2016 International Conference on Wireless Communication and Network Engineering (WCNE 2016)
ISBN: 978-1-60595-403-5

Security Enhancement on Firmware for the Internet of Things

Xiao MA, Yu-qian LI, Shu-hui CHEN and Jin-shu SU

College of Computer, National University of Defense Technology, Changsha, Hunan 410073, China

Keywords: Internet of things, Firmware, Security, Hardware architecture, Bootloader.

Abstract. The Internet of Things (IoT) depicts a bright future, where any devices having computing capabilities can interact with each other. However, due to limit of energy, computing, size, and storage, the security mechanisms for IoT is still in its infancy. In this paper, we focus on the security requirements in the device to prevent possible physical attacks to expose the secure data such as user's privacy data from the device. First, we give an architecture model on hardware for the Internet of Things, to defeat the physical attack that compromises the security of the device. Following this, discuss the software support required to enforce the security within the device. We brief on the security enforced in a device by the use of secure bootloader technology and propose several design principles about it. The paper also discusses the security measures taken during the production of the device.

Introduction

The Internet of Things is omnipresent in our everyday life. Is is the core of various devices such as home routers, video surveillance systems, SCADA and basically anything we normally call electronics. All these IoT devices run special software, often called firmware, which is usually used for firmware images or firmware updates. Users can run almost any application that run in a personal computer on these devices. If the firmware of the IOT device does not have a reliable storage method, then anyone who comes into contact with the device will get the information stored on the device. Hackers who get the device firmware can break through the device to achieve the purpose of hijacking control. Even further affecting the user's property safety, and a threat to their lives and health.

With the rapid development of IoT, its firmware security protection technology has not received enough attention. According to the researchers, it is found that a large number of firmware have the problems, such as stored RSA private key and self-signed certificate in plaintext, a simple encrypted password or password hash and so on. Therefore, effective protection measures must be taken to ensure the security of the firmware stored on the IOT devices.

In this paper, we discuss the hardware and software security requirements in an IoT device, to meet the need of prevent any unauthorized access to access or retrieve firmware information.

Security Hardware Architecture Model of IoT

SoC

The SoC internally contains components such as secure ROM, bootloader, and processor that provide physical protection for information such as firmware and privacy data.

The level of physical protection varies depending on the value of the protected content. When physical access is attempted, the protection can be simply SoC tamper detection, or even zeroing out all the storage in the SoC. Tamper detection protection method does not prevent the attacker from the chip to obtain data, but can only know whether the chip has been tampered with. Zeroing requires special power and hardware support, making the cost of the chip higher.

During execution, the protected security key from the secure ROM must be loaded into RAM in clear text. At the same time, an attacker can monitor the bus from secure ROM to RAM to obtain the key. This problem can be avoided by arranging an internal RAM within the SoC. The internal RAM

holds the key or involves the intermediate value of the key encryption operation. This prevents an attacker from obtaining a protected key from any bus external to the SoC.

The bootloader for the firmware in the SoC ensures that the device boots with the correct operating system or firmware with proper processing rights. The memory management unit configured by the operating system authorizes access to the storage space in the internal RAM, which contains key operations only for security processes with special operating system privileges.

In the case where the Secure ROM is limited or pre-programmed by the hardware manufacturer, the Secure ROM can be programmed with the Master Key. This Master Key can be used to encrypt and store the device key in the internal ROM.

SoC ideal situation is as follows:

- You can not physically access the Secure ROM to retrieve the key.
- The bus in the SoC can not be monitored for protected data or keys.
- It is not possible to remove or replace any components in the SoC, or the SoC should be interrupted.

The role of each component in protecting the firmware and privacy data in the SoC is explained below.

Secure ROM

One way to securely store the device key in the persistent storage of the device is to encrypt the key before storing it. Therefore, even if anyone tried to remove the data from the permanent memory, he will never be able to get the real key. Two things are usually required to encrypt the data, the encryption algorithm and the key used for encryption. If any public algorithm, such as AES, is used to encrypt the key, the strength of the encryption is only consistent with the confidentiality of the key used for the encryption. Thus, the same problem faced with storing keys is equally applicable to storing keys used to encrypt keys. This problem will continue to occur unless an encryption algorithm known only to the device manufacturer is used. If the device has a proprietary algorithm to encrypt and store the key, the security of the key is only consistent with the confidentiality of the algorithm. Since the algorithmic code binary is stored explicitly in the device memory, and a number of tools for redesigning code such as "objdump" are available, the key remains exposed.

Another method of storing keys is to store them in secure ROM. Secure ROM is located inside the SoC in the device. Before retrieving data from the ROM, the hardware controller of the Secure ROM decodes the data. The hardware prevents unauthorized physical access from retrieving the keys stored in the secure ROM.

The intermediate value of the memory holding the key or the encryption operation involving the key is allocated in the internal RAM of the SoC. Thus, the key is protected from being intercepted by the attacker from any bus other than the SoC.

In the case where the Secure ROM is limited or pre-programmed by the hardware manufacturer, the Secure ROM can be programmed using the device Master Key. The device Master Key is a unique key for each device hardware or SoC that can be further used to encrypt and store device keys in less secure ROM.

Vulnerabilities that may exist in a secure ROM implementation are:

- The security ROM is physically deleted from the SoC, placed in another device, and made to work to retrieve the protected key.
- Access the bus between secure ROM and RAM to retrieve the protected key.
- An unprivileged application running on the device can access the API to retrieve the key from the secure ROM.

In the ideal case for the SoC, the first two vulnerabilities do not exist. The third vulnerability can be prevented by using the bootloader and implementing the correct process permissions. Therefore, an unprivileged application is not allowed to run and access the device's restricted memory location.

Internal RAM and Secure Processes

The memory space of the key or the cryptographic operations containing the key are allocated in the internal RAM of the SoC to prevent the key from being intercepted by the hacker from any bus other than the SoC. This area of memory in the internal RAM is called the secure memory area. Not every process can access this memory area. Only a process with special operating system privileges, the security process, can access the secure memory area. This is similar to a process that has administrative or root privileges in the operating system.

The operating system during boot-time configures the memory management unit to only allow secure memory access through secure processes. It is also important that the MMU configuration code in the operating system is not modified by an unauthorized user to access the secure memory area. This can be ensured by using Bootloader and code signing. The security process is configured to start during device startup. The operating system should prohibit any unauthorized process from running as a security process or starting a new security process. This prevents any downloaded application from accessing the secure store to read the key.

The result of the key process by the security process is typically public data, such as encrypted data or a public key. These output data require other less privileged processes to perform operations, such as sending output data to other devices. There are several ways to invoke a security process with a less privileged process, as in the following example.

The security process waits for the input data and begins execution. The input data storage space will be a non-secure storage area. Any lesser privilege can fill the input memory and signal the security process to begin execution by releasing the semaphore. When a security process receives a signal, it uses the key to encrypt the input data. The output of the operation is placed in a non-secure storage area, and fewer privileged processes can read output from the non-secure storage area.

Encryption and Decryption Module

In many security protocol implementations, the shared key generated by the key negotiation algorithm is used as the master key and a subkey for the encryption and decryption process is generated. When used as a master key, the shared secret is stored in the device until the expiration date specified in the protocol is exceeded. And according to the data transfer protocol, the lifetime will be days or even months. However, the lifetime of the subkey is usually very small, only seconds. In this case, the shared secret has a higher security requirement and is therefore stored securely in the device. But the security requirements of the subkeys are not as important as the security requirements of the keys. The subkey generation protocol will also make it impossible to derive the Master Key from the subkey. Encryption and decryption modules may reside outside of the SoC, where the security of the keys used for encryption and decryption are less critical. The subkeys are generated internally in the SoC and passed to the encryption / decryption module outside the SoC. In some other protocols, the shared secret or the key itself is used for encryption / decryption. In this case, the encryption / decryption module should reside within the SoC to prevent the key from being intercepted by the attacker from the bus external to the SoC.

The encryption and decryption module therefore depends on the security requirements of the key used for encryption / decryption, and the choice resides internally or externally of the SoC.

Bootloader

Das U-Boot full name is "the Universal Boot Loader", generally referred to U-Boot. It is the main open source bootloader for embedded devices, which load the device's operating system kernel. The "Universal" refers to two aspects: one is support lots of the operating system. On the other hand is to support most of the processor. It can be used in a variety of computer architectures, including ARM, MIPS, PPC and x86. U-Boot can not only boot the embedded linux system. But also to guide FreeBSD, VxWorks, NetBSD and other systems.

Bootloader and Code Signing

Although the key is protected by hardware security, it must be invoked by some APIs. Therefore, you must ensure that the device's firmware can not be modified, thereby preventing unauthorized users from extracting keys from the device by using these APIs. The firmware also contains security-critical code, such as code for handling security-critical hardware configurations. Therefore, any attempt to rewrite the firmware component of the device must be rejected. The existence of bootloader can ensure this.

At boot-up before the firmware code is loaded, the bootloader checks to see if the firmware is correct and prevents the device from starting when the device firmware is modified or replaced. The following discussion is an example of a secure bootloader implementation.

The public and private keys are generated by the device manufacturer and are used for signing and verifying the device firmware code. The private key must be kept secret by the device manufacturer.

The bootloader is located on a write-protected ROM in the SoC. Leaving the bootloader in a write-protected ROM ensures that the bootloader itself is not modified. In addition to the initialization code that boots the boot device, the bootloader includes a signature verification module for the firmware code and a code validation public key to validate the firmware code.

Use the device manufacturer's code verification private key to sign the firmware code. The bootloader checks the validity of the code at boot time by verifying the signature using the code validation public key. Although the private key used to sign the firmware code is never provided with the device, it must be kept secret by the device manufacturer. The compromise of the private key of the firmware code makes it possible for anyone who has access to the private key to write and sign the acceptable code for the bootloader.

Modifications to some files may pose a threat to the security of the device. This type of file also needs to be signed with the device's firmware code.

There are also device-specific files, such as encryption keys or device certificates. They are modified to affect the equipment for safe data transmission, but this does not endanger the safety of equipment. Signing these device-specific files will result in overhead during device production or during device firmware upgrades, and the device manufacturer needs to sign the firmware code for each device. Because device security is not affected by modifications to these files, these files can be saved to the device without signing.

Principles of Bootloader Design Based on U-Boot for IoT

Size. U-Boot is a bootloader, i.e. its primary purpose in the shipping system is to load some operating system. That means that U-Boot is necessary to perform a certain task, but it's nothing you want to throw any significant resources at. Typically U-Boot is stored in relatively small NOR flash memory, which is expensive compared to the much larger NAND devices often used to store the operating system and the application.

At the moment, U-Boot supports boards with just 128 KiB ROM or with 256 KiB NOR flash. We should not easily ignore such configurations - they may be the exception in among all the other supported boards, but if a design uses such a resource-constrained hardware setup it is usually because costs are critical, i.e. because the number of manufactured boards might be tens or hundreds of thousands or even millions.

A usable and useful configuration of U-Boot, including a basic interactive command interpreter, support for download over Ethernet and the capability to program the flash shall fit in no more than 128 KiB.

Speed. The end user is not interested in running U-Boot. In most embedded systems he is not even aware that U-Boot exists. The user wants to run some application code, and that as soon as possible after switching on his device.

It is therefore essential that U-Boot is as fast as possible, especially that it loads and boots the operating system as fast as possible.

To achieve this, the following design principles shall be followed:

- Enable caches as soon and whenever possible
- Initialize devices only when they are needed within U-Boot, i.e. don't initialize the Ethernet interface unless U-Boot performs a download over Ethernet; don't initialize any IDE or USB devices unless U-Boot actually tries to load files from these, etc.

Also, building of U-Boot shall be as fast as possible. This makes it easier to run a build for all supported configurations or at least for all configurations of a specific architecture, which is essential for quality assurance. If building is cumbersome and slow, most people will omit this important step.

Portable. U-Boot is a bootloader, but it is also a tool used for board bring-up, for production testing, and for other activities that are very closely related to hardware development. Please make sure that any code you add can be used on as many different platforms as possible.

Avoid assembly language whenever possible, maybe a static DRAM initialization and the C stack setup should be in assembly. All further initializations should be done in C using assembly/C subroutines or inline macros. These functions represent some kind of HAL functionality and should be defined consistently on all architectures. E.g. Basic MMU and cache control, stack pointer manipulation. Non-existing functions should expand into empty macros or error codes.

Don't make assumptions over the environment where U-Boot is running. It may be communicating with a human operator on directly attached serial console, but it may be through a GSM modem as well, or driven by some automatic test or control system. So don't output any fancy control character sequences or similar.

Debuggable. Of course debuggable code is a big benefit for all of us contributing in one way or another to the development of the U-Boot project. But as already mentioned in section "3.2.3. Portable" above, U-Boot is not only a tool in itself, it is often also used for hardware bring-up, so debugging U-Boot often means that we don't know if we are tracking down a problem in the U-Boot software or in the hardware we are running on. Code that is clean and easy to understand and to debug is all the more important to many of us.

- One important feature of U-Boot is to enable output to the console as soon as possible in the boot process, even if this causes tradeoffs in other areas like memory footprint.
- All initialization steps shall print some "begin doing this" message before they actually start, and some "done" message when they complete. For example, RAM initialization and size detection may print a "RAM: " before they start, and "256 MB\n" when done. The purpose of this is that you can always see which initialization step was running if there should be any problem. This is important not only during software development, but also for the service people dealing with broken hardware in the field.
- U-Boot should be debuggable with simple JTAG or BDM equipment. It shall use a simple, single-threaded execution model. Avoid any magic, which could prevent easy debugging even when only 1 or 2 hardware breakpoints are available.

Usable. Please always keep in mind that there are at least three different groups of users for U-Boot, with completely different expectations and requirements:

- The end user of an embedded device just wants to run some application; he does not even want to know that U-Boot exists and only rarely interacts with it.
- System designers and engineers working on the development of the application and/or the operating system want a powerful tool that can boot from any boot device they can imagine, they want it fast and scriptable and whatever - in short, they want as many features supported as possible. And some more.
- The engineer who ports U-Boot to a new board and the board maintainer want U-Boot to be as simple as possible so porting it to and maintaining it on their hardware is easy for them.
- Make it easy to test. Add debug code.

Please always keep in mind that U-Boot tries to meet all these different requirements.

Security. U-Boot make sure that the device boots with the correct operating system or firmware with proper processing rights. By using U-Boot and code signing, ensure that the operating system during boot-up configures the memory management unit to only allow secure memory access through secure processes. An unauthorized or unauthorized application running on the device can not retrieve data in the ROM by calling the API, which means that the MMU configuration code in the operating system is not modified by an unauthorized user to access the secure memory area.

Conclusions

The safeguards used to securely transfer data between devices of the Internet of Things are ripe enough to prevent any third party from decrypting and gaining access to the protected content. However, the safeguards used to protect stored data within the IoT device are not yet foolproof.

When a physical attack on the SoC occurs, the tamper resistant protection mechanism in the device may require hardware circuitry to zero the data. The more hardware security measures implemented in the device to protect its data, the more expensive the device is. Therefore, the hardware security measures implemented in the device are the result of a combination of data value and data protection costs. Although the implementation of high cost, but the use of tamper resistant method to protect the data within the IoT device and the need to protect data security users are effective.

Unlike the traditional Internet devices, the hardware bootloader of the IOT is strictly limited in terms of storage resources and computing resources. Therefore, the firmware bootloader procedure is designed strictly according to the principle of lightweight. In some cases, people do not attach great importance to the boot speed of the firmware. Therefore, for such equipment, firmware boot speed is not one of the factors to consider.

Acknowledgement

This research was financially supported by my teacher. I would like to express my special thanks of gratitude to my teacher Shu-hui Chen who gave me the golden opportunity to do this wonderful project on the topic "Security Enhancement on Firmware for the Internet of Things", which also helped me in doing a lot of Research and I came to know about so many new things I am really thankful to.

Secondly I would also like to thank my wife and friends who helped me a lot in finalizing this project within the limited time frame.

References

[1] FIPS PUB 186-2, Digital Signature Standard (DSS), January 2000, Available at http://csrc.nist.gov/publications/fips/fips186-2/fips186-2-change1.pdf

[2] RSA Laboratories, PKCS#1 v2.1: RSA Cryptography Standard, June 2002, http://www.rsa.com/rsalabs/node.asp?id=2125

[3] ITU, Recommendation X.509, Available at http://www.itu.int/rec/T-REC-X.509- 200508-I

[4] FIPS 197, Advanced Encryption Standard (AES), November 2001, Available at http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf

[5] FIPS 140-2, Security requirements for cryptographic modules, May 2001, Available at http://csrc.nist.gov/publications/fips/fips140-2/fips1402.pdf

[6] RSA Laboratories, http://www.rsa.com/rsalabs/node.asp?id=2193