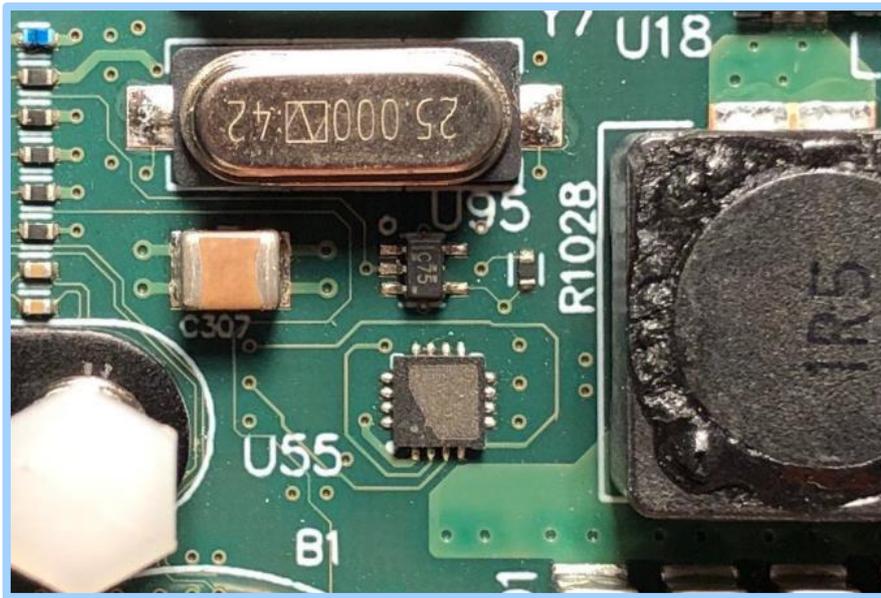


Figure 10. U55, an unknown component with top marking removed.

The other component with its top marking removed was U10006, bearing a nondescript top marking even in a genuine unit: 1341604/QQ2Q8/B1837. Unfortunately, this made determining the exact function of this component very challenging.

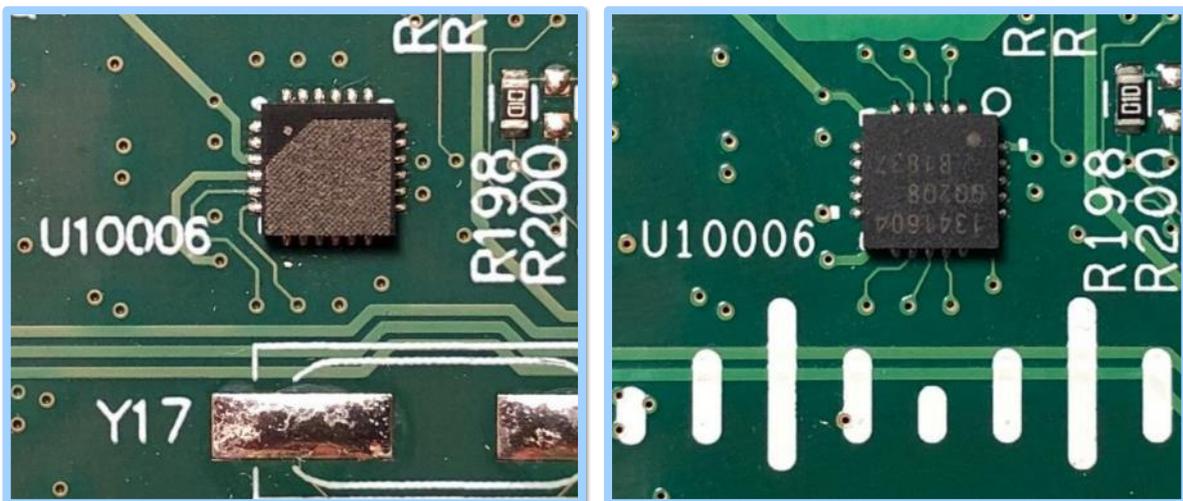


Figure 11. U10006, another component with top marking removed in Counterfeit B. The Genuine unit to the right for comparison. Note silkscreen quality was lacking on the counterfeit board.

Note the many similarities in via positioning; they matched almost exactly and followed almost the same routing for traces connecting to the passive components on the right. However, U10006 was not only rotated, it also appeared to have a different pinout compared to the genuine board. Pin 1 marking was also very different both on silkscreen and the component itself.

2.4 Boot log acquisition and analysis

Any sufficiently sophisticated device needs to have a way to provide an insight for manufacturing engineers and end users into its state as well as enable controlling its functioning. Such a way is usually implemented through some sort of a console, allowing the operator to observe system messages as the device boots and to input commands to control the boot process and device operation in general. Naturally, these messages provide a wealth of useful information when investigating device workings and as such are important to capture.

For example, this allowed identification of the software version installed on each device, and was instrumental in obtaining clean images from the vendor for further comparison.

Inventory information was also displayed during the boot, mainly consisting of serial numbers for various parts comprising the device. This allowed us to verify the numbers against labels present on each part. While not a direct indication of being counterfeit, a mismatch was indicative of the part being replaced.

Probably the most interesting aspect was the analysis of the inoperable device, which failed the platform authentication procedure. From the very start of the boot:

```
CPU rev: B
Image passed digital signature verification
Board rev: 18
Testing DataBus...
Testing AddressBus...
Testing Memory from 0x00000000 to 0x1fffffff.../
Using driver version 4 for media type 1
...
```

It is worth noting that the device reported (some) digital signature verification passed, even though the boot process resulted in a non-functional device. Similarly:

```
...
...done Initializing Flash.
Loading "flash:c2960x-universalk9-mz.152-4.E7.bin"...Verifying image flash:c2960x-
universalk9-mz.152-
4.E7.bin.....
.....
.....
.....Image passed digital signature verification
@@@...@@@
File "flash:c2960x-universalk9-mz.152-4.E7.bin" uncompressed and installed, entry point:
0x3000
executing...
```

Note the two-pass loading process, with verification being performed separately.

However:

```
...done Initializing flashfs.
Checking for Bootloader upgrade..
Boot Loader upgrade not needed(v)

FIPS: Flash Key Check : Begin
FIPS: Flash Key Check : End, Not Found, FIPS Mode Not Enabled

POST: MA BIST : Begin
POST: MA BIST : End, Status Passed

POST: TCAM BIST : Begin
POST: TCAM BIST : End, Status Passed

POST: ACT2 Authentication : Begin
POST: ACT2 Authentication : End, Status Failed
extracting front_end/front_end_ucode_info (43 bytes)
Waiting for Stack Master Election...
POST: Thermal, Fan Tests : Begin
POST: Thermal, Fan Tests : End, Status Passed
...
```

To summarize, the platform consisting of bootloaders, together with any potential pre-boot mechanisms successfully authenticated the application image. However, the application then failed to authenticate the platform. This seemed to correlate well with the high-level symptoms outlined before, which may have either meant that the software image was patched covertly when loaded, or the patched version was already provisioned; this would be easy to verify once images had been extracted from the devices.

2.5 Content extraction from live systems

The Cisco devices implement advanced management capabilities through the serial console, allowing not only to change the configuration settings but also explore the available file systems. This was leveraged as a non-invasive method of extracting contents – we did not know in advance whether raw filesystem data extracted from Flash ICs could actually be used with non-Cisco systems, for example, mounted in Linux.

This stage followed directly after boot log acquisition, and leveraged the console access to explore, and tried to discover and back-up any interesting files present on the local filesystems. Unless a standard filesystem is used, extracting files from a raw Flash image is usually significantly harder than copying those off the device when it is powered on.

The devices store the application software image in a file located in the main `flash:` file system. Therefore, it was relatively easy to obtain a copy of the software being executed on counterfeit devices. The fact that the units stopped working after a software update, yet still reported the updated version during the boot, can be considered evidence that no hidden software copy was present elsewhere on the system.

2.6 Direct Flash content extraction

As already noted, the boards sported a prominent parallel NOR Flash chip of considerable size. This made the chip the prime candidate for storing at least the application part of the overall software package, so was the first one to be checked for suspicious traces.

All three devices underwent the same extraction procedure. First, the chips were removed from the boards. Then, Flash content extraction was performed with the Elnec BeeProg2 with 70-3170 TSOP56 adapter, using the Elnec software PG4UW. Content inspection showed the chips were written with bytes swapped with 16-bit words, thus requiring a quick adjustment to accommodate for that.

```

3E 65 00 00 4E 0A 20 6F 170 73 63 61 20 65 6E 69 | >e N opsca eni
6F 66 6D 72 74 61 6F 69 120 6E 76 61 69 61 61 6C | ofmrtaoi nvaiaal
6C 62 0A 65 00 00 00 00 125 0A 20 64 79 62 65 74 | lb e % dybet
20 73 76 61 69 61 61 6C 16C 62 20 65 25 28 20 64 | svaiaallb e%(< d
79 62 65 74 20 73 73 75 164 65 0A 29 00 00 00 | ybet ssude)
62 2D 00 00 6E 49 61 76 169 6C 20 64 75 62 66 66 | b- nlavil dubff
72 65 73 20 7A 69 20 65 170 73 63 65 66 69 65 69 | res zi epscefiei
0A 64 00 00 6E 55 62 61 165 6C 74 20 20 6F 6C 61 | d nUbaelt ola
6F 6C 61 63 65 74 25 20 120 64 79 62 65 74 20 73 | olacet% dybet s
6F 66 20 72 75 62 66 66 172 65 00 0A 6F 4E 20 74 | of rubffre oN t
6E 65 75 6F 68 67 73 20 161 70 65 63 6F 20 20 6E | neuohgs apeco n
65 64 69 76 65 63 00 0A 100 2E 00 00 69 46 65 6C | edivec . iFel
22 20 73 25 20 22 75 73 163 63 73 65 66 73 6C 75 | " s% "usccefsllu
79 6C 63 20 70 6F 65 69 120 64 6F 74 22 20 73 25 | ylc poei dot" s%
0A 22 00 00 6C 66 73 61 120 68 72 70 62 6F 20 65 | " lfsa hrpbo e
75 73 63 63 73 65 66 73 16C 75 00 0A 61 46 6C 69 | usccsefsllu aFli
74 20 20 6F 6C 61 6F 6C 161 63 65 74 6D 20 6D 65 | t olaolacetm me
72 6F 20 79 6F 74 63 20 165 72 74 61 20 65 73 25 | ro yotc erta es%
00 0A 00 00 72 63 61 65 165 74 69 20 73 66 25 20 | rcaeti sf%
0A 73 00 00 61 46 6C 69 174 20 20 6F 72 63 61 65 | s aFlit orcae
65 74 25 20 20 73 69 66 165 6C 79 73 74 73 6D 65 | et% sifelystsme

```

Figure 12. Evident byte swapping in the extracted content. The text was intelligible but required a certain mental strain to understand it.



Figure 13. TSOP56 adapter used. (Image taken from the official Elnec web site)

Three images were obtained, each 128MB in size. After content extraction, the Flash chips were installed back on the boards to return them to an operable state.

2.7 Flash content analysis

The main purpose behind this step was to gather intelligence on how data was stored on the physical medium, and whether there was anything not accounted for during the live system analysis step performed before, such as bootloader code, any signatures, etc.

A quick content inspection using the entropy graphing feature of the `binwalk` tool showed us several areas of interest.

- The high-entropy area of about 20M bytes was probably the main software image, fitting the size and the entropy level as compressed data has high entropy
- What followed were two areas of distinct entropy footprints, likely some sort of uncompressed data
- The zero-entropy area is where the same 0xFF value was written and can be considered empty
- At the very end, there were several small but distinct blocks of data

After manual inspection at the very beginning of the image and direct binary comparison, we concluded that no bootloader code of any kind was placed there; the data looked more like a file system. Supporting that assumption was the difference in the composition of the blocks on the devices. Therefore, it made more sense to assume bootloader code was located at the end of the Flash; this was supported by the similar-looking entropy graph for all images towards the end.

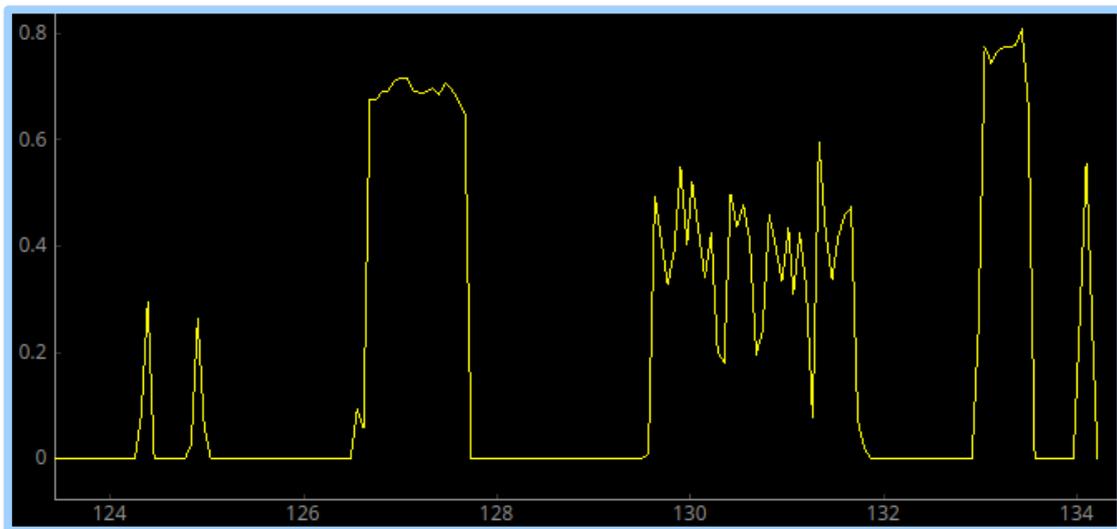


Figure 14. Zoomed view showing the very end of the image. The picture was very similar for all three Flash images.

Indeed, inspecting the last megabyte revealed the presence of what appeared to be two bootloader programs, easily identified with the help of embedded strings (examples from Counterfeit A):

```
C2960X Boot Loader (C2960X-BROM) Version 15.2(2r)E1, RELEASE SOFTWARE (fc1)
Compiled Wed 23-Apr-14 02:21 by abhakat
```

and

```
C2960X Boot Loader (C2960X-HBOOT-M) Version 15.2(2r)E1, RELEASE SOFTWARE (fc1)
Compiled Wed 23-Apr-14 02:21 by abhakat
```

These boot loaders corresponded to the last two peaks on the entropy graph above.

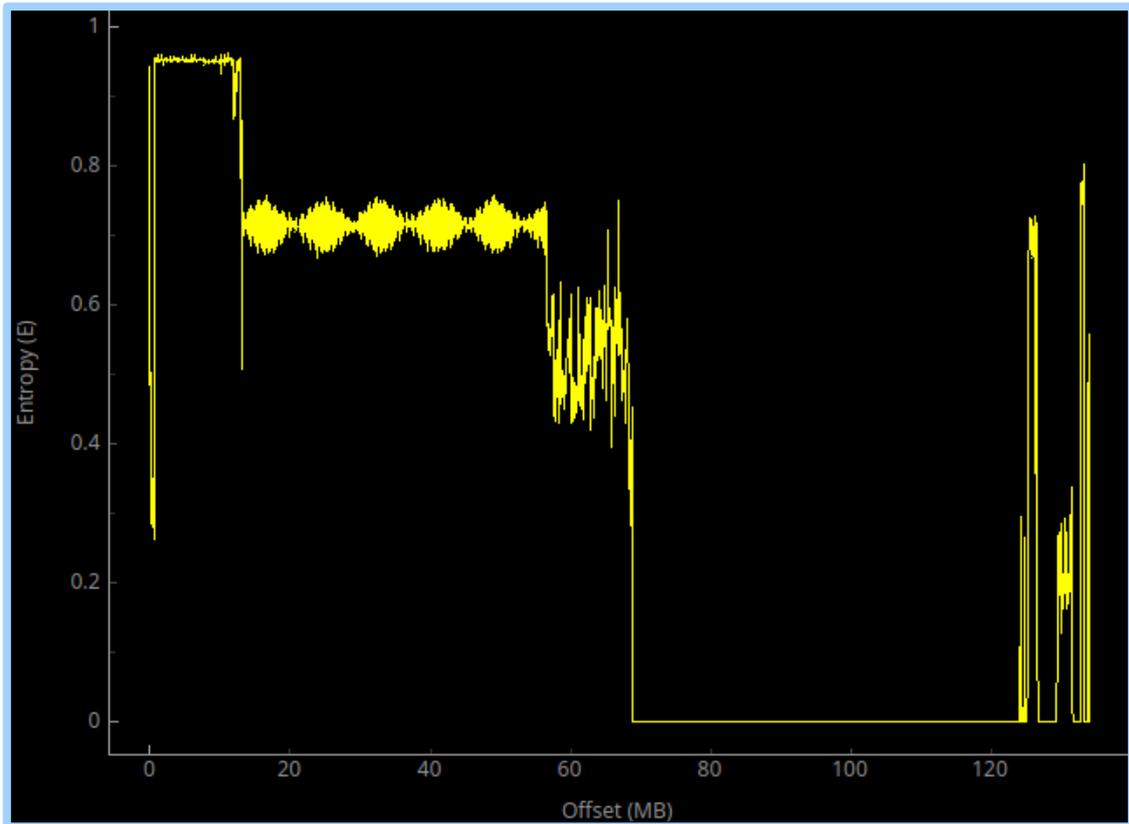


Figure 15. Entropy graph of the known-good Flash image obtained from the Genuine unit.

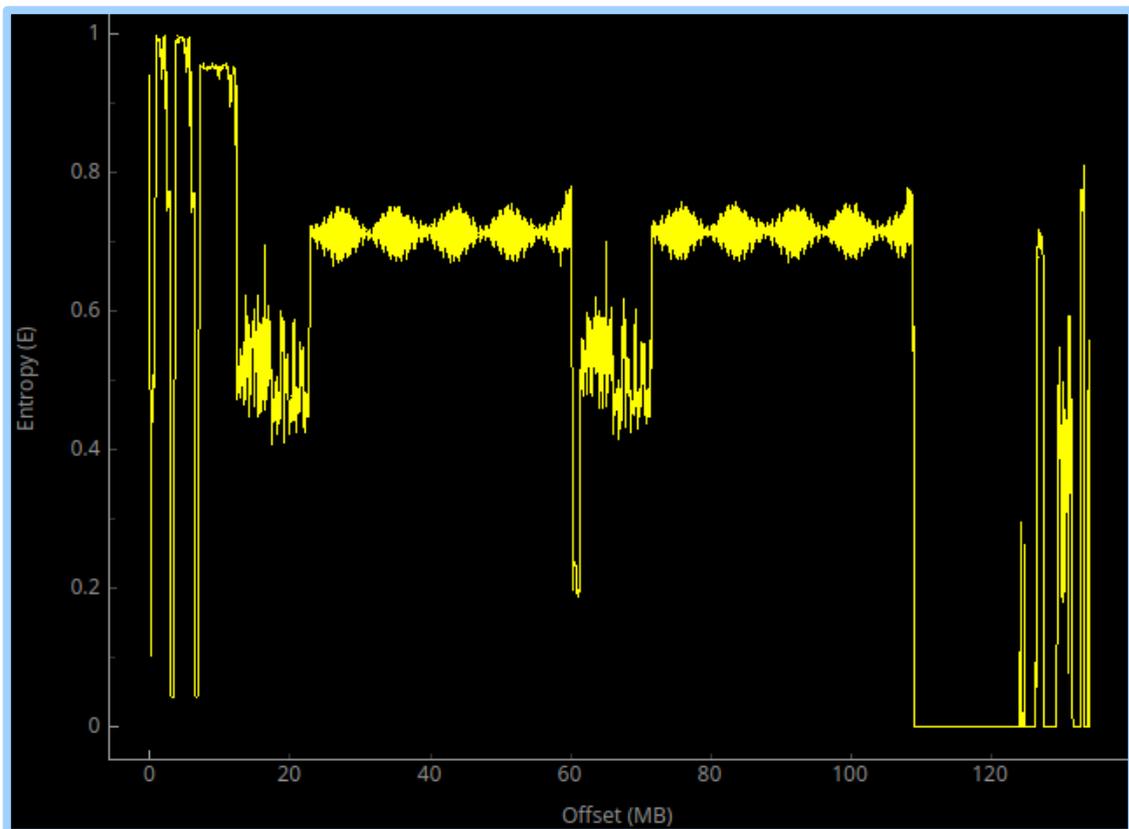


Figure 16. Entropy graph of the suspect Flash image obtained from the Counterfeit A unit.

The next series of peaks of around 130 to 132MB represent data that was hard to attribute to anything that was known, but the best guess was that it was related to the Flash file systems. However, one interesting piece of information at the very end stood out; it was marked with the magic string `AMCC` and contained what appeared to be file names such as `ppc.bin.key` and `pka_fw.bin`. The analysis of this file format is summarised in appendix 6.3.

The peak around 127MB belonged to the microcode binary named `c2960x_d1pd_porter.bin`. This file was located on the `ucode0` filesystem.

Finally, two small peaks represented the `lic0` and `lic1` filesystems.

The following table summarizes the investigation of the last few megabytes of the Flash contents of the Counterfeit A unit.

Offset	Contents
0x76A0000	The lic1 filesystem
0x77720000	The lic0 filesystem
0x78C0000	The ucode0 filesystem
0x7DA0800	Block of data marked AMCC
0x7DC0000	Board configuration (text based)
0x7EE0000	C2960X Boot Loader (C2960X-HBOOT-M)
0x7FDD800	Inventory data
0x7FDDC00	Signature 1
0x7FDFFFC	4 bytes, 55 AA AA 55
0x7FE0000	C2960X Boot Loader (C2960X-BROM)
0x7FFDC00	Signature 2
0x7FFF000	Unknown; appears to be PowerPC code but no text strings to identify it
0x7FFFFFC	4 bytes, 4B FF F0 02 (decodes as the ba 0xffff000 PowerPC instruction)

It can also be assumed the main `flash` filesystem started right at offset zero. While offsets of the first three entries differed on other units, the overall composition was expected to be the same.

With some general understanding of what was where in the Flash, it was then possible to perform a meaningful comparison of Flash contents between the genuine unit and the counterfeit one, which was not upgraded.

Looking at the very end of the file, some differences stood out.

Both counterfeit units had offsets `0x7FFF800..0x7FFFFFC` and `0x7FFE000..0x7FFF000` filled with apparently random noise. Given that Flash chips in the erased state had all bits set, it was unlikely this was uninitialized data.

7FFFF08: 0E 22 39 2C 21 87 71 91 ."9, !+q`	7FFFF08: FF FF FF FF FF FF FF FF YYYYYYYY
7FFFF10: 05 DF 9F A5 37 40 D5 3B .BŸW7eÖ;	7FFFF10: FF FF FF FF FF FF FF FF YYYYYYYY
7FFFF18: 0B 43 95 E1 A8 FD F0 70 .C•á"ý&p	7FFFF18: FF FF FF FF FF FF FF FF YYYYYYYY
7FFFF20: 54 31 F0 E1 F2 C3 34 62 Tl8ábÄ4b	7FFFF20: FF FF FF FF FF FF FF FF YYYYYYYY
7FFFF28: C8 70 DA A9 86 BD A2 89 ÈpÚ@+*ctz	7FFFF28: FF FF FF FF FF FF FF FF YYYYYYYY
7FFFF30: 84 2D 4F C3 4C B7 51 1E ,,-OÄL·Q.	7FFFF30: FF FF FF FF FF FF FF FF YYYYYYYY
7FFFF38: 58 77 A2 BA 1D A4 EE A1 Xwc°.mí;	7FFFF38: FF FF FF FF FF FF FF FF YYYYYYYY
7FFFF40: 48 C2 F6 9F 47 19 36 54 HÂöŸG.6T	7FFFF40: FF FF FF FF FF FF FF FF YYYYYYYY
7FFFF48: 0B 65 C5 87 0C CF 7E BD .eÄ+.î~+;	7FFFF48: FF FF FF FF FF FF FF FF YYYYYYYY
7FFFF50: 8E 1C 5D 0A 23 25 2B 28 Ž.]#*+(7FFFF50: FF FF FF FF FF FF FF FF YYYYYYYY
7FFFF58: 72 87 5E C6 36 9B 3B 22 r+^E6>,"	7FFFF58: FF FF FF FF FF FF FF FF YYYYYYYY
7FFFF60: 8A A9 3D DA 63 58 BA 00 Še=ÚcX"	7FFFF60: FF FF FF FF FF FF FF FF YYYYYYYY
7FFFF68: 5E 69 C4 6C BC A5 4E 56 ^iÄl~wNV	7FFFF68: FF FF FF FF FF FF FF FF YYYYYYYY
7FFFF70: AA 14 8F 24 C6 70 AC 80 .·Ep-C	7FFFF70: FF FF FF FF FF FF FF FF YYYYYYYY
7FFFF78: E0 D9 91 AE FD CA 1F 1C àù'@ýÊ..	7FFFF78: FF FF FF FF FF FF FF FF YYYYYYYY
7FFFF80: A2 4C 8D 3B 4F 6A 08 8E cL;Oj.Ž	7FFFF80: FF FF FF FF FF FF FF FF YYYYYYYY
7FFFF88: 49 E4 9D 01 9C 28 59 7B Iä.œ(Y{	7FFFF88: FF FF FF FF FF FF FF FF YYYYYYYY
7FFFF90: 61 7D AF B6 3D 82 19 4F a}~Ÿ=,·O	7FFFF90: FF FF FF FF FF FF FF FF YYYYYYYY
7FFFF98: 2A D7 02 19 7A 1A E4 B7 *x...z.ä.	7FFFF98: FF FF FF FF FF FF FF FF YYYYYYYY
7FFFFA0: 19 15 AE 6A 12 35 68 27 ..@j.5h'	7FFFFA0: FF FF FF FF FF FF FF FF YYYYYYYY
7FFFFA8: 54 3E 19 ED B7 3E E9 55 T>.í>éU	7FFFFA8: FF FF FF FF FF FF FF FF YYYYYYYY
7FFFFB0: 3A C0 83 6B 8E 41 BD BC :ÄfkŽA*+;	7FFFFB0: FF FF FF FF FF FF FF FF YYYYYYYY
7FFFFB8: D9 E8 7B B0 B4 71 D0 1A Ùè{°`qD.	7FFFFB8: FF FF FF FF FF FF FF FF YYYYYYYY
7FFFFC0: 75 6B 67 0E B4 A3 22 F3 ukg.°ž"ó	7FFFFC0: FF FF FF FF FF FF FF FF YYYYYYYY
7FFFFC8: 08 E0 00 D8 12 D0 44 0C .à.Ø.BD.	7FFFFC8: FF FF FF FF FF FF FF FF YYYYYYYY
7FFFFD0: BD 43 D4 E8 C3 97 DF F0 +CÔèÄ-Bß	7FFFFD0: FF FF FF FF FF FF FF FF YYYYYYYY
7FFFFD8: 74 73 C2 1B B3 B9 2D 6E tsÄ.~¹-n	7FFFFD8: FF FF FF FF FF FF FF FF YYYYYYYY
7FFFFE0: 41 B5 81 CF 3E 9B 7D 19 ApÏ>}.)	7FFFFE0: FF FF FF FF FF FF FF FF YYYYYYYY
7FFFFE8: EC 30 1A 6C B7 C7 B3 C7 i0.1·Ç³Ç	7FFFFE8: FF FF FF FF FF FF FF FF YYYYYYYY
7FFFFF0: A0 81 FF 2E 18 B8 5B 1D ý.., [.	7FFFFF0: FF FF FF FF FF FF FF FF YYYYYYYY
7FFFFF8: 4B 14 B2 19 4B FF F0 02 K.°.Kyø.	7FFFFF8: FF FF FF FF 4B FF F0 02 YYYYKYø.
8000000: FF FF FF FF FF FF FF FF YYYYYYYY	8000000: FF FF FF FF FF FF FF FF YYYYYYYY

Figure 17. Differences at the very end between the Counterfeit A and Genuine units. While the last four bytes were the same, the counterfeit unit had some extra bits and pieces, unlike the erased genuine bits.

Unfortunately, further comparison was thwarted by the difference in versions of installed software and bootloaders. It was necessary to locate intact copies of the bootloader code in order to perform any kind of meaningful differential analysis.

2.8 Bootloader analysis

Further analysis required obtaining software images of corresponding versions from the official sources. However, first the software versions needed to be identified. This could be done via the file name of the installed application image. Furthermore, the following bootloader image versions were present on the counterfeit units at hand:

- Counterfeit A (operational):

```
C2960X Boot Loader (C2960X-HBOOT-M) Version 15.2(2r)E1, RELEASE SOFTWARE (fc1)
Compiled Wed 23-Apr-14 02:21 by abhakat
```

```
C2960X Boot Loader (C2960X-BROM) Version 15.2(2r)E1, RELEASE SOFTWARE (fc1)
Compiled Wed 23-Apr-14 02:21 by abhakat
```

- Counterfeit B (inoperable):

```
C2960X Boot Loader (C2960X-HBOOT-M) Version 15.2(4r)E3, RELEASE SOFTWARE (fc4)
Compiled Wed 04-Apr-18 10:35 by smaddasa
```

```
C2960X Boot Loader (C2960X-BROM) Version 15.2(2r)E2, RELEASE SOFTWARE (fc1)
Compiled Fri 05-Dec-14 01:35 by abhakat
```

The version strings appeared unique enough to allow a search for them within the uncompressed data sections of the images at hand; see appendix 6.2 regarding the file format of the software image. In fact, they were included as one monolithic image, likely directly copied to Flash in parts, starting from offset 0x7EE0000, taking care to preserve inventory data. Bootloader images extracted from genuine software updates could then be directly compared with their counterparts obtained from Flash content extracted directly from the units.

The `c2960x-universalk9-mz.150-2.EX5.bin` software image obtained from Counterfeit A did not contain embedded bootloaders, so a search was conducted by grabbing multiple official images and matching version numbers found in there. The `c2960x-universalk9-mz.152-2.E.bin` image provided the correct versions.

Checking the bootloader images obtained from Counterfeit A:

- Two changes were made to HBOOT: the first one to 8 bytes inside the code section, and the second which was considerably larger, appearing appended to the HBOOT image
- Signature 1 data was intact
- 12 bytes modified right before the BROM code
- BROM code was intact
- Signature 2 data was different
- Extra data is present around the code block at 0x7FFF000, as highlighted before

At this point, the process in place for verification of each software piece was not known. However, it was evident that some tampering with HBOOT had taken place. To understand these modifications and their significance within the context of the system, it is important to consider what was known about the System-on-Chip and the way it booted up. The collected reference data is summarized in appendix 6.1.

Assuming the PowerPC core started executing at 0xFFFFF0FC, the Flash ROM could be safely placed at the upper addresses so that the last 4 bytes in Flash, where we assumed the unconditional branch instruction was, map starting at 0xFFFFF0FC.

After loading the last 1 megabyte or so of Flash into IDA Pro in the manner described above, we could start exploring the code. The assumption made above regarding Flash mapping is validated by the fact that IDA Pro could explore the code and create references automatically.

The branch instruction at 0xFFFFF0FC transferred control to the small blob of PowerPC instructions at 0xFFFFF000. This piece of code appeared to perform some basic initialization activities and passed control further to BROM.

Judging by the contents of text strings included, BROM appeared to authenticate and start HBOOT and provided a rudimentary set of commands to rescue the system from the state where HBOOT could be started. While we were not as interested in the details of the command system, the image authentication functionality was important to understand.

This functionality could be located easily by finding references to the tell-tale string `Image passed digital signature verification` that was being printed on the serial console when image authentication succeeded. The function referencing this string together with the companion failure message took 6 parameters, most important of which were the base address of HBOOT and the signature 1 address. This meant signature 1 was in fact the HBOOT signature.

By repeating the search and analysis on the HBOOT image, a similar function was identified that was used for the same purpose of image authentication. This function was used to authenticate the application software image; the process also showed up in console output. However, another use of the same function was made to authenticate BROM code when it was being copied over to the Flash memory. Careful inspection of the parameters being passed revealed that the BROM signature was located 0x2400 bytes before the image end, which was exactly where signature 2 was located. We could therefore conclude that signature 2 authenticated BROM code.

Further inspection of the signature verification implementation in both BROM and HBOOT showed the use of some functions related to "SlimPro", as evident by corresponding error messages referenced by those functions:

```
WR: Timeout waiting for SlimPro response
RD: Timeout waiting for SLimPro msg
RD: Timeout waiting for SLimPro response
```

The answer to the question of what this component might be came from various materials published by the vendor. The SLIMpro was a separate computing unit integrated into the System-on-Chip and was responsible for system-related security operations. The strings presented above confirmed this information. Further information on the SoC can be found in appendix 6.1.

With a clear overview of what was verified, when, and how, a prominent question arose: how did BROM report successful verification of the modified HBOOT code? Answering this required a review of what was already known, and a deeper dive into the modifications performed on the hardware of the Counterfeit A unit. But first, the changes done to HBOOT and their purpose were investigated.

2.9 HBOOT patch analysis

The analysis required a good understanding of what HBOOT did and how. We began by observing that like BROM, HBOOT implemented a console with an impressive set of commands. Finding out how these commands were added allowed us to spot every implemented command, and rename the corresponding handler functions. This provided at least some insight into what parts were patched.

cmd_arp	FFF10E64
cmd_boot	FFF05AD0
cmd_cat	FFF0672C
cmd_copy	FFF06E78
cmd_delete	FFF065E0
cmd_dir	FFF068B8
cmd_flash_init	FFF059F4
cmd_format	FFF06118
cmd_fsck	FFF05ED8
cmd_help	FFF03944
cmd_memory	FFEE648C
cmd_mgmt_clr	FFEE9534
cmd_mgmt_init	FFEEB29C
cmd_mgmt_show	FFEE9BF4
cmd_mkdir	FFF063A8
cmd_ping	FFF10A70
cmd_rename	FFF06DF0
cmd_reset	FFF03784
cmd_rmdir	FFF0625C
cmd_set	FFEF1C20
cmd_set_bs	FFF058FC
cmd_set_param	FFEF1198
cmd_sleep	FFF036AC
cmd_unset	FFEF1B30
cmd_version	FFF0372C

Figure 18. List of console commands supported by HBOOT.

Starting with the first modification of two PowerPC instructions in the middle of the HBOOT code section, we saw the `boot` command implementation was modified to include a call to the other added code fragment. The call was patched in to be performed after the application image was loaded into memory and authenticated.

Analysis of the inserted code revealed this to be the first stage of a de-obfuscator (XOR-based with the key derived from the unit serial number), processing the "random" data previously discovered at the end of Flash. This data is de-obfuscated into a stack-based buffer, and control was then passed there. Care was taken to verify the operation was performed correctly, so the unit did not crash even when there was no obfuscated data present.

That code was found to be a stage 2 de-obfuscator, with the key based on certain data from the Flash IC not accessible directly through conventional tools we had. Due to these circumstances, the key had to be brute forced. Similarly, the de-obfuscated code of stage 3 was again placed into a stack-based buffer and executed.

Stage 3 was found to be the actual patching code, searching for the `serialNu` string and applying some modifications to the IOS image expected to be already loaded in memory. The modifications were few and consisted of mainly "return OK" kind of patches; full details of functions being patched will not be published for obvious reasons. However, it appeared the only purpose of this "added functionality" was to circumvent software licensing protections.

It was due to this added patching functionality that the counterfeit units could bypass platform authenticity verification. This also explained why units stopped working after a software update: the latest software will almost certainly rewrite the patched HBOOT code, removing the work done to bypass the checks. The case of the Counterfeit B unit confirmed this hypothesis.

When the CISO of the victim company provided us the counterfeit devices for investigation one of the main tasks was to answer whether there were any backdoor-like functionalities being introduced. We concluded this did not appear to be the case for application and HBOOT code.

2.10 SLIMpro analysis

We had reached the point where we were ready to investigate the question posed previously with regards to the bypassing of HBOOT image verification. As the SLIMpro component was responsible for authentication because PowerPC cores only initiated the process by posting a message to a "mailbox", it was reasonable to conclude that some changes were implemented, resulting in this component always reporting a successful authentication. By identifying and understanding these changes we would be able to explain how the counterfeiters are able to bypass the code authentication function.

What piqued our interest in the case of Counterfeit A was the "implant" added in conjunction with a serial EEPROM chip. What was the reason for such an unusual and obvious addition? Furthermore, the same chip had been replaced with a completely different package in Counterfeit B.



Figure 19. The implant PCB, disconnected from the unit, with the resin blob removed. As with Counterfeit B, the top marking was erased.

Since the protocol used to interface with EEPROM was relatively simple and slow, it was easy to intercept and record the communications to gain insight into the workings of this implant. Any existing tool able to decode I²C communications and export the decoded traffic could be used for this purpose. Below is a short excerpt of this intercepted traffic as produced by the Logic software shipped with Saleae logic analyzer:

```
Time [s],Packet ID,Address,Data,Read/Write,ACK/NAK
3.256962416666667,0,'164' (0xA4),'0' (0x00),Write,ACK
3.257046583333334,0,'164' (0xA4),'0' (0x00),Write,ACK
3.257230166666667,1,'164' (0xA4),'240' (0xF0),Read,ACK
3.257315166666667,1,'164' (0xA4),'240' (0xF0),Read,ACK
3.257400333333333,1,'164' (0xA4),'3' (0x03),Read,ACK
3.257485416666666,1,'164' (0xA4),'18' (0x12),Read,ACK
3.257570500000000,1,'164' (0xA4),'0' (0x00),Read,ACK
3.257655583333333,1,'164' (0xA4),'252' (0xFC),Read,ACK
...
```

According to the M24512 data sheet, random-access read operation was performed by writing two address bytes followed by reading multiple bytes of data. This corresponded to the observed traffic. A simple script was

written to parse such output and to create a dump file containing the intercepted contents, as well as to provide some overview on the read transactions executed.

```
Below is the output of the script showing what addresses were accessed and how many bytes read (in hexadecimal), which is much easier to analyze compared to the raw transactions. It has been abbreviated due to its size.
Start address: 0000, byte count: 0020
Start address: 021E, byte count: 0004
Start address: 0020, byte count: 0040
Start address: 0060, byte count: 0040
Start address: 0060, byte count: 0040
Start address: 00A0, byte count: 0040
Start address: 00E0, byte count: 0040
Start address: 0120, byte count: 0040
Start address: 0160, byte count: 0040
Start address: 6320, byte count: 0400
[SEQUENTIAL ACCESS PATTERN CONTINUES]
Start address: 7320, byte count: 0400
Start address: 7720, byte count: 0364
Start address: 0060, byte count: 0040
Start address: 0060, byte count: 0040
Start address: 00A0, byte count: 0040
Start address: 00E0, byte count: 0040
Start address: 0060, byte count: 0040
Start address: 00A0, byte count: 0040
Start address: 00E0, byte count: 0040
Start address: 0120, byte count: 0040
Start address: 61A0, byte count: 0100
Start address: 6020, byte count: 0100
Start address: 0060, byte count: 0040
Start address: 00A0, byte count: 0040
Start address: 00E0, byte count: 0040
Start address: 6020, byte count: 0100
Start address: 0060, byte count: 0040
Start address: 00A0, byte count: 0040
Start address: 5EA0, byte count: 0100
Start address: 0220, byte count: 0200
[SEQUENTIAL ACCESS PATTERN CONTINUES]
Start address: 5E20, byte count: 0050
Start address: 0220, byte count: 0400
[SEQUENTIAL ACCESS PATTERN CONTINUES]
Start address: 5E20, byte count: 0050
```

Exactly the same behavior was observed when traffic was captured on the Genuine unit.

The following could be immediately noted:

- Repeating small accesses to addresses 0x60 through 0xE0 of 0x40 bytes each, followed by a significant sequential read.
- Data starting at address 0x220 was read twice, but with different transaction sizes.

Here is a closer look at what was being read:

0000h:	F0F00312	00FC0100	F4100081	85082554	21160001	000126E0	0408002E	E0050800	88...ü..ô.....%T!.....&à....à...
0020h:	414D4343	80000000	05000000	00000000	00000000	00000000	00000000	00000000	AMCCÉ.....
0040h:	00000000	00000000	00000000	00000000	00000000	00000000	00000000	4A1C99D9J.™Ü
0060h:	6970702E	62696E00	73626F6F	745F6F00	00020000	505C0000	00000000	00000000	ipp.bin.sboot_o.....P\.....
0080h:	00000000	00000000	00000000	00000000	00000000	00000000	00000000	0393D21F"ò.
00A0h:	6970702E	62696E2E	73696700	506F7200	805E0000	00010000	00000000	00000000	ipp.bin.sig.Por.€^.....
00C0h:	00000000	00000000	00000000	00000000	00000000	00000000	00000000	D90DAABAÛ.°
00E0h:	6970702E	62696E2E	6B657900	506F7200	00600000	00010000	00000000	00000000	ipp.bin.key.Por..`.....
0100h:	00000000	00000000	00000000	00000000	00000000	00000000	00000000	4435B69CD59œ
0120h:	6970702E	62696E2E	6B736700	2E2E2F00	80610000	00010000	00000000	00000000	ipp.bin.ksg.../.€a.....
0140h:	00000000	00000000	00000000	00000000	00000000	00000000	00000000	3345098D3E..
0160h:	706B615F	66772E62	696E0049	50505F00	00630000	64170000	00000000	00000000	pka_fw.bin.IPP...c.d.....
0180h:	00000000	00000000	00000000	00000000	00000000	00000000	00000000	F7193D4E÷.=N

Figure 20. The beginning of the obtained EEPROM/implant dump of the Counterfeit A unit.

After a 32-byte block of unknown data, we could see the AMCC magic bytes, identifying the container format described in appendix 6.3. By correlating the accesses with file offsets within the container, it was easy to identify which files from this container were being read and in what order, so a higher-level overview could be pieced together:

- pka_fw.bin
- ipp.bin.ksg
- ipp.bin.key
- ipp.bin.sig
- ipp.bin
- ipp.bin (again)

The fact that the ipp.bin file was being read twice stands out. This mirrored the situation with the main application binary being read twice; first to verify the signature, then to decompress and pass control to. Therefore, it is easy to assume the similar situation here as well: verify, then execute.

Such an implementation, however, is vulnerable to a classic race condition called time-of-check to time-of-use (TOCTOU) where verified content could be manipulated after it had been verified but before its use. This immediately prompted a comparison between the two reads (it was easy to truncate the I²C traffic dump to exclude the second series of transactions starting at address 0x220).

Quite similarly to the HBOOT patch, two differences were detected: one small patch within the bulk of the file and another, larger binary blob appended to the image.

Assuming this file contained software for some CPU architecture and not data, we could attempt to identify which architecture this was meant to be executed on. Unfortunately, binwalk -A did not produce any meaningful output. However, the following facts were observed:

- The beginning of the file appeared to contain a set of 32-bit little-endian integers similar in magnitude except the very first one
- The byte sequence 70 47 was encountered quite often in the file

These two facts point at the possibility of this being ARM Thumb code, with the exception vectors located at the beginning as is common with ARM-based embedded systems. From there, it was easy to verify the assumption and guess the correct loading address with IDA Pro.

It was also possible to locate the previously observed BOOT FAIL string in this file.

With that, we concluded that the `ipp.bin` file contained software running on the SLIMpro SoC component.

After loading the image in IDA Pro and spending some time marking up known library functions, we turned our attention to the changes made to the image by the implant. It transpired that a call to `memcmp()` was replaced in a certain function with a call to another function introduced by the patch; the new function inherited the original semantics. Below is the pseudocode of the replacement function:

```
int __fastcall ADDED_sub_xxxx(const void *a1, const void *a2, unsigned int a3)
{
    signed int i; // r3
    int result; // r0

    // Check the conditions
    if ( (MEMORY[0x50000088] ^ MEMORY[0x5000008C]) == (MEMORY[0x2FFFFFF0] ^ MEMORY[0x2FFFFFF4])
        || *a2 == 0x27 && *(a2 + 1) == 0x4F )
    {
        n = 2;
    }
    // Compare the bytes
    for ( i = 0; i < n && *(a1 + i) == *(a2 + i); ++i )
        ;
    // Report the result
    if ( i == n )
        result = 0;
    else
        result = -1;
    return result;
}
```

Not having access to the SoC reference manual or other sources of information concerning the SoC memory map as seen by SLIMpro, made it very challenging to understand what the first condition meant. The second condition took into account the contents of one of the input buffers: it should have started with bytes `27 4F`. In both cases, the number of bytes to be compared was reset to two.

Considering the goal of the whole effort was to bypass signature verification checks, a reasonable guess was that this function made that possible in some cases. On inspecting signatures present in the extracted Flash images, we found the following BROM signature (after RSA decryption):

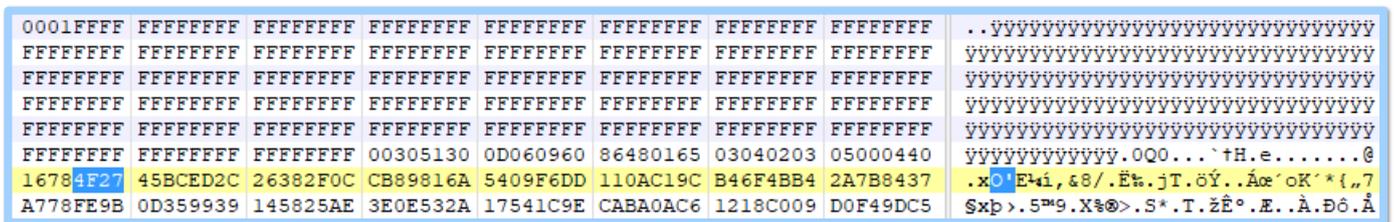


Figure 21. Decrypted BROM signature.

An immediate conclusion was that the second condition was used to circumvent the BROM signature check, while we could only assume that the first condition was somehow involved in circumventing HBOOT signature check.

The same exercise could be repeated against Counterfeit B. The integrated circuit installed on the processor board appeared to have the same pinout as the one found on the implant PCB on Counterfeit A. This was performed by tapping the correct vias on the board, given connecting to the pads of the QFN package footprint was not feasible. Here is the list of transactions performed on the bus for this unit.

```

Start address: 0000, byte count: 0020
Start address: 0020, byte count: 0100
Start address: 0120, byte count: 0400
Start address: 0520, byte count: 0400
Start address: 0920, byte count: 0400
Start address: 0D20, byte count: 0400
Start address: 1120, byte count: 0400
Start address: 1520, byte count: 0400
Start address: 1920, byte count: 0400
Start address: 1D20, byte count: 0400
Start address: 2120, byte count: 0400
Start address: 2520, byte count: 0400
Start address: 2920, byte count: 0400
Start address: 2D20, byte count: 0400
Start address: 3120, byte count: 0400
Start address: 3520, byte count: 0400
Start address: 3920, byte count: 0400
Start address: 3D20, byte count: 0400
Start address: 4120, byte count: 0400
Start address: 4520, byte count: 0400
Start address: 4920, byte count: 0400
Start address: 4D20, byte count: 0400
Start address: 5120, byte count: 0400
Start address: 5520, byte count: 0400
Start address: 5920, byte count: 0400
Start address: 5D20, byte count: 0250

```

The differences were obvious: apart from two shorter reads at the beginning, the whole content was read in one go, with no per-file read patterns as observed in Counterfeit A. Further inspection of the dumped data revealed pseudo-random data with no discernible structure, apart from the first 0x120 bytes. The 32-byte header contained the same data; however, the AMCC file structure was not found. Instead, it appeared the software image starts as is. We could only conclude that the contents were encrypted in some way, and that no further analysis was possible.

0000h:	F0F00312	00FC0100	F4100081	85082554	21160001	000126E0	0408002E	E0050800	šš...ü..ô.....%T!.....&à....à...
0020h:	001CFF3F	319E0000	7D440000	A5440000	CD440000	F5440000	1D450000	00000000	..ÿ?1ž..}D..¥D..ÍD..šD...E.....
0040h:	00000000	00000000	00000000	00000000	00000000	00000000	00000000	DD890000Y%...
0060h:	AF750000	11850000	4B450000	B1450000	4D8D0000	8F850000	79850000	55450000	u.....KE...±E..M.....y...UE..
0080h:	57450000	59450000	5B450000	5D450000	00000000	00000000	00000000	00000000	WE..YE.. [E..]E.....
00A0h:	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
00C0h:	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
00E0h:	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
0100h:	00000000	00000000	00000000	00000000	414D4343	02000100	00400000	505F0000AMCC.....@..P..
0120h:	D41745EF	48F3CF15	B64ACD7C	1FD3022A	4593E749	B7AF4E21	6286AAD7	3273F01B	ô.EiHóÍ.¶JÍ .ó.*E`çI·N!b+*×2sô.
0140h:	2BE3BB3D	5754DC50	3448CFE5	6BD6B47A	F182DBB7	C153ECE9	5E766E5A	F32352B4	+ã»=WTÚP4HÍákÖ`zñ,Û·ÁSié^vnZó#R´
0160h:	B7A924CA	7C0918FC	909E29FF	86261FF0	4B6FC6F1	4AF6662C	873A2E47	848731E1	·@ŠÉ ..ü.ž)ÿ+&.šKoŽñJóf,+:.G.,+lá
0180h:	86028CF1	7A32D259	98109A4E	4ED6B25D	6076C06A	4263A4A7	7A13F5F6	CD24633F	†.Gñz2ÖY~.šNNÖ² `vAjBc*Šz.šóÍšc?

Figure 22. The beginning of the obtained EEPROM/implant dump of Counterfeit B.

Given that, there was an answer to the last question concerning the operation of Counterfeit A: a TOCTOU vulnerability affecting SLIMpro ROM code was exploited in the wild to bypass software signature checks against the SLIMpro secure processing unit. By extension, the issue affects the Genuine unit as well. While one previously published report⁶ regarding issues in the Cisco Catalyst secure boot process was accessible, at the time of writing this paper, no public information was available detailing this or similar issues affecting the Catalyst 2960-X series. This led us to believe this was indeed a previously unknown vulnerability.

It is important to note that the comparison of EEPROM data extracted from the Genuine unit and unpatched data extracted from Counterfeit A showed them to be identical. As the patches were designed to bypass signature checks only, we could conclude there was no "backdoor" code introduced into the SLIMpro environment.

⁶ <https://tools.cisco.com/security/center/content/CiscoSecurityAdvisory/cisco-sa-20190513-secureboot>

3 CONCLUSIONS

The problem of counterfeiting is wide and raises a number of concerns. Not only does it mean a loss of trust in the brand, and loss of revenue for the company whose products get copied, but counterfeit devices also pose a security risk to the victim companies.

The two counterfeit devices provided to us for this research were detected after a software upgrade resulted in a failure. These units were assessed from both software and hardware perspectives to investigate whether the victim company's networks had been compromised via introducing "backdoor access", and to understand how and why counterfeit devices bypassed the platform's authentication security control.

Both units reached their goal of circumventing the implemented platform authentication checks with similar means on the software level by relying on patching the loaded and authenticated application image before control was passed over to the application. The functionality implementing the patches was wrapped into multiple layers of obfuscation. However, these authentication bypasses were performed on each boot and thus were not persistent.

No further functionality was identified on the software level that could be considered as backdoors – both in PowerPC and SLIMpro code. This conclusion was supported by the fact that genuine software was patched on-the-fly and the patches only served to circumvent authenticity checks.

On the hardware level, the two units took quite different approaches as to circumventing boot-time software authentication.

Counterfeit A contained "add-on" circuitry which exploited a race condition in the SLIMpro ROM code to bypass SLIMpro software verification. It did this by intercepting EEPROM control signals, replacing certain bytes in the image being loaded to modify software behavior. It appears the processor PCB in this unit was not modified.

While Counterfeit A only received a post-manufacturing add-on circuitry, the PCB design of Counterfeit B was changed to incorporate the modification of Counterfeit A and replaced the EEPROM completely with an unknown integrated circuit. This signified a considerable resource investment in design, manufacture, and testing of such forged products compared to the more low-cost ad-hoc approach used in Counterfeit A. The board layout and silkscreen similarities also suggested that the people behind this forgery might have either had access to Cisco proprietary engineering documentation such as PCB design files in order to be able to modify them, or they invested heavily in the complicated process of replicating the original board design files based solely on genuine boards.

4 ABOUT THE AUTHOR

Dmitry Janushkevich began his career as a testing- and later embedded-software engineer working on the development of leading-edge solid-state drive technologies. Together with a bachelor's degree in computer systems design, this has given him a strong background in embedded systems design and development for future explorations in their security.

After joining F-Secure Consulting and gaining experience in customer-facing consultancy, embedded systems security became his primary focus. Currently a senior consultant, he has a strong track record in providing security-related consulting for automotive, aerospace, and consumer electronics industries.

5 ABOUT F-SECURE HARDWARE SECURITY TEAM

F-Secure Consulting's Hardware Security team provides information security consulting to the most unique, challenging and critical industries in the world. It delivers industry-leading services to secure hardware, safety-critical embedded systems, software applications and IT infrastructure.

It also provides detailed and comprehensible security analysis of software and hardware systems, along with practical and effective mitigation and protection strategies.

With a vast breadth of experience in hardware and software design and engineering, it's trusted by companies across the globe to assess and test their products and processes. Its work safeguards products from malicious compromise, and in doing so protects the safety of passengers, ensures the resilience of critical infrastructure, and secures company trade secrets and intellectual property.

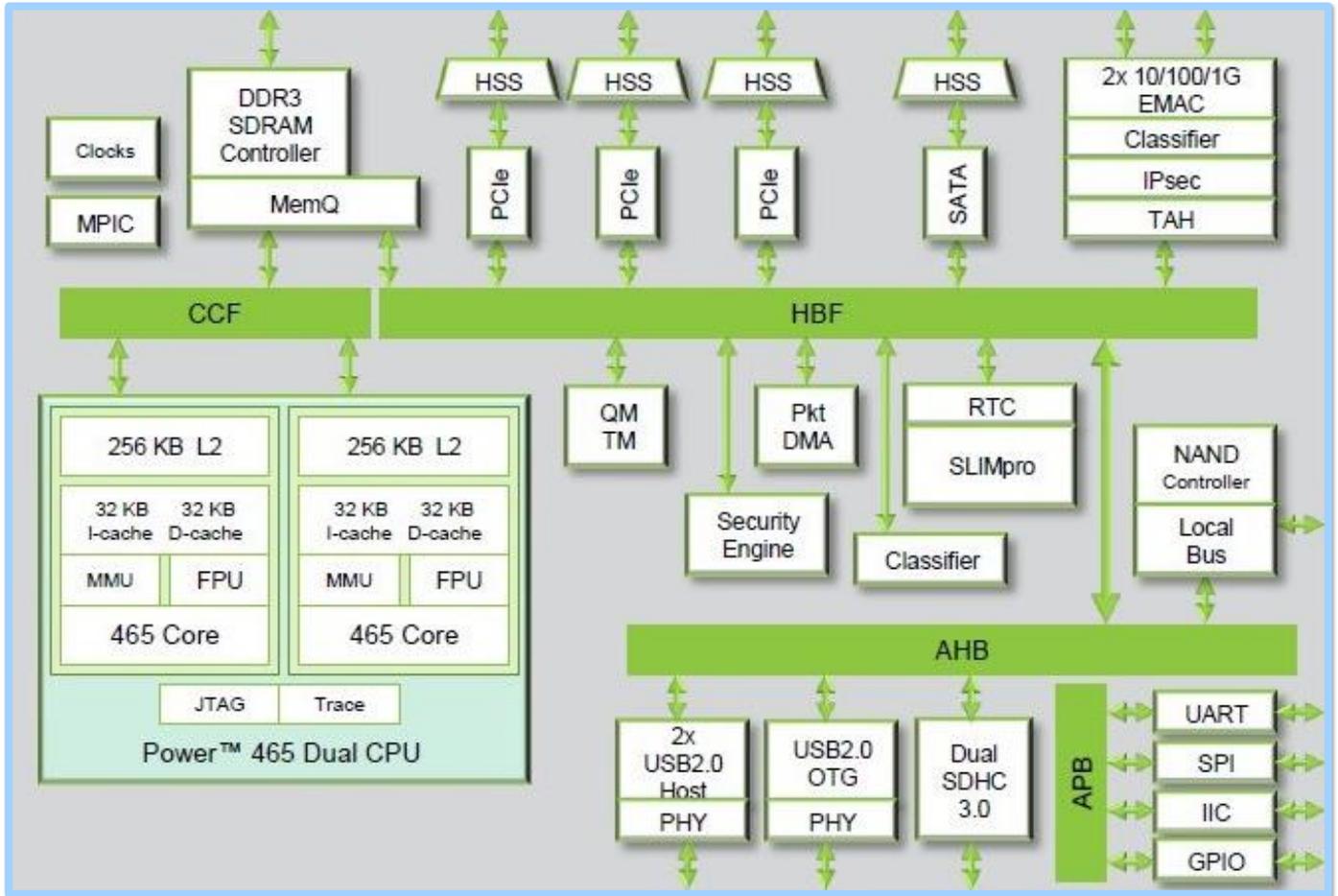


Figure 24. Block diagram of APM86392. Image taken from linuxdevices.org.

Any units relating to security were of particular interest; sometimes these units were not labelled as such but were given cryptic or trademarked names. Two such units stick out: "Security Engine" and "SLIMpro". The "Security Engine" was quite likely to be the one mentioned by the manufacturer as "security subsystem (optional) with acceleration for IPsec, SSL/TLS, SRTP/SRTCP, Kasumi, and public-key protocols (PKA)" on the product page. However, what is SLIMpro?

The same media source describing¹⁰ a previous generation of the same SoC cites the manufacturer: "... AppliedMicro also added its Scalable Lightweight Intelligent Management Processor (SlimPro) coprocessor, which provides advanced power management, security, and concurrency features ..." while also mentioning Secure Boot and namedropping "SlimPro Trusted Management Module" which appeared to be relevant to this research.

Searching for "AppliedMicro Trusted Management Module" yielded a very interesting – and apparently public – presentation¹¹ documenting exactly that unit. We strongly recommended the reader to read through the whole presentation, but here we note that the SLIMpro unit indeed represented the security epicentre of the whole SoC.

¹⁰ <https://linuxdevices.org/dual-core-15ghz-soc-touted-for-power-management-concurrency/>

¹¹ <https://docplayer.net/995240-Appliedmicro-trusted-management-module.html>

Speculatively, and in accordance with what the presentation tried to show, the SLIMpro unit booted first and was responsible for authenticating and starting any code on the PowerPC cores, apart from overall system configuration tasks. This speculation was easy to verify by swapping NOR Flash chips between a counterfeit unit and the genuine one. The genuine unit stopped booting completely, and the message displayed over the serial console was:

```
BOOT FAIL
```

The counterfeit unit, on the other hand, proceeded to start the bootloaders as expected.

6.1.1 THE BOOT PROCESS

There should be enough information – with some guesswork – to piece together the overall boot process from the security standpoint, considering all known processing cores that participated.

1. SLIMpro start up
 - a) SLIMpro was expected to perform authentication of Flash contents
 - b) SLIMpro started (one?) PPC core, setting PC to 0xFFFFF000 (most common configuration)
2. PPC core ran PBL code at Flash offset 0x7FFF000 (previously unidentified)
3. PPC core ran BROM code (by correlating console messages with BROM contents)
 - a) BROM authenticated HBOOT code
4. PPC core ran HBOOT code (by correlating console messages with HBOOT contents)
 - a) HBOOT authenticated application code
5. PPC core ran application code
 - a) Application code authenticated the platform
 - b) Depending on the result, the unit became inoperable

6.2 The MZIP file format

Cisco delivers software updates for Catalyst devices as a single binary file. This meant updates for all system components were carried in this file, whether application software, bootloaders, or microcode. This warranted a closer look into the format of this file.

The file format was identified with the first four bytes being "MZIP" and was referred to as such. Apparently, this was a "Cisco IOS MZIP compressed image" as noted on some sources on the Internet. No specifications were publicly available.

Some existing tools¹² were found which served as a starting point for researching the format, however no tool was found which would handle unpacking of the images at hand. Naturally, the code that was responsible for loading MZIP files should serve as the best reference, so the already obtained genuine bootloader images could be used for that purpose. Quite a lot of references to MZIP were found in the HBOOT image, so the relevant parts were reverse engineered.

On a very high level, the file format was very similar to what would be found in executable file formats – this could easily be seen from what `mziptools` was intended to output: some fixed header data including an entry point address as well as a collection of segments. This meant the file was simply a program image which was loaded and executed by the bootloader chain.

Segments could also be optionally compressed with PKZIP or BZIP2. Judging by the presence of the usual `BZh91AY...` signature, data in our files was indeed compressed with BZIP2. As data is BZIP2 compressed, it was possible to apply usual tools such as `binwalk` for data carving and decompression of relevant sections. While this did not add much insight into the file format, it served as the first step to understand what was contained inside.

While provisions were present for more complicated arrangements, the contents proved to be very simple: one code section and one data section.

Some trailer data was also included. This contained some textual information possibly related to the build configuration and provided versioning information which we didn't really need. Appended there, however, was the image signature apparently used to authenticate the image, likely to be RSA2048 judging by the size. The signature followed the same format used to authenticate both HBOOT and BROM images.

To facilitate future research, a dedicated tool was also developed to unpack and recreate MZIP files.

¹² <https://github.com/bvanheu/linux-cisco/tree/master/mziptools>

Following the AMCC container format description (appendix 6.3), the files inside could be extracted.

This left us with the ppc.bin.key file (the ppc1.bin.key is identical) which might have contained software verification keys, but its format was not immediately obvious. However, one could note that there were repeating data pieces such as hex strings AB 12 34 CD and BE EF CA FE at the start and the end, and a text string C2960X in the middle; all repeated 4 times. This could mean there were 4 keys contained within this file.

0000h:	AE0225AB	1234CD01	00010102	00010103	010A0401	009F0269	A7DE698B	FEC8F57D	0.%.4í.....ÿ.i\$pi<pEö}
0020h:	A81E51E9	DD8C2131	34F17D07	FCB83FB8	8D076AFE	E659987D	03D65431	9EB500E9	..QéY@!14ñ).ú.?,...jpeY"}.ÓTlžn.é
0040h:	90281167	D7595D36	B25DBA15	A269D18A	1FBDBFC5	C6D1DAFF	D93697D2	63AED7DC	.(.g×Y 6°j°.ciñŠ.%.ĂĚŇŮÿŮĚ-Ōc×Ů
0060h:	57C82266	EDBF0A35	BD2896D6	496819B6	0B79EA16	A0CD61FE	A7727736	1EE6BD45	WĚ"fiž.5%(-ŌIn.Ŧ.yè. İap\$rw6.æ%Ě
0080h:	40CDD221	49353D06	659FF4ED	03DAD221	D7B7FB7F	0AFCBE25	C8A8DBDA	01270A88	@İŌ!İ5=-.eYđi.ŮŌ!×.ú.ú.%Ě"ŮŮ.'.
00A0h:	D707AA09	83C7BA4C	1D1B218C	C582E55E	F7328AC5	2484CA33	44193754	D88B65EE	×.°.fç°L...!ĚĂ,â^+2ŠĂĚ,Ě3D.7TŌ<ei
00C0h:	1067F54C	DE2A4626	0BE1B0E1	F1DDFF96	676F5E4D	ACD339F6	77D5D96C	1A1A55A9	.gŌLp*F&.á°áñŸy-go^M-Ō9öwŮŮ!..U@
00E0h:	E11E6930	ED864576	346C37E1	0F15F75F	B8C3ECC2	F6034ED2	C6B002DE	52AC78FA	á.ı0i†Ev4l7á..._ĚİĂŌ.ŇŌĚ°.PR-xú
0100h:	2E1A38AC	F80649FA	E7C19329	1C9B7F40	18B5FC13	AD050004	00010001	06000141	..8-ø.İúçĂ").>.@.nú.-.....A
0120h:	07000643	32393630	58080100	8B57DD7C	40D96A38	45A3779B	6E55BF08	0A1C8BD0	...C2960X...<WY @Ůj8Eew>nŮž....<Đ
0140h:	4D28BFCE	1BD565F8	89BC231B	4D79649D	7B740F33	84840A37	2644D9CE	3055B49E	M(žĚ.Ōeø%#Myd.(t.3...7&DŮİŮŮ'ž
0160h:	3EE3429A	7FFBCED5	55F2D2C5	82279F03	26EA011A	5B80601F	0849A528	41BA0E9C	>ĂŠš.úİŮŮŌĂ,'Ÿ.æè..[Ě...İŸ(A°.œ
0180h:	5A5F4E79	C8D84A63	9CF10FB7	50783ECE	09B7B4E3	B8B9CD66	A00C4159	0A05B19B	Z.NyĚŮJceñ.·Px>İ.·'ă.İf .AY..±
01A0h:	BF8383E4	29EEFD8D	690F9ABA	C9D3A647	EF659769	9F0FAE9F	AD224AD0	FF14B938	(žfă)ıy.ı.š°ÉŌ;Gie-ıŸ.ŌŸ-"JĐŸ.'8
01C0h:	718B4205	E5710B96	3D57C78C	E7113F02	674D623F	7A718E2F	B543DF2C	CCBBFC64	q<B.ăq.-=WçĚç.?.gMb?zqž/μCB,İ,üđ
01E0h:	EABEFB8B	ABD8670C	D0FBA887	0D8DF8C6	46306706	31030EEE	6DED2CB9	2FB5FE57	ê%ú<«Ōg.Đù'+...øEFŮg.l..ımi,°/μpW
0200h:	3B2F7E7F	726DEC15	15864B64	A571E670	506C8BED	521B6912	B76E9F1B	891E7AE1	;/~.rmi..†KđŸgqPł<ıR.ı.nŸ.%.zá
0220h:	4A9E2E5D	C4228725	0A80E1DA	BEEFCAFE	AE0225AB	1234CD01	00010202	00010103	Jž.]Ă"%.ĚĂŮpĚp@.%.4í.....
0240h:	010A0401	00D12BAD	146859BB	19CB3F09	62F46EDE	F40C3249	373D8AEL	DC243E73Ň+-.hY».Ě?.bŏnBđ.2İ7=ŠăŮŮ>s

Figure 26. The beginning of ppc.bin.key and the BEEFCAFE marker.

After spending several hours on decoding the format of these entries, we were able to provide the following summary: the format loosely followed a tag-length-value (TLV) structure with 7-byte fixed header AE 02 25 AB 12 34 CD and 4-byte fixed trailer BE EF CA FE, tags were one byte, lengths were two bytes. Tag 04 was the RSA modulus, and tag 05 is the RSA public exponent (always 0x10001, a typical value).

Two unique public key moduli were extracted from this file; they are reproduced below for reference. It is important to highlight the fact these keys can only be used to verify signatures.

```

9f0269a7de698bfec8f57da81e51e9dd8c213134f17d07fcb83fb88d076afee659987d03d654319eb500e9902811
67d7595d36b25dba15a269d18a1fbd8fc5c6d1daffd93697d263aed7dc57c82266edbf0a35bd2896d6496819b60b
79ea16a0cd61fea77277361ee6bd4540cdd22149353d06659ff4ed03dad221d7b7fb7f0afcb25c8a8dbda01270a
88d707aa0983c7ba4c1d1b218cc582e55ef7328ac52484ca3344193754d88b65ee1067f54cde2a46260be1b0e1f1
ddff96676f5e4dacd339f677d5d96c1a1a55a9e11e6930ed864576346c37e10f15f75fb8c3ecc2f6034ed2c6b002
de52ac78fa2e1a38acf80649fae7c193291c9b7f4018b5fc13ad

d12bad146859bb19cb3f0962f46edef40c3249373d8ae1dc243e735825b39073b2d0a507d658e4815eb0fceccce9
84741f2b69e8637388db967c337469001f1727201355242a60158fc5f84cce0bada77c626b33fc7334e1f8ebacfa
de0485721bdb6df4cf5c496deb69c152eb67fa2edf0bc8531a875fc6d7b9d29662d3022e894805956d76587f624b
0ed12cabbef4d91f4754bc1ae091070072610fb54d03d1efe7075b70e62473f914503186e550c2d5e4177cba2302
453cd8c07b78918604eb648a5dac02d4649a5d3bdeb6c1ac5a129e553f3905226d2abc291b0293cfe7d3b260bfb1
320c7186f6700ea4729b2f26a402cae22aedd2add7cc96ebf027

```

Investigating the signature format, we could conclude it followed the same TLV format albeit without headers and trailers. Tag 0B was identified to be the RSA signature.

Putting this knowledge to work, we could easily confirm whether our deductions were correct with Python:

```

>>>                                     pubmod1                                     =
0x9f0269a7de698bfec8f57da81e51e9dd8c213134f17d07fcb83fb88d076afee659987d03d654319eb500e99028
1167d7595d36b25dba15a269d18a1fbd8fc5c6d1daffd93697d263aed7dc57c82266edbf0a35bd2896d6496819b6
0b79ea16a0cd61fea7277361ee6bd4540cdd22149353d06659ff4ed03dad221d7b7fb7f0afcbe25c8a8dbda0127
0a88d707aa0983c7ba4c1d1b218cc582e55ef7328ac52484ca3344193754d88b65ee1067f54cde2a46260be1b0e1
f1ddff96676f5e4dacd339f677d5d96c1a1a55a9e11e6930ed864576346c37e10f15f75fb8c3ecc2f6034ed2c6b0
02de52ac78fa2e1a38acf80649fae7c193291c9b7f4018b5fc13ad
>>>                                     pubmod2                                     =
0xd12bad146859bb19cb3f0962f46edef40c3249373d8ae1dc243e735825b39073b2d0a507d658e4815eb0fceccc
e984741f2b69e8637388db967c337469001f1727201355242a60158fc5f84cce0bada77c626b33fc7334e1f8ebac
fade0485721bdb6df4cf5c496deb69c152eb67fa2edf0bc8531a875fc6d7b9d29662d3022e894805956d76587f62
4b0ed12cabbef4d91f4754bc1ae091070072610fb54d03d1efe7075b70e62473f914503186e550c2d5e4177cba23
02453cd8c07b78918604eb648a5dac02d4649a5d3bdeb6c1ac5a129e553f3905226d2abc291b0293cfe7d3b260bf
b1320c7186f6700ea4729b2f26a402cae22aed2add7cc96ebf027
>>> pubexp = 0x10001
>>>                                     sig                                       =
0x8ec4a43c0658ea28ea529604116e56f2c5924cc865937ed4d33ad037eb95696dcce523d6f3f6f8d0724b48b6a0
1e1a6cd21984a9325ca72cf9a6c326502a565c0e7564ac7365f1e8e62d277f7072cdc22e861a17bfbce6d57fbfa7
ba2455b88389da2667e600b0599a0e069f7300c8ca14298c12db5440dab7007fc4e24369aff9313f1b98f35f0726
bc951f7076a12fdfa10447b93c824fae3aca57c20a6def3317a38a59936ea29a74a6d4e8696c7eb9aeab3b02f37b
6dae9c0ab9c01db40988581799f13838b3325b1f13ffbc012b89a6f2b8f2baec554578440f7e68b0c46ce525a487
b91f0f7fc628e7f952d8d3a8059b654d4c4b40b539b2b82e15cfab
>>> "%0512x" % pow(sig, pubexp, pubmod1)
'0001ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
003051300d0609608648016
503040203050004408ff1850fc5313c39ef350d8c2871797c8b42c5530ae6366bc47d3ea81d1ec8ccee5d1087e5
6b1b330787445237a69b61730ff25d697cfe6e64ee3e77cad4489'

```

Judging by the well-formed decrypted data, we found the right software signature verification key; if the key was incorrect, the decrypted data would appear random. The format appeared to fit what was defined in PKCS#1 v1.5¹³ and the data told us the hash function used was SHA-512.

¹³ <https://tools.ietf.org/html/rfc2313>