Looking for sophisticated malware in IoT devices

securelist.com/looking-for-sophisticated-malware-in-iot-devices/98530/

Authors

Expert Noushin Shabab

One of the motivations for this post is to encourage other researchers who are interested in this topic to join in, to share ideas and knowledge and to help build more capabilities in order to better protect our smart devices.

Research background

Smart watches, smart home devices and even smart cars – as more and more connected devices join the IoT ecosystem, the importance of ensuring their security becomes patently obvious.

It’s widely known that the smart devices which are now inseparable parts of our lives are not very secure against cyberattacks. Malware targeting IoT devices has been around for more than a decade. Hydra, the first known router malware that operated automatically, appeared in 2008 in the form of an open-source tool. Hydra was an open-source prototype of router malware. Soon after Hydra, in-the-wild malware was also found targeting network devices. Since then, different botnet families have emerged and become widespread, including families such as Mirai, Hajime and Gafgyt.

Apart from the malware mentioned above, there are also vulnerabilities found in communication protocols used in IoT devices, such as Zigbee, which can be exploited by an attacker to target a device and to propagate malware to other devices in a network, similar to computer worms.

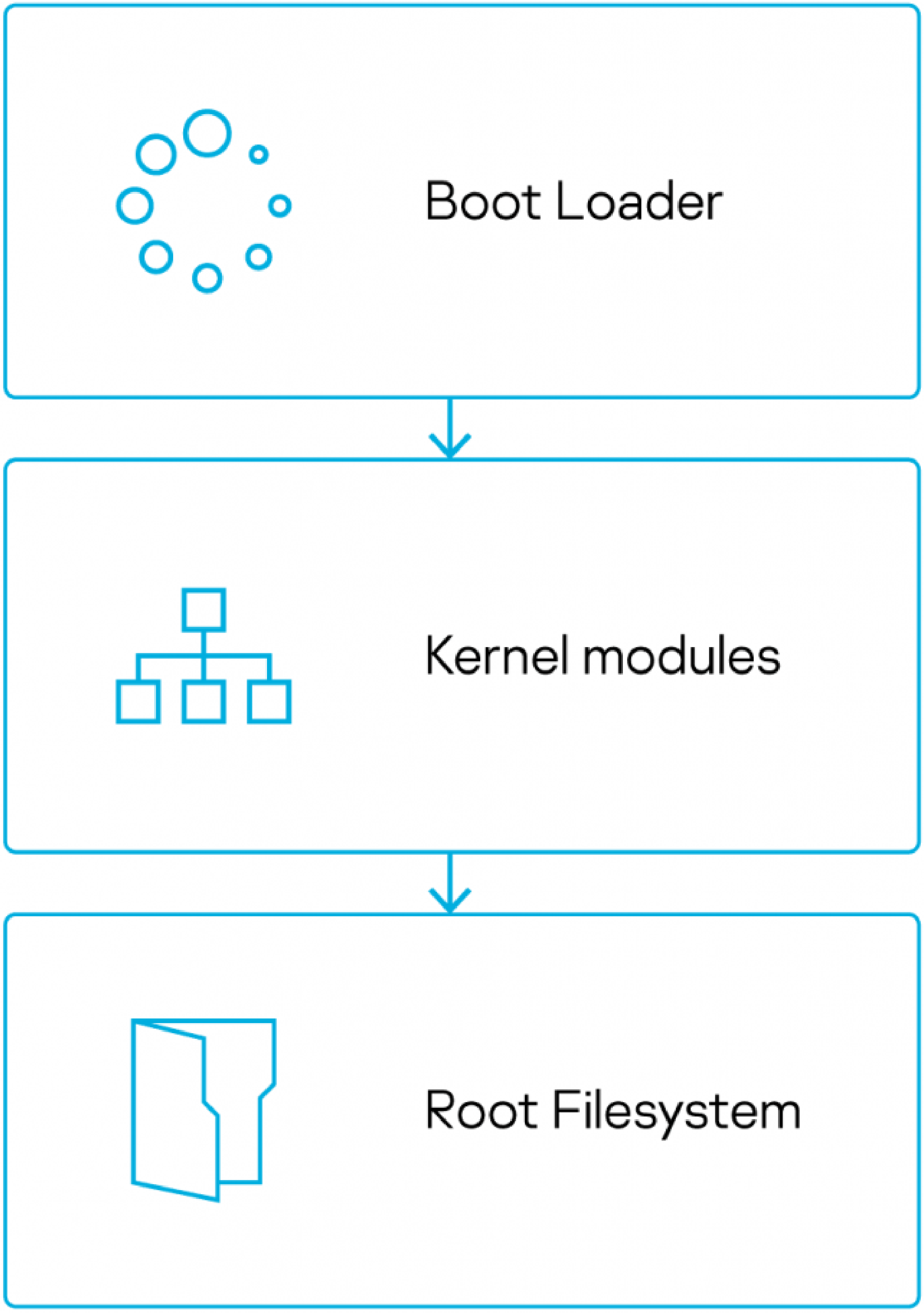
In this research, we are focusing on hunting low-level sophisticated attacks targeting IoT devices and, in particular, taking a closer look at the firmware of IoT devices to find backdoor implants, modifications to the boot process and other malicious alterations to different parts of the firmware.

Now, let’s talk about the structure of the firmware of an IoT device in order to get a better understanding of the different components.

IoT firmware structure

Regardless of the CPU architecture of an IoT device, the boot process consists of the following stages: the boot loader, the kernel and the file system (shown in the figure below). When an IoT device is switched on, the code from the onboard SoC (System on Chip) ROM transfers control to the bootloader, the bootloader loads the kernel and kernel then mounts the root file system.

The boot loader, the kernel and the file system also comprise the three main components of typical IoT firmware.



IoT boot process

There are a variety of CPU architectures used in IoT devices. Therefore, being able to analyze and understand the different components of firmware requires a good understanding of these architectures and also their instruction set. The most common CPU architectures among IoT devices are:

- ARM
- MIPS
- PowerPC
- SPARC

Possible attack scenarios

Understanding the firmware structure enables us to think about how an attacker might take advantage of the various components when deploying a stealth attack that's difficult to detect.

The bootloader is the first component that takes control of the system. Therefore, targeting the bootloader offers an attacker a perfect opportunity to carry out malicious tasks. It also means that an attack can remain persistent after a reboot.

An attacker can also manipulate the kernel modules. The majority of IoT devices use the Linux kernel. As easy as it is for a developer to customize and choose whatever they need from the Linux kernel, an attacker who manages to access and manipulate the device firmware can also add or edit kernel modules.

Moving on to the file system, there are also a number of common file systems used in IoT devices. These file systems are usually easy to work with. An attacker can extract, decompress and also mount the original file system from the firmware, add malicious modules and compress it again using common utilities. For instance, SquashFS is a compressed file system for Linux that is quite common among IoT manufacturers. It's very straightforward to mount or uncompress a SquashFS file system using the Linux utilities "squashfs" and "unsquashfs".

Challenges of this research

Obtaining firmware

There are different ways to obtain firmware. When deciding to investigate, sometimes you want the acquired firmware to belong to the exact same device with the same specifications; and you also want it to be deployed on the device through some specific means. For example, you suspect that the network through which the firmware is updated has been compromised and you consider the possibility of the firmware being manipulated in transition between the vendor's server and the device, hence you want to investigate the updated firmware to validate its integrity. In another example scenario, you might have bought a device from a third-party vendor and have doubts about the firmware's authenticity.

There are also a large number of IoT devices where the manufacturers don't implement any ways to get access to the firmware, not even for an update. The device is released from the manufacturer with firmware for its lifetime.

In such cases the surest way to obtain the exact firmware you are after, is to extract the firmware from the device itself.

The main challenge here is that this process requires a certain domain-specific knowledge and also specialist hardware/software experience of working with embedded systems. This approach also lacks scalability if you want to find sophisticated attacks targeting IoT devices in general.

Among the various ways of obtaining IoT firmware, the easiest way is to download the firmware from the device manufacturer's website. However, not all manufacturers publish their firmware on their website. In general, a large number of IoT devices can only be updated through the device physical interface or via a specific software application (e.g. mobile app) used to manage the device.

When downloading firmware from a vendor's website, a common issue is that you might not be able to find older versions of the firmware for your specific device model. Let's also not forget that in many cases the published firmware binaries are encrypted and can only be decrypted through the older firmware modules installed on the device.

Understanding firmware

According to Wikipedia, "firmware is a specific class of computer software that provides the low-level control for a device's specific hardware. Firmware can either provide a standardized operating environment for more complex device software (allowing more hardware-independence), or, for less complex devices, act as the device's complete operating system, performing all control, monitoring and data manipulation functions."

Even though the main components of firmware are almost always the same, there is no standard architecture for firmware.

The main components of firmware are typically the bootloader, the kernel module and the file system; but there are many other components that can be found in a firmware binary, such as the device tree, the digital certificates, and other device specific resources and components.

Once the firmware binary has been retrieved from the vendor's website, we can then begin analyzing it and taking it apart. Given the specialized nature of the firmware, its analysis is very challenging and rather involved. To get some more details about these challenges and how to tackle them, refer to the "IoT firmware analysis" section.

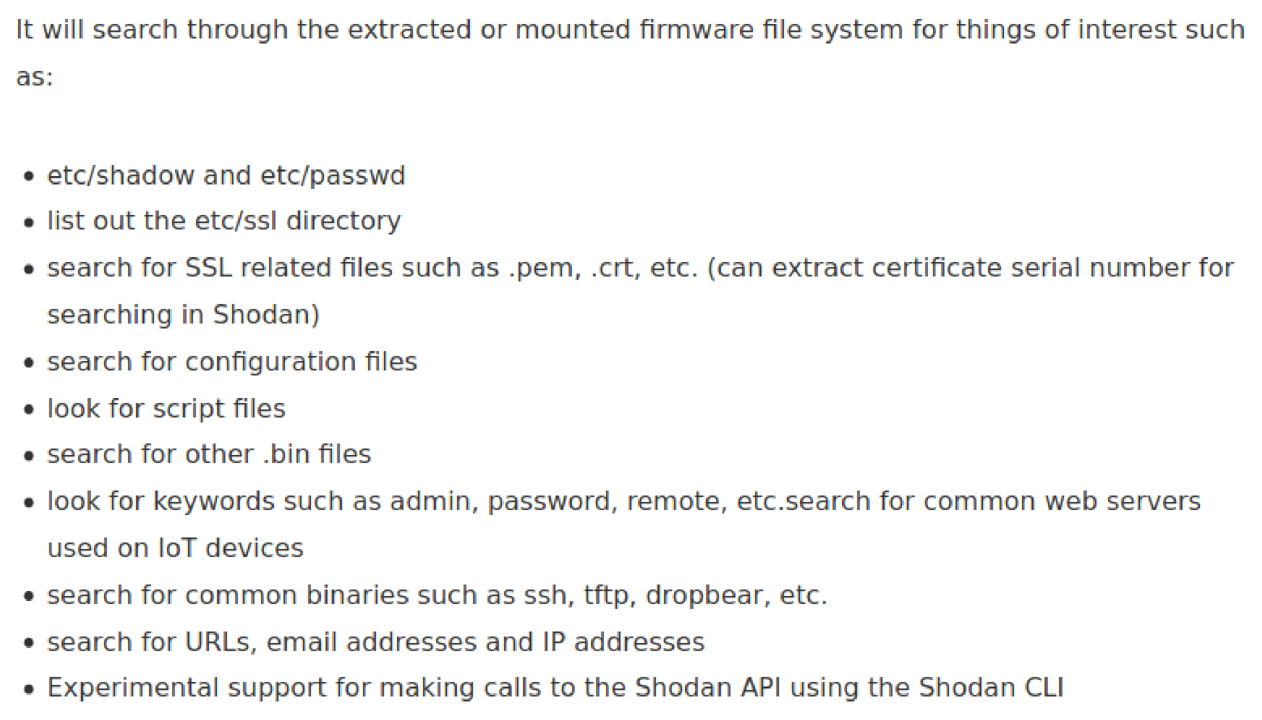
Finding suspicious elements in firmware

After the components of the firmware have been extracted, you can start to look for suspicious modules, code snippets or any sort of malicious modifications to the components.

An easy step to start with, is to scan the file system contents against a set of YARA rules which can be based on known IoT malware or heuristic rules. You can also scan the extracted file system contents with an antivirus scanner.

Something else you can do is look for the startup scripts inside the file system. These scripts contain lists of modules that get loaded every time the system boots up. The address to a malicious module might have been inserted in a script like this with malicious intent.

Here the Firmwalker tool can help with scanning an extracted file system for potentially vulnerable files.



It will search through the extracted or mounted firmware file system for things of interest such as:

- etc/shadow and etc/passwd
- list out the etc/ssl directory
- search for SSL related files such as .pem, .crt, etc. (can extract certificate serial number for searching in Shodan)
- search for configuration files
- look for script files
- search for other .bin files
- look for keywords such as admin, password, remote, etc.search for common web servers used on IoT devices
- search for common binaries such as ssh, tftp, dropbear, etc.
- search for URLs, email addresses and IP addresses
- Experimental support for making calls to the Shodan API using the Shodan CLI

.

Firmwalker capabilities (https://craigsmith.net/firmwalker/)

Another place to investigate is the bootloader component, though this is more challenging.

There are a number of common bootloaders used in IoT devices with U Boot being the most common. U Boot is highly customizable, which makes it very difficult to determine whether the compiled code has been manipulated or not. Finding malicious modifications becomes even more complicated with uncommon or custom bootloaders.

IoT firmware analysis

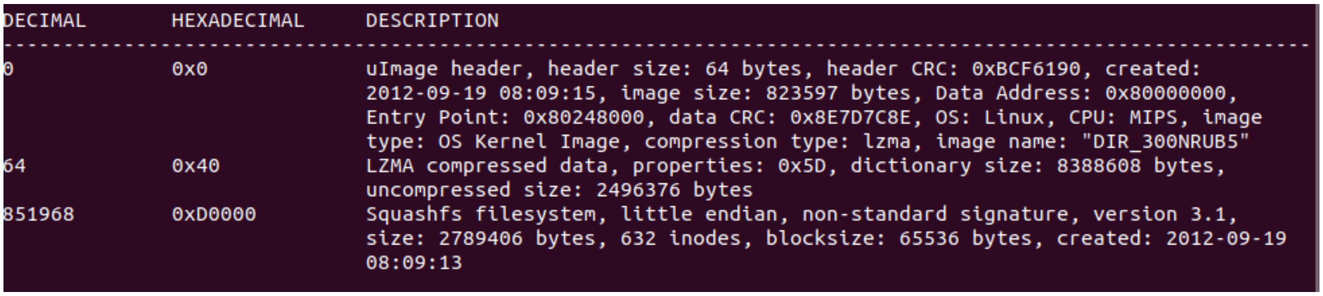
There are a variety of open-source and closed-source tools that can help with firmware analysis. The best approach is to use a combination of the tools and techniques suggested by experienced firmware analysts.

Let’s begin with Binwalk, the most comprehensive firmware analysis tool. Binwalk scans the firmware binary and looks for known patterns and signatures.

It has a large collection of signatures for various bootloaders and file systems used in IoT devices. It also has signatures for common encryption and compression algorithms along with the respective routines for decompression and decoding.

Binwalk is also capable of extracting the components it finds in the firmware binary.

The following screenshot shows the output of a Binwalk scan on a sample firmware binary:



```
DECIMAL       HEXADECIMAL     DESCRIPTION
--------------------------------------------------------------------------------------------------------------
0             0x0             uImage header, header size: 64 bytes, header CRC: 0xBCF6190, created:
                              2012-09-19 08:09:15, image size: 823597 bytes, Data Address: 0x80000000,
                              Entry Point: 0x80248000, data CRC: 0x8E7D7C8E, OS: Linux, CPU: MIPS, image
                              type: OS Kernel Image, compression type: lzma, image name: "DIR_300NRUB5"
64            0x40            LZMA compressed data, properties: 0x5D, dictionary size: 8388608 bytes,
                              uncompressed size: 2496376 bytes
851968        0xD0000         Squashfs filesystem, little endian, non-standard signature, version 3.1,
                              size: 2789406 bytes, 632 inodes, blocksize: 65536 bytes, created: 2012-09-19
                              08:09:13
```

Binwalk scan output

In this screenshot, Binwalk has found and printed out the header, the bootloader and the Linux kernel as well as the file system. There are also metadata details that have been extracted from the headers and the components themselves, such as the type and size of each component, CRC checksums, important addresses, CPU architecture, image name and so on. Now you can go on and use Binwalk itself to extract the above-mentioned parts, or manually calculate the sizes and extract the parts based on the start offset found by Binwalk.

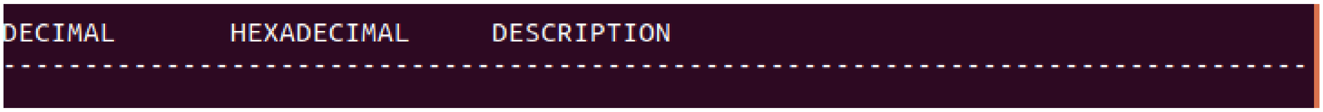
After extracting the components of the firmware, you can go on and extract, decompress or even mount the file system and start investigating the file system content. You can also look at the bootloader code in a disassembler, or debug it through a debugger.

However, doing firmware analysis is not always that straightforward. Firmware is so varied and diverse that understanding its structure and extracting the components is usually quite complicated.

Let’s take a close look at another sample firmware and try to understand its structure.

1. Binwalk firmware.bin

The Binwalk scan shows nothing in the result. This means that Binwalk could not find any known signatures.



```
DECIMAL       HEXADECIMAL     DESCRIPTION
--------------------------------------------------------------------------------
```

Binwalk scan output

We can see in this case that the simple Binwalk scan was not very helpful. However, be aware that there are other tools and techniques we can use to learn more about the structure of this firmware.

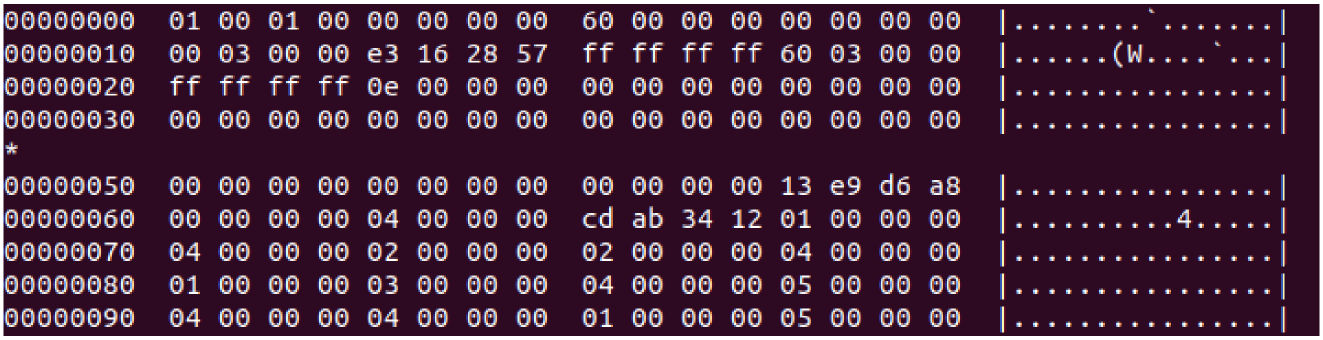
2. File firmware.bin

Let's next try the Linux file utility on the firmware binary.

```
firmware.bin: Targa image data - Map 65536 x 65536 x 0 +96 - 3-bit alpha ""
```

File utility output

The file utility shows the file type as Targa image data. By looking at the beginning of the binary file, and doing a Google search on the Targa image data signature, the result is obviously a false positive.



```
00000000  01 00 01 00 00 00 00 00  60 00 00 00 00 00 00 00  |........`.......|
00000010  00 03 00 00 e3 16 28 57  ff ff ff ff 60 03 00 00  |......(W....`...|
00000020  ff ff ff ff 0e 00 00 00  00 00 00 00 00 00 00 00  |................|
00000030  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
00000050  00 00 00 00 00 00 00 00  00 00 00 00 13 e9 d6 a8  |................|
00000060  00 00 00 00 04 00 00 00  cd ab 34 12 01 00 00 00  |..........4.....|
00000070  04 00 00 00 02 00 00 00  02 00 00 00 04 00 00 00  |................|
00000080  01 00 00 00 03 00 00 00  04 00 00 00 05 00 00 00  |................|
00000090  04 00 00 00 04 00 00 00  01 00 00 00 05 00 00 00  |................|
```

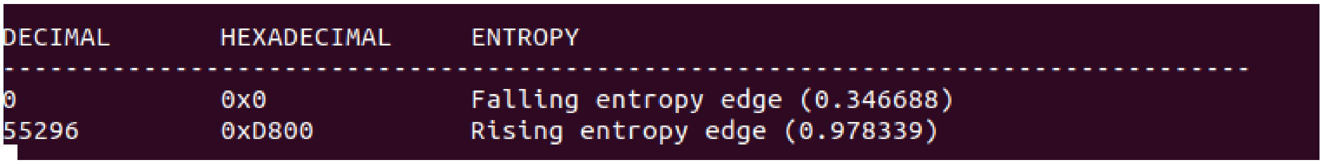
First bytes of the firmware binary

This is because the first bytes of the firmware file, 0x01010000, match the Targa image data signature. See the screenshot above.

3. Binwalk -E firmware.bin

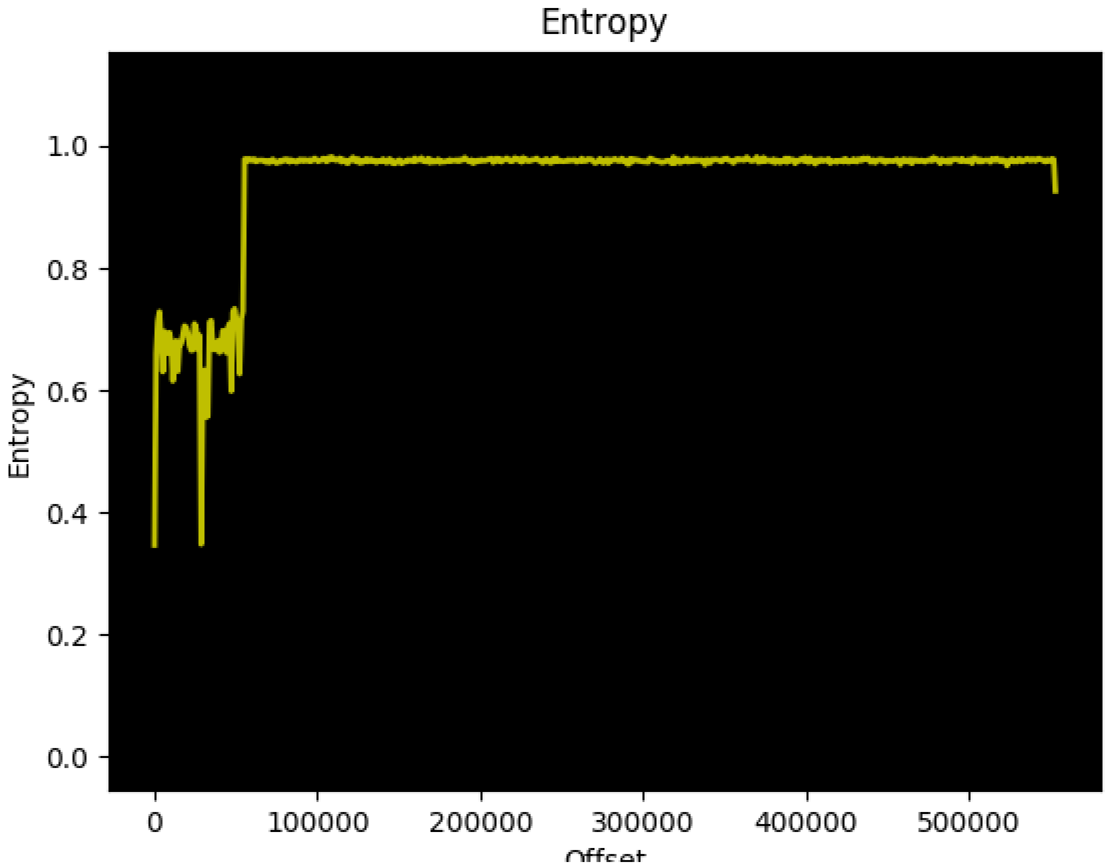
Let's use another capability of Binwalk and check the entropy of the firmware binary.

Running Binwalk using the “-E” command option gives an entropy diagram for the firmware file and some additional details such as the offset for falling and rising entropy.



```
DECIMAL       HEXADECIMAL     ENTROPY
--------------------------------------------------------------------------------
0             0x0             Falling entropy edge (0.346688)
55296         0xD800          Rising entropy edge (0.978339)
```

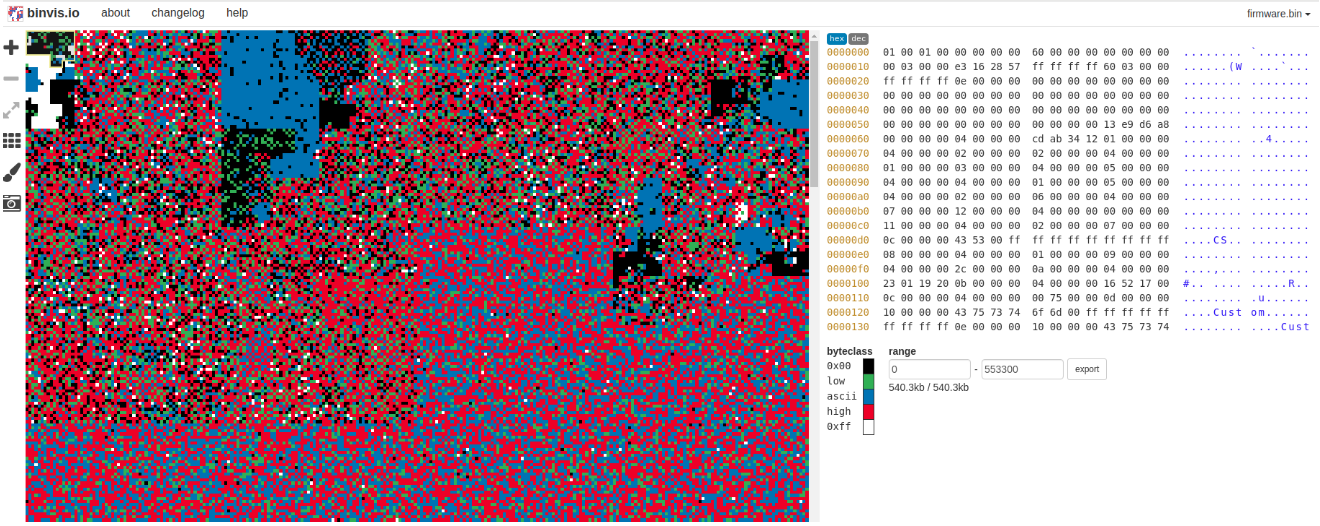
Entropy details



Entropy diagram

Entropy figures close to 1 indicate compression, while the lower entropy figures indicate uncompressed and unencrypted areas. As can be seen from the screenshots above, the offset 55296 (0xD800) is the beginning of the high entropy part.

There is also another tool that can be helpful in visualizing the binary. With the help of binvis.io you can see the contents of the firmware file and its visualization in two side-by-side panes. Different parts are shown in different colors based on their entropy. (binvis.io)

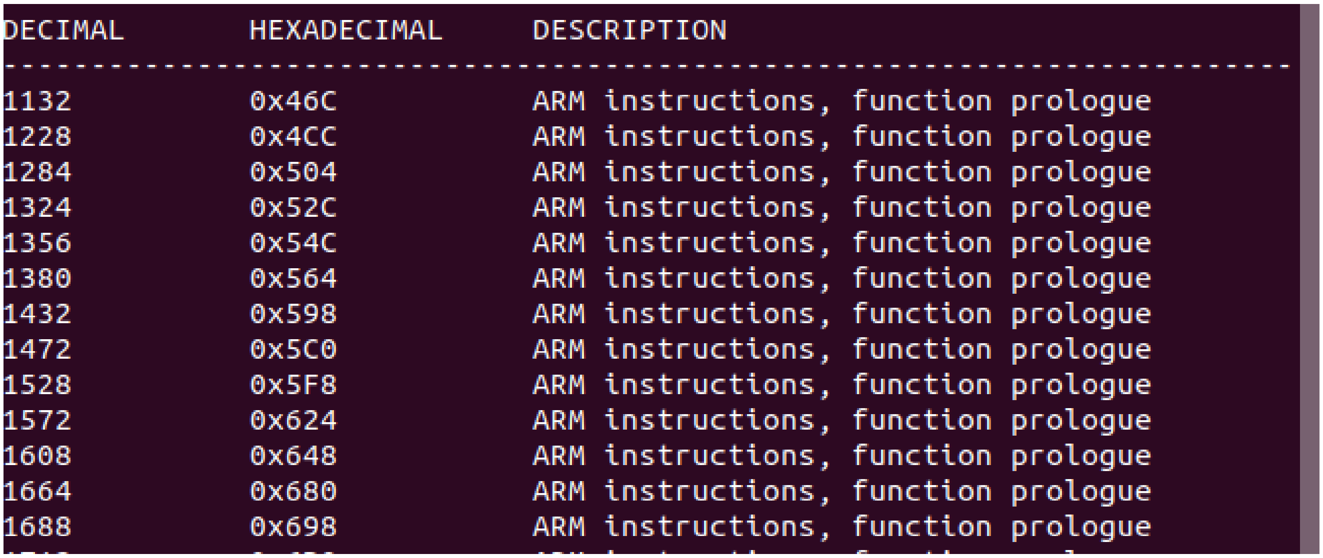


```
0000000  01 00 01 00 00 00 00 00  60 00 00 00 00 00 00 00  ........ `.......
0000010  00 03 00 00 e3 16 28 57  ff ff ff ff 60 03 00 00  ......(W ....`...
0000020  ff ff ff ff 0e 00 00 00  00 00 00 00 00 00 00 00  ........ ........
0000030  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ........ ........
0000040  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ........ ........
0000050  00 00 00 00 00 00 00 00  00 00 00 00 13 e9 d6 a8  ........ ........
0000060  00 00 00 00 04 00 00 00  cd ab 34 12 01 00 00 00  ........ ..4.....
0000070  04 00 00 00 02 00 00 00  02 00 00 00 04 00 00 00  ........ ........
0000080  01 00 00 00 03 00 00 00  04 00 00 00 05 00 00 00  ........ ........
0000090  04 00 00 00 04 00 00 00  01 00 00 00 05 00 00 00  ........ ........
00000a0  04 00 00 00 02 00 00 00  06 00 00 00 04 00 00 00  ........ ........
00000b0  07 00 00 00 12 00 00 00  04 00 00 00 00 00 00 00  ........ ........
00000c0  11 00 00 00 04 00 00 00  02 00 00 00 07 00 00 00  ........ ........
00000d0  0c 00 00 00 43 53 00 ff  ff ff ff ff ff ff ff ff  ....CS.. ........
00000e0  08 00 00 00 04 00 00 00  01 00 00 00 09 00 00 00  ........ ........
00000f0  04 00 00 00 2c 00 00 00  0a 00 00 00 04 00 00 00  ....,... ........
0000100  23 01 19 20 0b 00 00 00  04 00 00 00 16 52 17 00  #.. .... .....R..
0000110  0c 00 00 00 04 00 00 00  00 75 00 00 0d 00 00 00  ........ .u......
0000120  10 00 00 00 43 75 73 74  6f 6d 00 ff ff ff ff ff  ....Cust om......
0000130  ff ff ff ff 0e 00 00 00  10 00 00 00 43 75 73 74  ........ ....Cust
```

Visualization of the firmware created by binvis.io

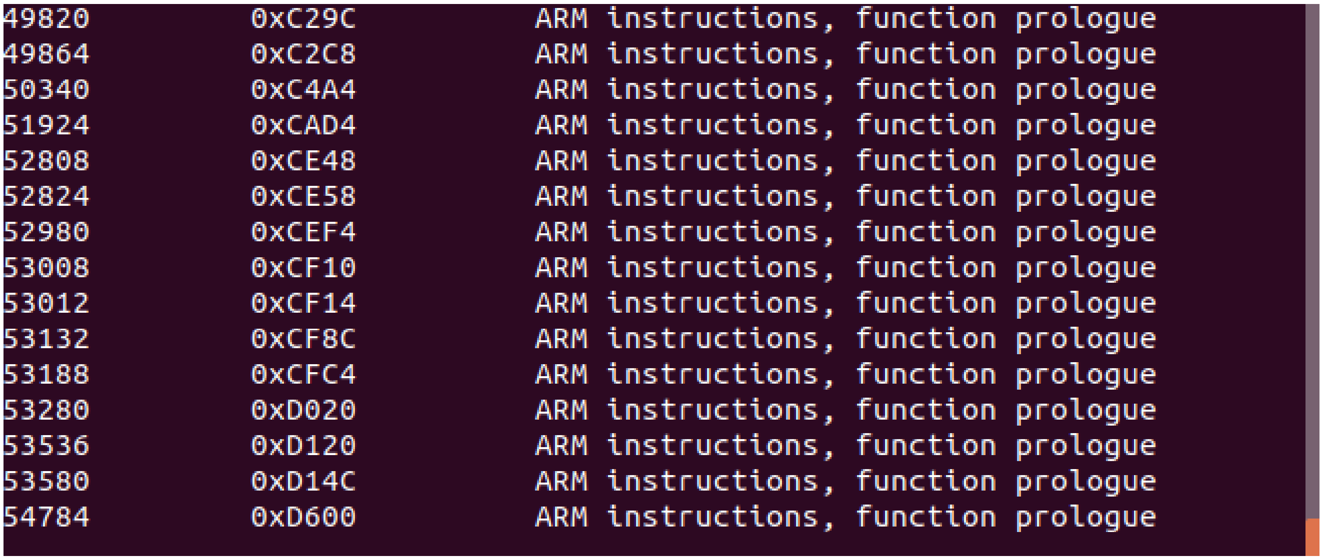
4. Binwalk -A firmware.bin

Binwalk can also scan the binary file for common executable opcode signatures.



```
DECIMAL       HEXADECIMAL     DESCRIPTION
--------------------------------------------------------------------------------
1132          0x46C           ARM instructions, function prologue
1228          0x4CC           ARM instructions, function prologue
1284          0x504           ARM instructions, function prologue
1324          0x52C           ARM instructions, function prologue
1356          0x54C           ARM instructions, function prologue
1380          0x564           ARM instructions, function prologue
1432          0x598           ARM instructions, function prologue
1472          0x5C0           ARM instructions, function prologue
1528          0x5F8           ARM instructions, function prologue
1572          0x624           ARM instructions, function prologue
1608          0x648           ARM instructions, function prologue
1664          0x680           ARM instructions, function prologue
1688          0x698           ARM instructions, function prologue
```

First function prologues found in the file



```
49820         0xC29C          ARM instructions, function prologue
49864         0xC2C8          ARM instructions, function prologue
50340         0xC4A4          ARM instructions, function prologue
51924         0xCAD4          ARM instructions, function prologue
52808         0xCE48          ARM instructions, function prologue
52824         0xCE58          ARM instructions, function prologue
52980         0xCEF4          ARM instructions, function prologue
53008         0xCF10          ARM instructions, function prologue
53012         0xCF14          ARM instructions, function prologue
53132         0xCF8C          ARM instructions, function prologue
53188         0xCFC4          ARM instructions, function prologue
53280         0xD020          ARM instructions, function prologue
53536         0xD120          ARM instructions, function prologue
53580         0xD14C          ARM instructions, function prologue
54784         0xD600          ARM instructions, function prologue
```

Last function prologues found in the file

As we can see from the screenshot above, the result of the opcode signature check is actually very helpful! First, we can see that the firmware belongs to an ARM device.

Second, if we consider the offsets of the first and last function prologue signatures, we get an indication that these are the sections of the firmware binary that contain code.

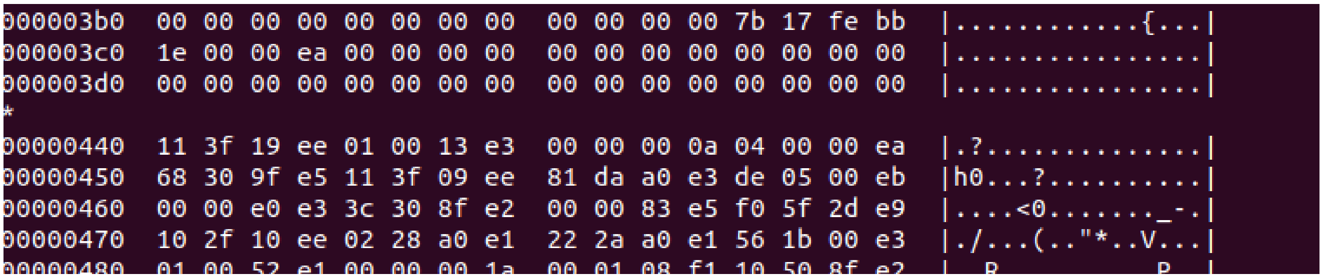
From the screenshot, we can also see that the last function is found at the address 0xD600, which is just 0x200 bytes before the part where the entropy goes up. From this, we can make an educated guess that this offset is likely the end of the code of the bootloader and the beginning of the compressed kernel modules.

5. Hexdump -C

hexdump -C firmware.bin | grep -C 4 -e “^*$”

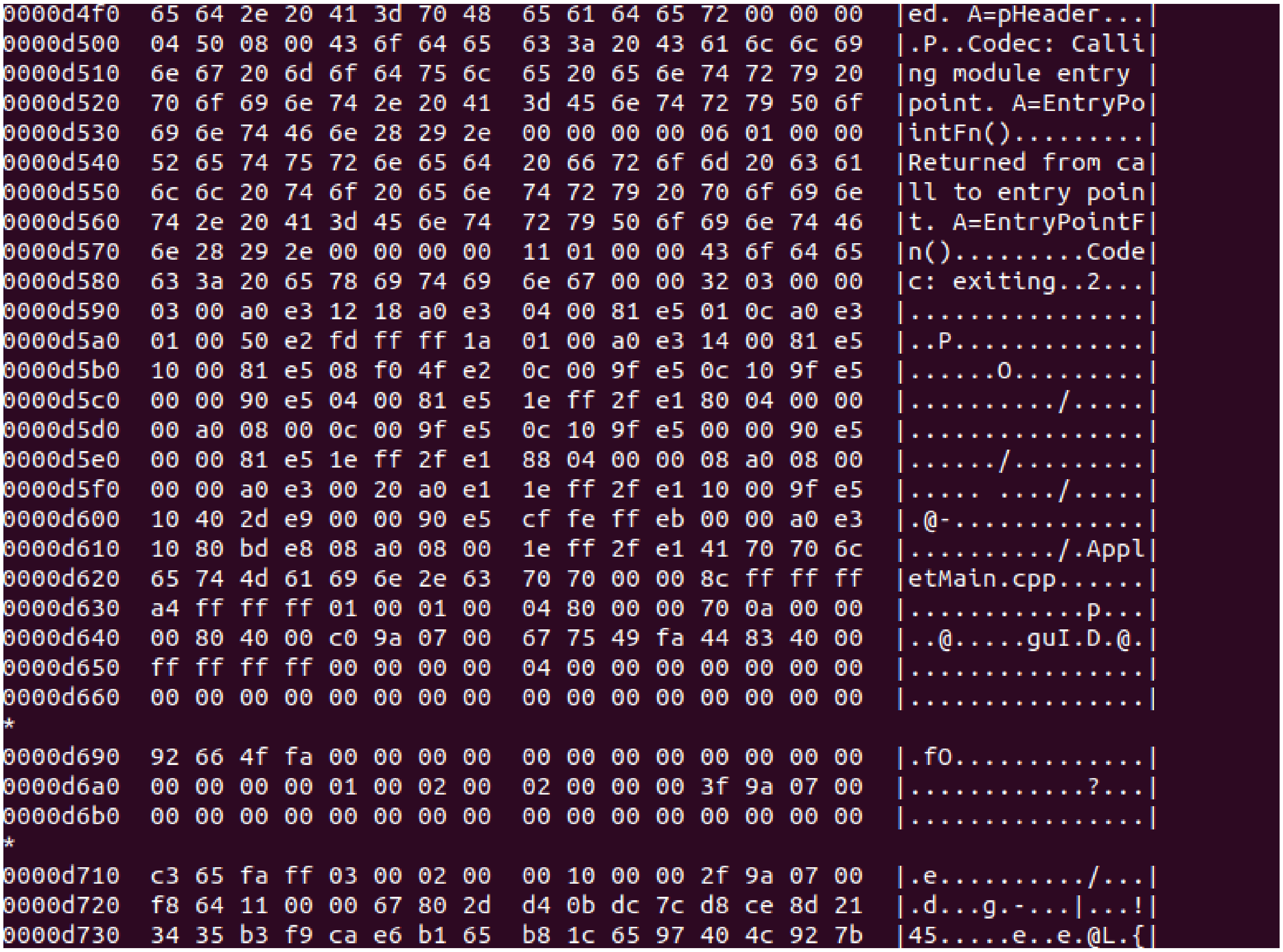
Now that we know the rough boundaries of some of the components of the firmware file, we can try to confirm these boundary offsets by looking at the actual contents of the firmware file around these areas.

If we run the firmware file through a hexdump, and look for lines that contain only an asterisk “*”, we can locate the compiler-added padding for each of the firmware components.



```
000003b0  00 00 00 00 00 00 00 00  00 00 00 00 7b 17 fe bb  |............{...|
000003c0  1e 00 00 ea 00 00 00 00  00 00 00 00 00 00 00 00  |................|
000003d0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
00000440  11 3f 19 ee 01 00 13 e3  00 00 00 0a 04 00 00 ea  |.?..............|
00000450  68 30 9f e5 11 3f 09 ee  81 da a0 e3 de 05 00 eb  |h0...?..........|
00000460  00 00 e0 e3 3c 30 8f e2  00 00 83 e5 f0 5f 2d e9  |....<0......._-.|
00000470  10 2f 10 ee 02 28 a0 e1  22 2a a0 e1 56 1b 00 e3  |./...(.."*..V...|
00000480  01 00 52 e1 00 00 00 1a  00 01 08 f1 10 50 8f e2
```

Contents of the firmware binary



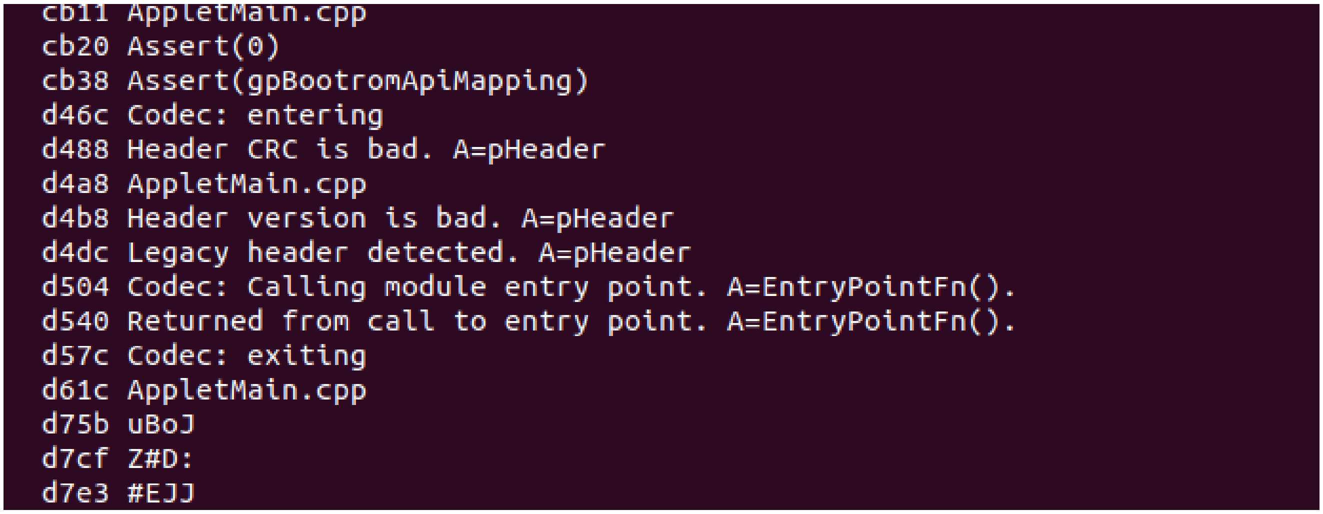
```
0000d4f0  65 64 2e 20 41 3d 70 48  65 61 64 65 72 00 00 00  |ed. A=pHeader...|
0000d500  04 50 08 00 43 6f 64 65  63 3a 20 43 61 6c 6c 69  |.P..Codec: Calli|
0000d510  6e 67 20 6d 6f 64 75 6c  65 20 65 6e 74 72 79 20  |ng module entry |
0000d520  70 6f 69 6e 74 2e 20 41  3d 45 6e 74 72 79 50 6f  |point. A=EntryPo|
0000d530  69 6e 74 46 6e 28 29 2e  00 00 00 00 06 01 00 00  |intFn().........|
0000d540  52 65 74 75 72 6e 65 64  20 66 72 6f 6d 20 63 61  |Returned from ca|
0000d550  6c 6c 20 74 6f 20 65 6e  74 72 79 20 70 6f 69 6e  |ll to entry poin|
0000d560  74 2e 20 41 3d 45 6e 74  72 79 50 6f 69 6e 74 46  |t. A=EntryPointF|
0000d570  6e 28 29 2e 00 00 00 00  11 01 00 00 43 6f 64 65  |n().........Code|
0000d580  63 3a 20 65 78 69 74 69  6e 67 00 00 32 03 00 00  |c: exiting..2...|
0000d590  03 00 a0 e3 12 18 a0 e3  04 00 81 e5 01 0c a0 e3  |................|
0000d5a0  01 00 50 e2 fd ff ff 1a  01 00 a0 e3 14 00 81 e5  |..P.............|
0000d5b0  10 00 81 e5 08 f0 4f e2  0c 00 9f e5 0c 10 9f e5  |......O.........|
0000d5c0  00 00 90 e5 04 00 81 e5  1e ff 2f e1 80 04 00 00  |........../.....|
0000d5d0  00 a0 08 00 0c 00 9f e5  0c 10 9f e5 00 00 90 e5  |................|
0000d5e0  00 00 81 e5 1e ff 2f e1  88 04 00 00 08 a0 08 00  |....../.........|
0000d5f0  00 00 a0 e3 00 20 a0 e1  1e ff 2f e1 10 00 9f e5  |..... ..../.....|
0000d600  10 40 2d e9 00 00 90 e5  cf fe ff eb 00 00 a0 e3  |.@-.............|
0000d610  10 80 bd e8 08 a0 08 00  1e ff 2f e1 41 70 70 6c  |........../.Appl|
0000d620  65 74 4d 61 69 6e 2e 63  70 70 00 00 8c ff ff ff  |etMain.cpp......|
0000d630  a4 ff ff ff 01 00 01 00  04 80 00 00 70 0a 00 00  |............p...|
0000d640  00 80 40 00 c0 9a 07 00  67 75 49 fa 44 83 40 00  |..@.....guI.D.@.|
0000d650  ff ff ff ff 00 00 00 00  04 00 00 00 00 00 00 00  |................|
0000d660  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
0000d690  92 66 4f fa 00 00 00 00  00 00 00 00 00 00 00 00  |.fO.............|
0000d6a0  00 00 00 00 01 00 02 00  02 00 00 00 3f 9a 07 00  |............?...|
0000d6b0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
0000d710  c3 65 fa ff 03 00 02 00  00 10 00 00 2f 9a 07 00  |.e........../...|
0000d720  f8 64 11 00 00 67 80 2d  d4 0b dc 7c d8 ce 8d 21  |.d...g.-...|...!|
0000d730  34 35 b3 f9 ca e6 b1 65  b8 1c 65 97 40 4c 92 7b  |45.....e..e.@L.{|
```

Contents of other parts of the firmware binary

The output of the Hexdump utility, together with the previous findings, confirm the section of the firmware binary containing ARM code. We previously suspected that this code belongs to the bootloader.

6. Strings –radix=x firmware.bin

Next, let’s extract the ASCII strings from the firmware together with their offsets.



```
cb11 AppletMain.cpp
cb20 Assert(0)
cb38 Assert(gpBootromApiMapping)
d46c Codec: entering
d488 Header CRC is bad. A=pHeader
d4a8 AppletMain.cpp
d4b8 Header version is bad. A=pHeader
d4dc Legacy header detected. A=pHeader
d504 Codec: Calling module entry point. A=EntryPointFn().
d540 Returned from call to entry point. A=EntryPointFn().
d57c Codec: exiting
d61c AppletMain.cpp
d75b uBoJ
d7cf Z#D:
d7e3 #EJJ
```

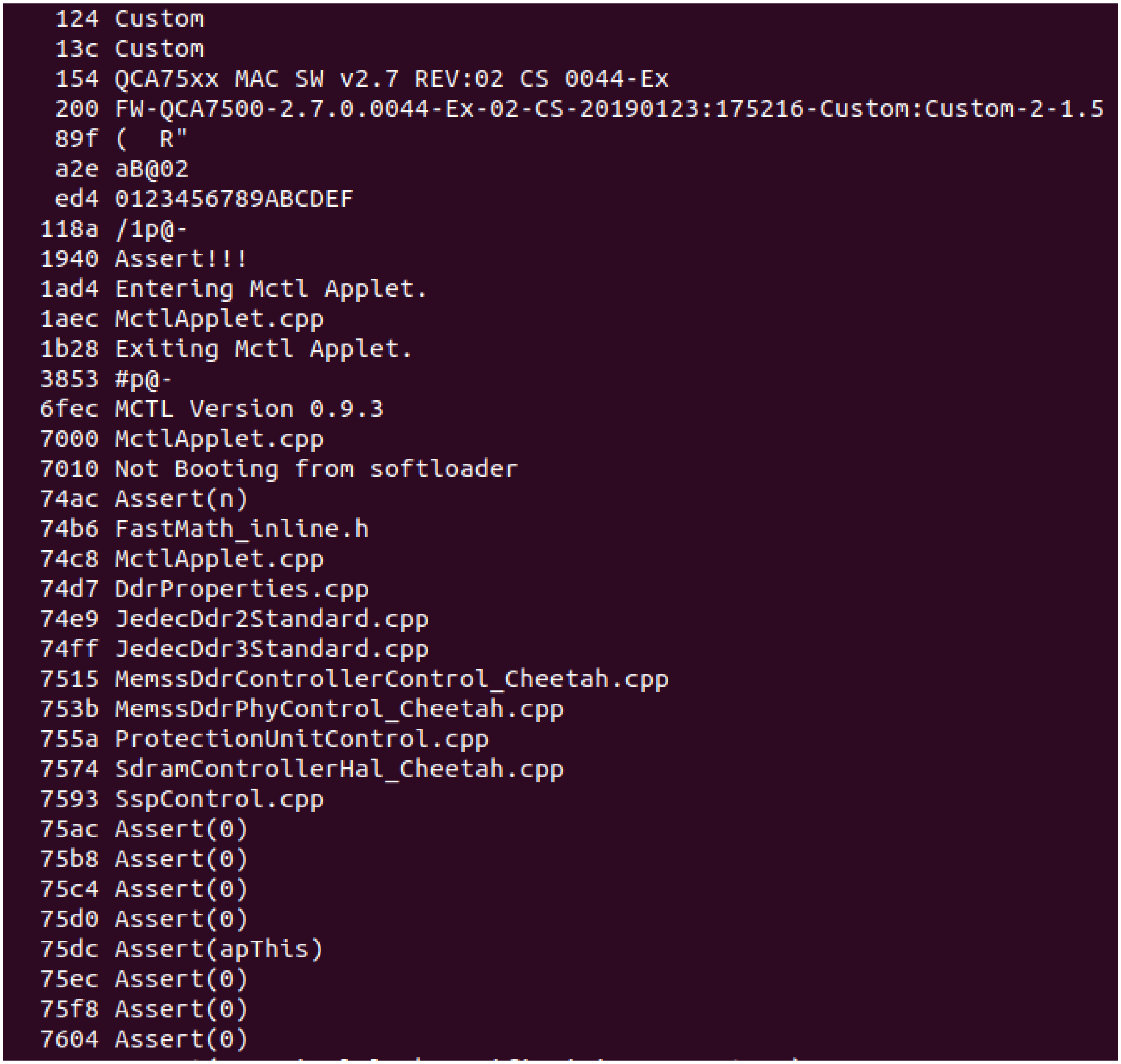
Last ASCII strings found in the firmware binary

Looking at the screenshot above, there are some strings related to the module entry point. These strings can give us a good indication of the nature of the code involved.

We can see some other interesting strings from the beginning of the firmware binary in the screenshot below. For example, the “MctlApplet.cpp” library name can be used to find other binaries or packages from the same developers. Having other firmware images from the same vendor helps to better understand the binary structure.

Another interesting string from the same screenshot is “Not Booting from softloader” which can indicate the process state or perhaps the nature of this module.

Strings containing “Assert()” can suggest different information about the code. Using Asserts is a common practice in firmware development, as it helps the developer to debug and troubleshoot the code during the development and production phase.



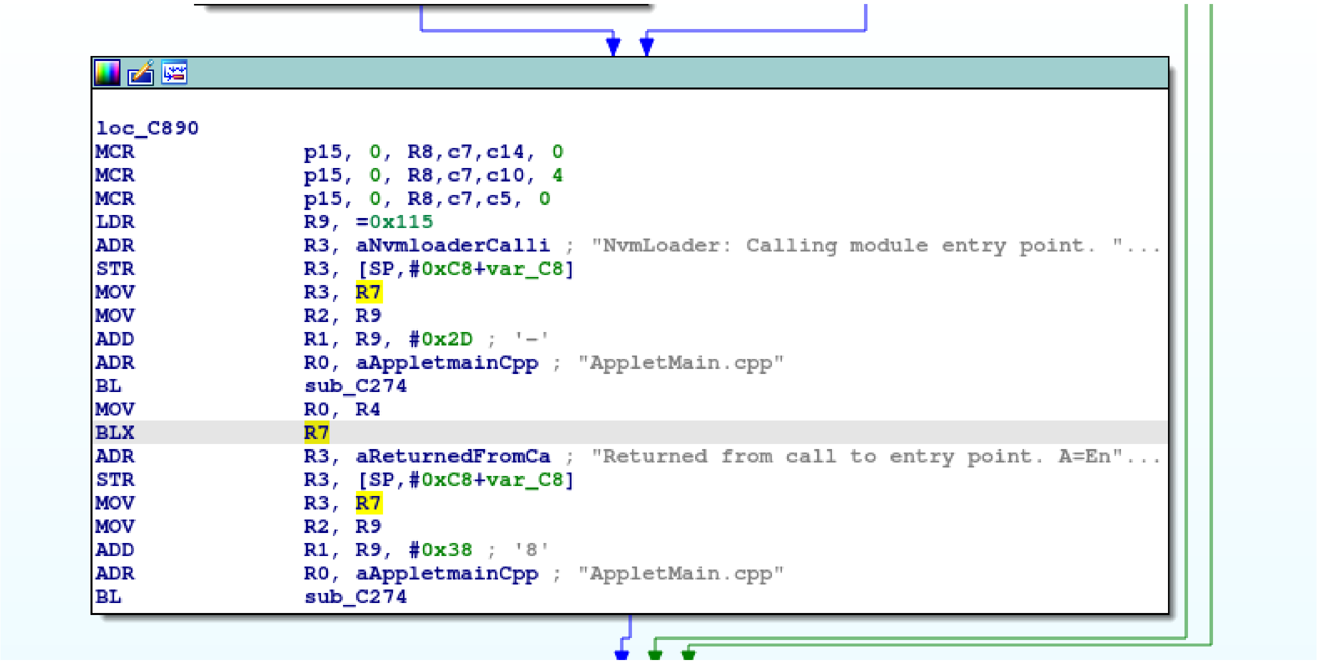
```
 124 Custom
 13c Custom
 154 QCA75xx MAC SW v2.7 REV:02 CS 0044-Ex
 200 FW-QCA7500-2.7.0.0044-Ex-02-CS-20190123:175216-Custom:Custom-2-1.5
 89f (  R"
 a2e aB@02
 ed4 0123456789ABCDEF
118a /1p@-
1940 Assert!!!
1ad4 Entering Mctl Applet.
1aec MctlApplet.cpp
1b28 Exiting Mctl Applet.
3853 #p@-
6fec MCTL Version 0.9.3
7000 MctlApplet.cpp
7010 Not Booting from softloader
74ac Assert(n)
74b6 FastMath_inline.h
74c8 MctlApplet.cpp
74d7 DdrProperties.cpp
74e9 JedecDdr2Standard.cpp
74ff JedecDdr3Standard.cpp
7515 MemssDdrControllerControl_Cheetah.cpp
753b MemssDdrPhyControl_Cheetah.cpp
755a ProtectionUnitControl.cpp
7574 SdramControllerHal_Cheetah.cpp
7593 SspControl.cpp
75ac Assert(0)
75b8 Assert(0)
75c4 Assert(0)
75d0 Assert(0)
75dc Assert(apThis)
75ec Assert(0)
75f8 Assert(0)
7604 Assert(0)
```

First ASCII strings found in the firmware binary

7. IDA -parm firmware.bin

We can see that we have already collected lots of valuable information from this firmware binary that seemed quite incomprehensible at the beginning.

Let’s now use IDA to inspect the code. As this binary is not an ELF file with standard headers that show the ISA, we need to explicitly tell IDA to use the ARM instruction set to disassemble the code.



```
loc_C890
MCR             p15, 0, R8,c7,c14, 0
MCR             p15, 0, R8,c7,c10, 4
MCR             p15, 0, R8,c7,c5, 0
LDR             R9, =0x115
ADR             R3, aNvmloaderCalli ; "NvmLoader: Calling module entry point. "...
STR             R3, [SP,#0xC8+var_C8]
MOV             R3, R7
MOV             R2, R9
ADD             R1, R9, #0x2D ; '-'
ADR             R0, aAppletmainCpp ; "AppletMain.cpp"
BL              sub_C274
MOV             R0, R4
BLX             R7
ADR             R3, aReturnedFromCa ; "Returned from call to entry point. A=En"...
STR             R3, [SP,#0xC8+var_C8]
MOV             R3, R7
MOV             R2, R9
ADD             R1, R9, #0x38 ; '8'
ADR             R0, aAppletmainCpp ; "AppletMain.cpp"
BL              sub_C274
```

Disassembly view of part of a function in IDA

The above screenshot from IDA shows how the strings found in the previous analysis steps can be used to help find the call to the entry point of the kernel module.

8. dd

We can now go ahead and extract the part of the firmware binary which our analysis found to be the bootloader module.

9. Qemu

After all the modules have been extracted from the firmware binary – the file system content, the kernel modules and other components – we can then use Qemu to run the binaries, and even emulate the files that were meant for a different architecture from our own machine, and start interacting with them.

Conclusion

The number of IoT devices is getting bigger and bigger every day. From industrial control systems, smart cities and cars to consumer-grade devices such as mobile phones, networking devices, personal assistants, smart watches and a large variety of smart home appliances.

IoT devices are derived from embedded systems that have been around for many years. The manufacture and development of software for embedded devices has always had different priorities from those of general-purpose computer systems due to the different nature of these devices. These priorities have been shaped by the limited and specific functions of the devices themselves, the limited capabilities and capacities of the underlying hardware as well as the inaccessibility of the developed code to subsequent alteration and modifications. However, IoT devices have significant differences to traditional embedded systems. Most IoT devices nowadays run on hardware that have similar capabilities to a general-purpose computer system.

As IoT devices become more prevalent, they are now accessing and controlling many aspects of our lives and day-to-day interactions. IoT devices can now potentially give malicious actors unprecedented opportunities to do harm. This highlights the importance of security in IoT devices and also shows the relevance of research around this topic. The good news is that there are many tools and techniques available to assist current and future research in this field. Acquiring a good understanding of the architecture of IoT devices, learning the language these devices speak and a good dose of determination and perseverance are what it takes to enter this research field.

This post has been written primarily to motivate individuals who want to start diving into IoT security research. You can reach out to us regarding this research at iot_firmware_research@kaspersky.com or via my twitter account, @Noushinshbb.

We'll be publishing more in the future! Stay tuned!

- Firmware
- Internet of Things
- Linux
- Malware

Authors

Expert Noushin Shabab

Looking for sophisticated malware in IoT devices

Your email address will not be published. Required fields are marked *