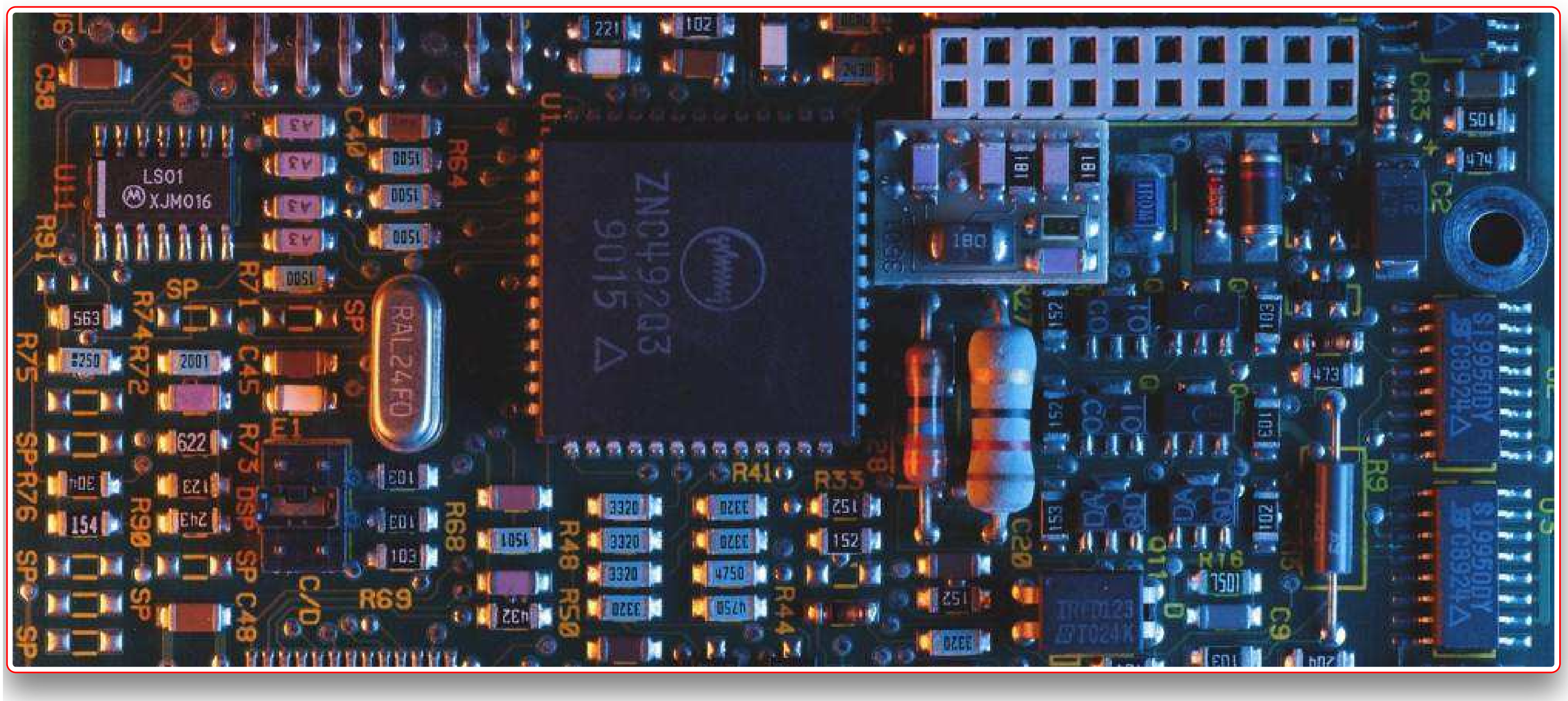
Payatu

IoT Security – Part 7 (Reverse Engineering An IoT Firmware)

Munawwar
19-June-2020



Firmware Reverse Engineering: Introduction

This blog is part of the "IoT Security" Series. If you haven't read the previous blogs (parts 1 - 6) in the series, I urge you to go through them first unless you are already familiar with those concepts and want to only read about the current topic.

IoT Security - Part 1 (101 - IoT Introduction And Architecture)

IoT Security - Part 6 (ZigBee Security - 101) previous blog in this series.

Firmware is the software counterpart of IoT devices. There are many technologies involved in building a functional IoT Firmware, and there are many vendors contributing to the development of these technologies some of which are like Cisco, Linux, Wind River, etc. In this post when I say firmware, I will mostly be referring to the software component of the device. Firmwares has varying complexity from a Bare-Metal firmware driving tiny less powerful micro-controller to microprocessor-based full-fledged Operating systems like Linux, which is used in more complex devices like router, television, etc.

In this post, we will look at different components involved in building a firmware, and then we will discuss how you can use various open-source tools to reverse engineer firmware. Reversing is an important step which will help you to do further analysis of the device and the firmware. So let's start looking at what bare-metal firmwares are?

Bare Metal Firmware

Depending upon the device's application, the technology stack is used. So, if you consider a device whose task is to read and report surrounding temperature and humidity, it won't need something very complicated software, like Linux. For those cases, Bare-Metal firmware is used. Now, what is a Bare Metal Firmware you might Ask? In simple terms, this type of firmware directly interfaces with the hardware, there is no driver or kernel involved.

Bare Metal firmware doesn't do much complicated things, they are usually tasked with not more than 3 or 4 tasks, and those tasks are put in the loop as they are scheduled to run in specific order/condition. The SDK's provided by the vendor for these devices provide life cycle methods that programmer write those functions are run in the loop. Some little complex feature variants of these SDK's are FreeRTOS, mbed-os which are real-time operating systems which allow you to do task scheduling and have a very quick response to some of the interrupt requests.

Base Metal firmwares are written in C, so all attacks like buffer overflow related vulnerabilities are also valid for these firmwares. These devices usually collect data and send it to the central server or communicate with other devices connected via UART/SPI Bus(its peripherals). There are chances of finding the vulnerability in the communication protocol, incorrect handling of packets, key exchange, buffer overrun, etc.

Micro-controller based devices are not just used in sensor networks; they are everywhere, from the refrigerator, microwave oven, security alarm system, your car has dozens of these, and so does your laptop/computers. Usually, when you don't have source code for this firmware, you take the approach of reverse engineering.

Reversing these binaries is a little different from reversing Windows EXE and Linux ELF files; they then don't have a predefined structure. For bare-metal binaries, you need the data-sheet for the chipset and create memory-map in your disassembly tool like IDA, Ghidra, etc, to get a proper disassembly. Another very critical question memory-map also helps users to answer is what GPIOs, other peripherals is the device interacting with? This information will help to understand the functionality of the device. A lot more can be said on this but I want to keep this post so for more details visit this link. Now lets look at what is fully-blown operating system based firmware.

Fully Fledged Operating System

When you see more complex systems like Routers, Smart Home Dashboard, Drones, and healthcare devices, they do little more than what bare-metal systems do. Take, for example; a typical wireless router has features like connecting by LAN and WIFI; they can blacklist/whitelist specified MAC addresses and some modern routers have antivirus software and other protection features. To implement all these functions you need a full-fledged Operation System to support all these sophisticated features.

There are many different Operating System options for embedded systems, like Linux, Windows CE, Cisco IOS, Symbian, Android, VxWorks, etc. Some of them are special-purpose like Cisco for routers, and most of them are more general purpose. General statics say that the most popular choice of OS amongst embedded system products is Linux. There are many reasons for that, some of which is its open-source nature, flexibility, and most importantly, it's free. You can find Linux powered Devices around like internet routers, infotainment systems, etc.

This kind of firmwares is usually packaged with at least three components, Bootloader, Kernel, and the file-system. The Bootloader is the piece of software that helps in loading Operating System, Kernel and passes various information needs. Once the bootloader has done executing the Kernel take over. Once the kernel has started executing, it starts other user applications to make an Operating System usable by the end-user. This usually involves starting various applications and services in the background. Once all this is done then the user can interact with the system. All the user application and app data are stored on the file-system.

Security assessments of these devices usually start with auditing applications running on these devices; Most juicy targets are the remote application services. Issues discovered in these services have a high impact on the security of the device as attackers don't need to be anywhere near it to compromise it. Anyone on the internet can take control of it if they can exploit the vulnerability. These services are usually web-servers used to configure and control the device features or other services like UPnP, which help in discovering the device by other devices.

Security Tools

Now that we have a high-level overview of how firmware looks like, the next question that comes to your mind is how do you start dissecting the firmware and start analyzing the firmware. An analysis can broadly be classified into the static and dynamic analysis. In static analysis, you read the code and look for bugs. If you don't have the source code you start reverse-engineering the binary and read the assembly instructions to understand the functionality. While on the other side, dynamic analysis involves running the application and observing its behaviours.

When analyzing a firmware, I usually used a combination of both the analysis techniques to achieve my goal. Below are the list of some of the tools which I frequently use:

1. Binwalk - this perhaps is one of the most popular tools for unpacking a firmware. It is a firmware extraction tool, it tries to crave out binaries inside any binary blob. It does this by searching for signatures for many of the common binary file formats like zip, tar, exe, ELF, etc. Binwalk has the database of binary header signatures against which the signature match is done. The common objective of using this tool is to extract the file system like Squashfs, yaffs2, Cramfs, ext*fs, jffs2, etc. which is embedded in the firmware binary. The file system has all the application code that will be running on the device. This tool also has many parameters that you can tweak to make extraction better. You can visit this **link** to read more details about what are different parameter and how to use them.
2. Qemu - this is a valuable tool for people working on cross-architectures (like ARM, MIPS, etc.) environment which usually is the case for embedded developers. This tool provides a way to emulate binary firmware for different architectures like ARM, MIPS, etc on the host system, which is of different architectures like x86, amd64. This kind of tool is handy when you want to audit a firmware but you don't have the device or setting up a debugger for that system is very difficult. Qemu can help you to do full-system emulation or a single binary emulation of ELF file for the Linux system and many different platforms. You can check this **link** to get more details about the project.
3. gdb-multiarch - GDB is a dynamic debugging tool used when you want to pause a running process and inspect its memory and registers. However, gdb supports just the architecture for which it is compiled. But when you want to debug the application of different architecture for cross-architecture support, you will need its sister project gdb-multiarch, which helps you to do cross-architecture debugging.

4. Firmware ModKit - This tool helps you to patch the firmware and repackage it. It extracts the firmware using Binwalk and gives you a directory that has the firmware file-system laid out. You can then patch whatever you want, add/delete a file or patch an existing and the ModKit can pack it back up such that you can flash the new firmware on the device storage and boot up the newly patched firmware. You can download the code for this tool from this **link**.

My workflow when working with firmware involves all of these tools. The first step usually starts with unpacking the firmware with Binwalk. Next, try to emulate the application of interest in the firmware with the Qemu. If I am not able to emulate the binary, then I investigate the issue with gdb and patch it so that Qemu can run it and look for security issues. This setup also helps me to fuzz the service.

Conclusion

We looked at what the firmware is and what are different types of IoT Firmware you will encounter when assessing an IoT Device. I gave you an overview of how to deal with a different kind of firmware, and we also looked at how different tools can help us to security analysis of a firmware.

Continue to the next part - IoT Security – Part 8 (Introduction to software defined radio)