Embedded Devices Security and Firmware Reverse Engineering

BH13US Workshop

Jonas Zaddach *
FIRMWARE.RE
jonas@firmware.re

Andrei Costin †
FIRMWARE.RE
andrei@firmware.re

ABSTRACT

Embedded devices have become the *usual presence* in the network of (m)any household(s), SOHO, enterprise or critical infrastructure.

The preached Internet of Things promises to *gazillion*-uple their number and heterogeneity in the next few years.

However, embedded devices are becoming lately the *usual suspects* in security breaches and security advisories and thus become the *Achilles' heel* of one's overall infrastructure security.

An important aspect is that embedded devices run on what's commonly known as firmwares. To understand how to secure embedded devices, one needs to understand their firmware and how it works.

This workshop aims at presenting a quick-start at how to inspect firmwares and a hands-on presentation with exercises on real firmwares from a security analysis standpoint.

General Terms

Computer System Security, Network and Distributed System Security, Embedded Devices, Firmware, Security, Reverse Engineering

*PhD candidate on
"Development of novel binary analysis techniques for security applications" at
EURECOM, Sophia-Antipolis, Biot, France,
jonas.zaddach@eurecom.fr

†PhD candidate on
"Software security in embedded systems" at
EURECOM, Sophia-Antipolis, Biot, France,
andrei.costin@eurecom.fr

Keywords

embedded devices, firmware, security, reverse engineering, exploitation, vulnerabilities, backdoors, static analysis, binary analysis, firmware unpacking, firmware analysis, firmware modification

1. INTRODUCTION

In the world of ever increasing interconnection of computing, mobile and embedded devices, their security has become critical. The security of embedded devices and their firmwares is the new differentiator in the embedded market.

The security requirements and expectations for computing devices are being constantly raised as the world moves towards the Internet of Things. This is especially true for embedded devices and their software counterpart – firmwares – which is also their weakest point as shown below.

On another hand, embedded devices still have much less to offer in terms of firmware security at this point. We can see almost daily security advisories related to embedded devices, many of them related to critical computer or cyber-physical systems. It's not accidental that anecdotical evidence vehiculate the term `Embedded and Firmware Security - Back to The 90s!`. It both shows how easy it is to find vulnerabilities in embedded firmware, as well as how bad is the state of affairs in the firmware world from a security view-point.

With this whitepaper and workshop, we aim at presenting a quick-start at how to inspect and analyze firmwares, delivering hands-on presentation on real firmwares and compiling exercises from a security analysis standpoint. This, on another hand, should help speed-up the responsible disclosure and fixing of those dormant vulnerabilities.

This paper is organized as follows: we start with presentation of minimal required theory in Section 2; we continue with survey on previous work and state of the art in Section 3; we present most commont firmware formats, their challenges and how to handle their unpacking end-to-end in Section 4; we reinforce the presented knowledge with hands-on exercises and solutions in Section 5; we conclude in Section 6.

1.1 Workshop Outline

The workshop which supports this whitepaper, is organized according to the following outline:

- what are the embedded systems

- what are the firmwares
- what are the challenges with firmwares
- how to overcome those challenges
- typical firmware formats and contents
- what's firmware packing
- how to tackle unpacking problems elegantly
- typical firmware analysis process
- introduction to firmware analysis process automation
- introduction to firmware emulation
- challenges and ideas to overcome those
- some use case studies on real-world vulnerability findings
- hands-on exercises

2. LITTLE BIT OF THEORY

2.1 Definition of firmware

The term "firmware" has been coined by Ascher Opler in in a 1967 Datamation article. His definition of firmware as a glue microcode layer between the CPU instruction set and the actual hardware has since been superseded, and the IEEE Standard Glossary of Software Engineering Terminology, Std 610.12-1990, defines firmware today as follows:

> The combination of a hardware device and computer instructions and data that reside as read-only software on that device.
> Notes: (1) This term is sometimes used to refer only to the hardware device or only to the computer instructions or data, but these meanings are deprecated.
> Notes: (2) The confusion surrounding this term has led some to suggest that it be avoided altogether.

For the sake of simplicity we will deviate from this definition in that we call the set of all code running on the hardware's processor (machine code and virtual machine code) the firmware of this device.

2.2 Device Classes

Firmware-driven devices can be found virtually everywhere - Nowadays our cars are controlled by hundreds microcontrollers, washing machines are programmable, and of course industrial control is automated and can be controlled from a central console. Here is a non-exhaustive list of all the domains that use firmware-driven devices:

- Networking – Routers, Switches, NAS, VoIP phones
- Surveillance – Alarms, Cameras, CCTV, DVRs, NVRs
- Industry Automation – PLCs, Power Plants, Industrial Process Monitoring and Automation
- Home Automation – Sensoring, Smart Homes, Z-Waves, Philips Hue
- Whiteware – Washing Machine, Fridge, Dryer
- Entertainment gear – TV, DVRs, Receiver, Stereo, Game Console, MP3 Player, Camera, Mobile Phone, Toys
- Other Devices - Hard Drives, Printers
- Cars
- Medical Devices

2.3 Embedded devices hardware architectures

This section highlights the different architecture elements of an embedded system, ranging from the processor architecture to memories, and connections between peripherals of the embedded device as well as connections to other systems.

Processor Architectures.

Contrary to the relative uniformity of the PC market, embedded device's architectures are very diverse. In middle to upper-class market segments of processors offering features like memory virtualization and high clock rates, ARM processors are wide-spread, and Intel is trying to catch up with its ATOM line. MIPS processors can be found, too. In the lower-class market processor architectures like Atmel AVR, Intel 8051 and Motorola 6800/68000 power microcontrollers with small memories and lower clock frequencies. Apart from that, more exotic architectures like Ambarella, Axis CRIS and others can be found in some devices.

On-board buses.

The processor cores communicate with other design blocks or chips around them through a variety of interfaces: Most commonly, SPI (Serial Peripheral Interface), I2C (Inter-IC), Dallas 1-Wire and UART serial buses can be found, but more complex systems also use buses more common in PC architectures like PCI and PCI Express. Finally, ARM cores can be connected to peripheral IP blocks through the AMBA (Advanced Microcontroller Bus Architecture) interface.

Common communication lines.

The above-mentioned buses are mainly used as communication interface on the same board. For communications with Computers or other systems, additional interfaces might be used:

- Ethernet - RJ45
- RS485
- CAN/FlexRay
- Bluetooth
- WIFI
- Infrared
- Zigbee
- Other radios (ISM-Band, etc)
- GPRS/UMTS
- USB

Memory.

Different types of memory can be mapped directly into the address space of the embedded system:

- DRAM is a volatile memory that can be accessed read/write. Though it is quite fast, some processor cycles might be needed to access content, which is why caching is employed in some architectures to speed up DRAM access. The DRAM controller needs to be set up with the memory's timings before this type of memory can actually be used, which normally happens at very early stages in the bootloader.
- SRAM is a volatile memory that can be accessed read/write. It is very fast and can be read or written without or much less delay than DRAM, but it is quite expensive, which is why you fill find only small quantities of this memory (typically < 1Mb) in a device.
- ROM is a non-volatile memory that can only be read. Typically it is programmed in factory and contains startup code that is absolutely needed, for example a mask ROM bootloader.
- Memory-Mapped NOR Flash is a non-volatile memory that can be accessed read/write. Contrary to previous memories, reads can happen for any offset, but writes need to take place for a whole block, which is why they are mainly used to store bootcode.

Common Storage.

While the above-mentioned memories, with the exception of NOR flash, serve only as volatile storage, other options exist for permanent data storage:

- NAND Flash is typically connected through a bus like SPI to the CPU and behaves similar to NOR flash: Any byte offset can be read, but writes and deletes need to happen for a whole erase block. Since each block may only be written so often before it breaks, special file systems exist that try to balance the wear between cells.
- SD Card or any other common storage card (MMC, ...) can directly be used as a block device in Linux. The connection of the controller to the main system varies (USB, integrated, SPI, ...).
- Hard Drive can be connected via SATA, SCSI or PATA to the system. Like for SD Cards, the actual connection of the controller to the system can be realized over other buses.

Common Operating Systems.

Embedded systems are powered by firmwares of varying complexities. More complex ones usually use a full-blown operating system like Linux or Windows NT. Less complex devices use operating systems like VxWorks or Windows CE, and lots of special purpose operating systems can be found, too. Here is a non-exhaustive list of operating systems that can be encountered in firmware analysis:

- Linux is by far the most popular operating system for more complex embedded devices.
- VxWorks is a popular proprietary real-time operating system.
- Cisco IOS
- Windows CE/NT
- L4
- eCos
- DOS
- Symbian
- JunOS
- Ambarella
- etc.

Common Bootloaders.

The bootloader is the first piece of software that is executed after a possible mask ROM bootloader. Its purpose is to load parts of the operating system into memory and bring the system in a defined state for the kernel (though this requirement is fluid, the Linux kernel takes over some of the former duties of a bootloader, like setting up the Pin Mux). It can be organized in one or two stages. In a two-stage setting, the first stage only knows how to load the second stage, while the second stage provides support for file systems etc.

- U-Boot is probably the most popular bootloader
- RedBoot
- BareBox
- Ubicom bootloader

Common Libraries and Dev Envs.

Today, there are several prepackaged toolchains available. These consist of build tools for the specific processor (compiler, assembler, etc). In most cases you will also get the standard library compiled for your target, and for some even a wide range of open-source packages, like openembedded's toolchain with its recipes.

- busybox + uClibc is probably the most used combination.
- buildroot
- openembedded
- crosstool
- crossdev

3. RELATED WORK AND STATE OF THE ART

In [12] the assessment of the security of current embedded management interfaces was conducted. Vulnerabilities were found in all 21 devices from 16 different brands, representing 8 different categories, including network switches, cameras, photo frames, and lights-out management modules. Along these, a new class of vulnerabilities was discovered, namely *cross-channel scripting (XCS)* [13]. XCS vulnerabilities are not particular to embedded devices, however it is indicated that embedded devices is the most affected population.

Results from [12] were subsequently used in [23]. Researchers address the challenge of building secure embedded web interfaces by proposing *WebDroid*, the first framework specifically dedicated to this purpose. To that end, they demonstrate and evaluate the efficiency of their framework in terms of performance and security.

In [14] RevNIC is presented. RevNIC is a tool for reverse engineering network drivers. The work presents a technique that helps automate the reverse engineering of device drivers. It takes a closed-source binary driver, automatically reverse engineers the driver's logic, and synthesizes new device driver code that implements the exact same hardware protocol as the original driver. This code can be targeted at the same or a different OS. No vendor documentation or source code is required.

Continuing in direction of [14], the works of [20–22] present on multiple aspects of firmware reversing and backdooring on the network cards.

[26] presents a time-of-check-to-time-of-use (TOCTTOU) attack via externally attached mass-storage devices. The attack is based on emulating a mass-storage device to observe and alter file access from the consumer device. The TOCTTOU attack was executed by providing different file content to the check and installation code of the target device, respectively. The presented attack shown to be effective to bypasses the file content inspection, resulting in the execution of rogue code on the device.

[18] presents the results of the study of a vulnerability assessment of embedded network devices within the worldâĂŹs largest ISPs and civilian networks, spanning North America, Europe and Asia. The observed data confirmed the intuition that these devices are indeed vulnerable to trivial attacks and that such devices can be found throughout the world in large numbers. This study was subsequently extended with works of [19] with a quantitative lower bound estimation on the number of vulnerable embedded device on a global scale.

Work of [16] presented the reverse-engineering of firmware images for multiple Xerox devices. This allowed discovery of lower-level APIs from the PostScript high-level document language. The attacks were delivered to the printers via standard printed documents, as previously demonstrated in [15]. Multiple attacks were presented, including memory dumping/scraping leading to password theft and passive network topology discovery, as well as outbound socket sending arbitrary data.

There were recent advances in firmware modification attacks by [10, 11, 17]. [11] addressed the network card based on Broadcom BCM4325 & BCM4329 chipsets and demonstrated how to put these cards in monitor mode. [17] presented a case study of the HP-RFU (Remote Firmware Update) LaserJet printer firmware modification vulnerability, which allows arbitrary injection of malware into the printerś firmware. While [15] demonstrated the proof-of-concept sending arbitrary or custom command to *any* printer via standard printed documents, including MS Office Word. Adobe PostScript and Java Applets-generated, [17] used the same attack vector to deliver a modified firmware. [10] examines the vulnerability of PLCs to intentional firmware modifications in order to obtain a better understanding of the threats posed by PLC firmware modification attacks and the feasibility of these attacks. A general firmware analysis methodology is presented, and a proof-of-concept experiment is used to demonstrate how legitimate firmware can be updated and uploaded to an Allen-Bradley ControlLogix L61 PLC.

On the deffensive side, however, there is slightly less previous work available.

In [24] addresses the important challenge of verifying the integrity of peripherals' firmware. Authors propose software-only attestation protocols to verify the integrity of peripherals' firmware, and show that they can detect all known software-based attacks. Authors also implement their scheme using a Netgear GA620 network adapter in an x86 PC, and evaluate theirs system with known attacks.

[25] presents a tool developed specifically for the SCADA environment to verify PLC firmware. The tool does not require any modifications to the SCADA system and can be implemented on a variety of systems and platforms. The tool captures serial data during firmware uploads and then verifies them against a known good firmware baseline. Attempts to inject modified and/or malicious firmware are identified by the tool.

3.1 Community Efforts and Tools

There are many community efforts dedicated to reverse engineering of firmwares and embedded devices. Each of these efforts have a specific goal and thus the tools produces by those efforts are influenced by their main goals.

We try to summarize in a comprehensive list the most used and visible efforts to date.

- binwalk – Binwalk is a firmware analysis tool designed to assist in the analysis, extraction, and reverse engineering of firmware images and other binary blobs. It is simple to use, fully scriptable, and can be easily extended via custom signatures, extraction rules, and plugin modules.

- firmware-mod-kit – This kit is a collection of scripts and utilities to extract and rebuild linux based firmware images. This kit allows for easy deconstruction and reconstruction of firmware images for various embedded devices.

- FRAK: Firmware Reverse Analysis Konsole – Unfortunately, after an year since BH12US and notes of FOSS license in [17], FRAK tool and it's source code remained unreleased as the site welcomes with the same message for an year: `SVN Repository: Coming soon! Please subscribe to mailing list for release date`. As of time of this writing, it was not possible to evaluate the tool, hence it was not possible to conclude over the state of this project.

- ERESI framework – The ERESI Reverse Engineering Software Interface is a multi-architecture binary analysis framework with a domain-specific language tailored to reverse engineering and program manipulation.

- signsrch – Tool for searching signatures inside files, extremely useful as help in reversing jobs like figuring or having an initial idea of what encryption/compression algorithm is used for a proprietary protocol or file. It can recognize tons of compression, multimedia and encryption algorithms and many other things like known strings and anti-debugging code which can be also manually added since it's all based on a text signature file read at runtime and easy to modify.
- hash-identifier – Software to identify the different types of hashes used to encrypt data and especially passwords. Over few dozens of hash supported. Encryption algorithms that can not be differentiated unless they have been decrypted, so the efficiency of the software also depends on the user's criteria.
- offzip – A very useful tool to unpack the zip (zlib, gzip, deflate, etc.) data contained in any type of file included raw files, packets, zip archives, executables and everything else. It's needed only to specify the offset where the zip data starts or using the useful -S search options able to find any possible zip block contained in the provided file. There are also other options for extracting all the zip blocks which have been found or dumping them as in their original compressed form.
- TrID – TrID is an utility designed to identify file types from their binary signatures. While there are similar utilities with hard coded logic, TrID has no fixed rules. Instead, it's extensible and can be trained to recognize new formats in a fast and automatic way.
- gpltool/bat – BAT, previously GPLtool, makes it easier and cheaper to look inside binary code, find compliance issues, and reduce uncertainty when deploying Free and Open Source Software.
- PFS – PFS archive file format (file system?) is used in images of routers like Benq ESG 103, NDC NWH8018, and probably many others.
- CNU_fpu – CNU_fpu is a pack/unpack utility for Cisco IP Phones firmware files (7941, 7961, 7911-12 and others based on CNU_File_Archive_3.0 format) Written by kbdfck at virtualab.ru
- ardrone-tool – Aims to develop tools for A.R. Drone, for example to create and flash custom linux kernels.
- UnYAFFS – Unyaffs is a program to extract files from a yaffs file system image. Now it can only extract images created by mkyaffs2image.
- squash-tools – SquashFS
- UbiFS – UbiFS

4. FIRMWARE FORMATS AND UNPACKING EXPLAINED END-TO-END

In this section, we are going to look at the various archive and filesystem image formats that can be encountered when inspecting a packed firmware image. Depending on the firmware complexity, you will find different levels of packing and different objects inside the archives. We classify the firmware complexity according to this categories:

- Full-blown (full-OS/kernel + bootloader + libs + apps) - This is typically a Linux or Windows firmware that carries a complete file system. The driving application will most likely run in User mode, though custom kernel modules/drivers might be used.
- Integrated (apps + OS-as-a-lib) - This is firmware with a small proprietary operating system or none at all - the application will typically run with the same privileges as the kernel.
- Partial updates (apps or libs or resources or support) The firmware image will not contain all files that form the complete system, but just an update for concerned files.

In a firmware of the first category, you will typicallyl find the following objects while you unpack the firmware:

- Bootloader (1st/2nd stage)
- Kernel
- File-system images
- User-land binaries
- Resources and support files
- Web-server/web-interface

Those objects can be grouped and packed in any of the following archives or filesystem images (non-exhaustive list):

- Pure archives (CPIO/Ar/Tar/GZip/BZip/LZxxx/RPM)
- Pure filesystems (YAFFS, JFFS2, extNfs)
- Pure binary formats (SREC, iHEX, ELF)
- Hybrids (any breed of above)

In this following paragraph, we list unpacking tools for each archive format:

Firmware Formats – Flavors.

- Ar - The ar tool is part of all Linux/FreeBSD distributions. Use "ar x <file>" to extract.
- YAFFS2 - There are tools in the yaffs2utils project to extract this filesystem [9].
- JFFS2 - BAT uses a python wrapper around the jffs2dump utility that is part of mtdtools.
- SquashFS - You can find unpacking tools at [7]
- CramFS - The firmware-mod-kit provides a tool called "uncramfs" to extract files [3].
- ROMFS - Harald Welte has developed a tool called "romfschk" that can extract files [5].
- UbiFS - Unfortunately no easy way to extract - see [8].
- xFAT - Mount as loopback device in Linux.

- NTFS - Mount as loopback device in Linux.
- ext2fs/ext3fs/ext4fs - Mount as loopback device in Linux
- iHEX - Convert to elf or binary by doing

  ```
  objcopy -I ihex -O elf32-little <input> <output>
  objcopy -I ihex-O binary <input> <output>
  ```

- SREC/S19 - Convert to elf or binary by doing

  ```
  objcopy -I srec -O elf32-little <input> <output>
  objcopy -I srec-O binary <input> <output>
  ```

- PJL
- CPIO/Ar/Tar/GZip/BZip/LZxxx/RPM - Your favorite Linux distribution should provide tools to handle these archive formats.

5. EXERCISES AND SOLUTIONS

5.1 Reversing a Seagate HDD's firmware file format

In this excercise, we want to inspect a firmware for an embedded system that does not have a known operating system, nor a known firmware file format.

The first step is to obtain the firmware for the MooseDT MX1A-3D4D-DMax22 from the Seagate website [6]. Then, the obtained file needs to be unpacked until the actual firmware file is found within.

Unpacking the firmware.

A quite stupid and boring mechanic task:

```
$ 7z x MooseDT-MX1A-3D4D-DMax22.iso -oimage
$ cd image
$ ls
[BOOT]  DriveDetect.exe  FreeDOS  README.txt
$ cd \[BOOT\]/
$ ls
Bootable_1.44M.img
$ file Bootable_1.44M.img
Bootable_1.44M.img: DOS floppy 1440k,
x86 hard disk boot sector
$ mount -o loop Bootable_1.44M.img /mnt
$ mkdir disk
$ cp -r /mnt/* disk/
$ cd disk
$ ls
AUTOEXEC.BAT  COMMAND.COM  CONFIG.SYS  HIMEM.EXE
KERNEL.SYS  MX1A3D4D.ZIP  RDISK.EXE  TDSK.EXE
unzip.exe
$ mkdir archive
$ cd archive
$ unzip ../MX1A3D4D.ZIP
$ ls
6_8hmx1a.txs  CHOICE.EXE  FDAPM.COM  fdl464.exe
flash.bat  LIST.COM  MX1A4d.lod  README.TXT
seaenum.exe
$ file *
6_8hmx1a.txs: ASCII text, with CRLF line terminators
CHOICE.EXE:   MS-DOS executable, MZ for MS-DOS
FDAPM.COM:    FREE-DOS executable (COM), UPX compressed
fdl464.exe:   MS-DOS executable, COFF for MS-DOS,
              DJGPP go32 DOS extender, UPX compressed
flash.bat:    DOS batch file, ASCII text, with CRLF
              line terminators
LIST.COM:     DOS executable (COM)
MX1A4d.lod:   data
README.TXT:   ASCII English text, with CRLF line
              terminators
seaenum.exe:  MS-DOS executable, COFF for MS-DOS,
              DJGPP go32 DOS extender, UPX compressed
$ less flash.bat
set exe=fdl464.exe
set family=Moose
set model1=MAXTOR STM3750330AS
set model2=MAXTOR STM31000340AS
rem set model3=
rem set firmware=MX1A4d.lodd
set cfgfile=6_8hmx1a.txs
set options=-s -x -b -v -a 20
...
:SEAFLASH1
%exe% -m %family% %options% -h %cfgfile%
if errorlevel 2 goto WRONGMODEL1
if errorlevel 1 goto ERROR
goto DONE
```

Unpacking the firmware (Summary).

- We have unpacked the various wrappers, layers, archives and filesystems of the firmware
  - ISO → DOS IMG → ZIP → LOD
- The firmware is flashed on the HDD in a DOS environment (FreeDOS)
- The update is run by executing a DOS batch file (flash.bat)
- There are
  - a firmware flash tool (fdl464.exe)
  - a configuration for that tool (6_8hmx1a.txs, encrypted or obfuscated/encoded)
  - the actual firmware (MX1A4d.lod)
- The firmware file is not in a binary format known to `file` and `magic` tools

→ Let's have a look at the firmware file!

Inspecting the firmware file: hexdump.

```
$ hexdump -C MX1A4d.lod
00000000  00 00 00 00 00 00 00 00  00 00 00 00 00 00 07 00  |................|
00000010  80 01 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00000020  00 00 00 00 00 22 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 79 dc  |..............y.|
00000040  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
000001c0  0e 10 14 13 02 00 03 10  00 00 00 00 ff 10 41 00  |..............A.|
000001d0  00 20 00 00 ad 03 2d 00  13 11 15 16 11 13 07 20  |. ....-........ |
000001e0  00 00 00 00 40 20 00 00  00 00 00 00 00 00 00 00  |....@ ..........|
000001f0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 3f 1d  |..............?.|
00000200  00 c0 49 00 00 00 2d 00  10 b5 27 48 40 68 41 42  |..I...-...'H@hAB|
00000210  26 48 00 f0 78 ee 10 bd  10 b5 04 1c ff f7 f4 ff  |&H..x...........|
00000220  a0 42 03 d2 22 49 40 18  00 1b 10 bd 00 1b 10 bd  |.B.."I@.........|
00000230  1d 48 40 68 40 42 70 47  10 b5 01 1c ff f7 f8 ff  |.H@h@BpG........|
00000240  41 1a 0f 20 00 f0 5e ee  10 bd 7c b5 04 1c 20 1c  |A.. ..^...|... .|
00000250  00 21 00 90 17 a0 01 91  0c c8 00 98 00 f0 f2 ed  |.!..............|
00000260  01 da 00 f0 ed ff ff f7  cf ff 05 1c 28 1c ff f7  |............(...|
00000270  d3 ff a0 42 fa d3 7c bd  7c b5 04 1c 20 01 00 1b  |...B..|.|... ...|
00000280  00 21 00 90 0b a0 01 91  0c c8 00 98 00 f0 da ed  |.!..............|
...
```

→ The header did not look familiar to me :(

Inspecting the firmware file: strings.

```
$ strings MX1A4d.lod
...
XlatePhySec, h[Sec],[NumSecs]
XlatePhySec, p[Sec],[NumSecs]
XlatePlpChs, d[Cyl],[Hd],[Sec],[NumSecs]
XlatePlpChw, f[Cyl],[Hd],[Wdg],[NumWdgs]
XlateSfi, D[PhyCyl],[Hd],[Sfi],[NumSfis]
```

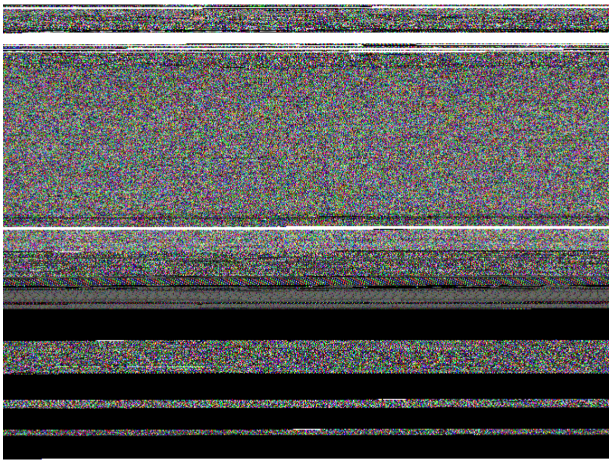

Figure 1: Bin2bmp output for the firmware file

```
XlateWedge, t[Wdg],[NumWdgs]
ChannelTemperatureAdj, U[TweakTemperature],[Partition],[Hd],[Zone],[Opts]
WrChs, W[Sec],[NumSecs],,[PhyOpt],[Opts]
EnableDisableWrFault, u[Op]
WrLba, W[Lba],[NumLbas],,[Opts]
WrLongOrSystemChs, w[LongSec],[LongSecsOrSysSec],[SysSecs],[LongPhySecOpt],,[SysOpts]
RwPowerAsicReg, V[RegAddr],[RegValue],[WrOpt]
WrPeripheralReg, s[OpType],[RegAddr],[RegValue],[RegMask],[RegPagAddr]
WrPeripheralReg, t[OpType],[RegAddr],[RegValue],[RegMask],[RegPagAddr]
...
```

→ Strings are visible, meaning the program is neither encrypted nor compressed
→ We actually know these strings ... they are from the diagnostic menu's help!

Inspecting the firmware file: binwalk.

```
$ binwalk MX1A4d.lod

DECIMAL         HEX             DESCRIPTION
-------------------------------------------------------------------------
499792          0x7A050         Zip archive data, compressed size: 48028,
                                uncompressed size: 785886, name: ""

$ dd if=MX1A4d.lod of=/tmp/bla.bin bs=1 skip=499792
$ unzip -l /tmp/bla.bin
Archive:  /tmp/bla.bin
  End-of-central-directory signature not found.  Either this file is not
  a zipfile, or it constitutes one disk of a multi-part archive.  In the
  latter case the central directory and zipfile comment will be found on
  the last disk(s) of this archive.
unzip:  cannot find zipfile directory in one of /tmp/bla.bin or
        /tmp/bla.bin.zip, and cannot find /tmp/bla.bin.ZIP, period.
```

→ binwalk does not know this firmware, the contained archive was apparently a false positive.

Inspecting the firmware file: Visualization.

To spot different sections in a binary file, a visual representation can be helpful.

- HexWorkshop is a commercial program for Windows. Most complete featureset (Hex editor, visualisation, ...) [4]
- Binvis is a project on google code for different binary visualisation methods. Visualisation is ok, but the program seems unfinished. [2].
- Bin2bmp is a very simple python script that computes a bitmap from your binary [1].

You can see the output of bin2bmp in figure 5.1. The output of the other tools is very similar to this plot. You can see that there are some clearly separate sections in the file, for example the shorter section in the beginning, separated by a sequence of 0xFF-bytes (white) from the next huge block, then another short separation and another block that shows a more regular pattern at its end. Finally there are three sections of different sizes in the end of the file, separated by 0x00-bytes (black).

Identifying the CPU instruction set.

- **ARM:** Look out for bytes in the form of 0xeX that occur every 4th byte. The highest nibble of the instruction word in ARM is the condition field, whose value 0xe means AL, execute this instruction unconditionally. The instruction space is populated sparsely, so a disassembly will quickly end in an invalid instruction or lots of conditional instructions.
- **Thumb:** Look out for words with the pattern 0xF000F000 (bl/blx), 0xB500BD00 ("pop XXX, pc" followed by "push XXX, lr"), 0x4770 (bx lr). The Thumb instruction set is much denser than the ARM instruction set, so a disassembly will go for a long time before hitting an invalid instruction.

In general, you should either know the processor already from the reconnaissance phase, or you try to disassemble parts of the file with a disassembler for the processor you suspect the code was compiled for. In the visual representation, executable code should be mostly colorful (dense instruction sets) or display patterns (sparse instruction sets).

In our firmware, searching for "e?" in the hexdump leads us to:

```
00002420  04 e0 4e e2 00 40 2d e9  00 e0 4f e1 00 50 2d e9  |..N..@-...O..P-.|
00002430  db f0 21 e3 8f 5f 2d e9  18 10 9f e5 00 00 91 e5  |..!.._-.........|
00002440  30 ff 2f e1 8f 5f bd e8  d1 f0 21 e3 00 50 bd e8  |0./.._....!..P..|
00002450  0e f0 69 e1 00 80 fd e8  44 00 00 00 08 20 fe 01  |..i.....D.... ..|
00002460  94 00 00 00 00 30 a0 e1  0c ce 9f e5 01 00 a0 e1  |.....0..........|
00002470  10 40 2d e9 14 10 93 e5  be c3 dc e1 d0 10 d1 e1  |.@-.............|
00002480  08 e0 93 e5 02 20 8c e0  92 01 01 e0 20 c0 e0 e3  |..... ...... ...|
00002490  81 22 61 e0 01 25 62 e0  42 29 a0 e1 82 0c 62 e1  |."a..%b.B)....b.|
000024a0  d8 cd 9f e5 82 11 81 e0  c6 20 51 e2 42 20 81 42  |......... Q.B .B|
000024b0  81 10 8c e0 f0 10 d1 e1  82 20 8c e0 04 c0 93 e5  |......... ......|
000024c0  f0 20 d2 e1 ac 01 2c e1  8e c2 2c e1 00 c0 83 e5  |. ....,...,.....|
000024d0  ac cd 9f e5 fc c9 dc e1  00 00 5c e3 10 40 bd a8  |..........\..@..|
000024e0  8e 1a 04 aa 10 80 bd e8  f0 41 2d e9 94 7d 9f e5  |.........A-..}..|
000024f0  80 40 a0 e1 07 00 54 e3  00 50 a0 e1 f7 6f 47 e2  |.@....T..P...oG.|
```

Let's verify that this is indeed ARM code ...

```
$ dd if=MX1A4d.lod bs=1 skip=$(( 0x2420 )) > /tmp/bla.bin
$ arm-none-eabi-objdump -b binary -m arm -D /tmp/bla.bin

/tmp/bla.bin:     file format binary


Disassembly of section .data:

00000000 <.data>:
       0:       e24ee004        sub     lr, lr, #4
       4:       e92d4000        stmfd   sp!, lr
       8:       e14fe000        mrs     lr, SPSR
       c:       e92d5000        push    ip, lr
      10:       e321f0db        msr     CPSR_c, #219    ; 0xdb
      14:       e92d5f8f        push    r0, r1, r2, r3, r7, r8, r9, sl, fp, ip, lr
      18:       e59f1018        ldr     r1, [pc, #24]   ; 0x38
      1c:       e5910000        ldr     r0, [r1]
      20:       e12fff30        blx     r0
      24:       e8bd5f8f        pop     r0, r1, r2, r3, r7, r8, r9, sl, fp, ip, lr
      28:       e321f0d1        msr     CPSR_c, #209    ; 0xd1
      2c:       e8bd5000        pop     ip, lr
      30:       e169f00e        msr     SPSR_fc, lr
      34:       e8fd8000        ldm     sp!, pc^
      38:       00000044        andeq   r0, r0, r4, asr #32
      3c:       01fe2008        mvnseq  r2, r8
      40:       00000094        muleq   r0, r4, r0
      44:       e1a03000        mov     r3, r0
      48:       e59fce0c        ldr     ip, [pc, #3596] ; 0xe5c
```

→ Looks good!

Navigating the firmware.

In this paragraph, we look at several starting points for a more in-depth analysis of the firmware contained in the firmware file. The first method is to look for the stack setup that has to happen for each ARM processor mode before a system can actually call functions. This typically happens in a sequence of "msr CPSR_c, XXX" instructions, which switch the CPU mode, and assignments to the stack pointer. The msr instruction exists only in ARM mode (not true for Thumb2 any more ... :( ) Very close you should also find some coprocessor initializations (mrc/mcr).

```
18a2c:      e3a000d7     mov    r0, #215         ; 0xd7
18a30:      e121f000     msr    CPSR_c, r0
18a34:      e59fd0cc     ldr    sp, [pc, #204]   ; 0x18b08
18a38:      e3a000d3     mov    r0, #211         ; 0xd3
18a3c:      e121f000     msr    CPSR_c, r0
18a40:      e59fd0c4     ldr    sp, [pc, #196]   ; 0x18b0c
18a44:      ee071f9a     mcr    15, 0, r1, cr7, cr10, 4
18a48:      e3a00806     mov    r0, #393216      ; 0x60000
18a4c:      ee3f1f11     mrc    15, 1, r1, cr15, cr1, 0
18a50:      e1801001     orr    r1, r0, r1
18a54:      ee2f1f11     mcr    15, 1, r1, cr15, cr1, 0
```

A second method is to find the exception handler table that contains the branches to the actual exception handlers. This piece of firmware is important as it reveals information about the address spaces from where code is run. In the ARMv5 architecture, exceptions are handled by ARM instructions in a table at address 0. Normally these have the form "ldr pc, XXX" and load the program counter with a value stored relative to the current program counter (i.e. in a table from address 0x20 on).

→ The exception vectors give an idea of which addresses are used by the firmware.

```
arm-none-eabi-objdump -b binary -m arm -D MX1A4d.lod \
  | grep -E 'ldr\s+pc' | less
```

→ We get the following output from `arm-none-eabi-objdump`

```
220e4:      e59ff018     ldr    pc, [pc, #24]    ; 0x22104
220e8:      e59ff018     ldr    pc, [pc, #24]    ; 0x22108
220ec:      e59ff018     ldr    pc, [pc, #24]    ; 0x2210c
220f0:      e59ff018     ldr    pc, [pc, #24]    ; 0x22110
220f4:      e59ff018     ldr    pc, [pc, #24]    ; 0x22114
220f8:      e1a00000     nop                     ; (mov r0, r0)
220fc:      e59ff018     ldr    pc, [pc, #24]    ; 0x2211c
22100:      e59ff018     ldr    pc, [pc, #24]    ; 0x22120
22104:      0000a824     andeq  sl, r0, r4, lsr #16
22108:      0000a8a4     andeq  sl, r0, r4, lsr #17
2210c:      0000a828     andeq  sl, r0, r8, lsr #16
22110:      0000a7ec     andeq  sl, r0, ip, ror #15
22114:      0000a44c     andeq  sl, r0, ip, asr #8
22118:      00000000     andeq  r0, r0, r0
2211c:      0000a6ac     andeq  sl, r0, ip, lsr #13
22120:      00000058     andeq  r0, r0, r8, asr r0
```

5.2 Exploring the Firmware of Vicon IPCAM 960 series

Downloading and Unpacking.

- Getting 51110.2.1800.96.bin
- Unpacking 51110.2.1800.96.bin
  - `$VICON_JFFS2` is the unpacked JFFS2 image inside 51110.2.1800.96.bin
- Exploring 51110.2.1800.96.bin web-interface
  - `$VICON_JFFS2/etc/lighttpd/lighttpd.conf`
  - `$VICON_JFFS2/mnt/www.nf`

Reconnaissance.

Web-interface of 51110.2.1800.96.bin

- first, quick-explore the web-interface
- lighttpd-based
  - `sudo apt-get install lighttpd php5-cgi`
  - `sudo lighty-enable-mod fastcgi`
  - `sudo lighty-enable-mod fastcgi-php`
  - `sudo service lighttpd force-reload`
- then, we want to emulate the web-interface on a PC
  - requires tweaking `$VICON_JFFS2/etc/lighttpd/lighttpd.`
  - requires some minor development and fixes

Tweaking.

Tweaking $VICON_JFFS2/etc/lighttpd/lighttpd.conf

- correct `document-root`
- replace `/mnt/www.nf` with `$VICON_JFFS2/mnt/www.nf`
- set `port` to 1337
- set `errorlog` and `accesslog`
- create plain basic-auth password file
- set `auth.backend.plain.userfile`
- replace all .fcgi files with a generic `action.bottle.fcgi.py`
- enable `.py` as `FastCGI` in `$VICON_JFFS2/etc/lighttpd/lighttpc`

Developing.

Writing a stub `action.bottle.fcgi.py`

- `sudo apt-get install python-pip python-setuptools`
- `sudo pip install bottle`

Fuzz/pentest/debug.

Running and debugging web-interface of 51110.2.1800.96.bin

- iterative-fixing approach
- `sudo lighttpd -D -f $VICON_JFFS2/etc/lighttpd/lighttpd.`
- check lighttpd logs for startup errors
- check Firefox web-developer console for client/server errors
  - console shows we need to define `INFO_SWVER` inside `info.js`
  - start from above by restarting lighttpd

6. CONCLUSIONS

We have presented on embedded devices and their underlying firmware from a security point of view. We surveyed the related work and existing state of the art. Along the way, we introduced most commong firmware formats, challenges they bring to the game, as well as the tools and techniques to unpack and analyze them. We also introduce on the topic of firmware emulation for easier and faster vulnerability discovery. We finish the presentation with hands-on workshop exercises and solutions to help the audience to better understand the presented material.

We conclude with the following:

- though firmware reverse engineering is becoming easier, there are still many challenges related to their unpacking and full-blown analysis
- firmware images rely on security by obscurity, rather than security in-depth
- many trivial and non-trivial security vulnerabilites can be found just by using existing tools and frameworks

7. REFERENCES

[1] Bin2bmp on sourceforge. `http://sourceforge.net/projects/bin2bmp/`.

[2] Binvis on google project. `http://code.google.com/p/binvis/`.

[3] firmware-mod-kit on github. `https://github.com/mirror/firmware-mod-kit/tree/master/src/uncramfs`.

[4] Hexworkshop website. `http://www.hexworkshop.com/`.

[5] Reveng tools. `http://svn.gnumonks.org/trunk/reveng-tools/romfsck/`.

[6] Seagate moosedt mx1a-3d4d dmax22 firmware download. `http://www.seagate.com/staticfiles/support/downloads/firmware/MooseDT-MX1A-3D4D-DMax22.iso`.

[7] Squashfs website. `http://www.squashfs-lzma.org/`.

[8] Ubifs faq. `http://www.linux-mtd.infradead.org/faq/ubifs.html#L_ubifs_extract`.

[9] yaffs2utils on google code. `https://code.google.com/p/yaffs2utils/`.

[10] Z. Basnight, J. Butts, J. Lopez Jr, and T. Dube. Firmware modification attacks on programmable logic controllers. *International Journal of Critical Infrastructure Protection*, 2013.

[11] A. Blanco and M. Eissler. One firmware to monitor'em all. 2012.

[12] H. Bojinov and E. Bursztein. Embedded management interfaces emerging massive insecurity. In *BlackHat USA 2009(BlackHat USA 09)*, July 2009.

[13] H. Bojinov, E. Bursztein, and D. Boneh. Xcs cross channel scripting and its impact on web applications. In *Computer and Communications Security (CCS)*, November 2009.

[14] V. Chipounov and G. Candea. Reverse engineering of binary device drivers with revnic. In *Proceedings of the 5th European conference on Computer systems*, pages 167–180. ACM, 2010.

[15] A. Costin. Hacking printers for fun and profit, 2010 – 2011. `http://andreicostin.com/papers/`.

[16] A. Costin. Postscript(um): You've been hacked, 2012. `http://andreicostin.com/papers/`.

[17] A. Cui, M. Costello, and S. J. Stolfo. When firmware modifications attack: A case study of embedded exploitation. In *Proceedings of the Symposium on Network and Distributed System Security (NDSS)*, 2013.

[18] A. Cui, Y. Song, P. V. Prabhu, and S. J. Stolfo. Brave new world: Pervasive insecurity of embedded network devices. In *Recent Advances in Intrusion Detection*, pages 378–380. Springer, 2009.

[19] A. Cui and S. J. Stolfo. A quantitative analysis of the insecurity of embedded network devices: results of a wide-area scan. In *Proceedings of the 26th Annual Computer Security Applications Conference*, pages 97–106. ACM, 2010.

[20] G. Delugré. Closer to metal: reverse-engineering the broadcom netextreme's firmware. *Hack. lu*, pages 27–29, 2010.

[21] L. Duflot, Y.-A. Perez, and B. Morin. Run-time firmware integrity verification: what if you can't trust your network card, 2011.

[22] L. Duflot, Y.-A. Perez, and B. Morin. What if you can't trust your network card? In *Recent Advances in Intrusion Detection*, pages 378–397. Springer, 2011.

[23] B. Gourdin, C. Soman, H. Bojinov, and E. Bursztein. Towards secure embedded web interfaces. In *Usenix Security(Usenix Security)*, August 2011.

[24] Y. Li, J. M. McCune, and A. Perrig. Viper: verifying the integrity of peripherals' firmware. In *Proceedings of the 18th ACM conference on Computer and Communications Security*, pages 3–16. ACM, 2011.

[25] L. R. McMinn. External verification of scada system embedded controller firmware. Technical report, DTIC Document, 2012.

[26] C. Mulliner and B. Michéle. Read it twice! a mass-storage-based tocttou attack. In *Proceedings of the 6th USENIX conference on Offensive Technologies*, pages 11–11. USENIX Association, 2012.