Towards Secure Remote Firmware Update on Embedded IoT Devices

Márton Juhász Dorottya Papp Levente Buttyán[1]

Abstract: An important security problem in IoT systems is the integrity protection of software, including the firmware and the operating system, running on embedded IoT devices. Digitally signed code and verified boot only partially solve this problem, because those mechanisms do not address the issue of run-time attacks that exploit software vulnerabilities. For this issue, the only known solution today is to fix the discovered vulnerabilities and update embedded devices with the fixed software. Such an update should be performed remotely in a secure and reliable way, as otherwise the update mechanism itself can be exploited to install compromised software on devices at large scale. In this work, we propose a system and related procedures for remotely updating the firmware and the operating system of embedded IoT devices securely and reliably.

Keywords: Embedded Systems, Internet of Things, Security

1 Introduction

IoT systems are built from network connected embedded devices, and their security heavily rely on the security of those embedded devices. One of the most important security aspects in this context is the integrity of the software running on embedded devices. The reason is that unauthorized modification of software can result in arbitrary behavior of those devices, and as a consequence, loss of trust in the entire IoT system built upon them. In particular, protecting low level software, such as the operating system (OS) and the firmware, is important, because typically these components are responsible for implementing many security controls and they provide trusted services (e.g., in the form of system calls) to higher layer software.

Digitally signing software components, including the firmware and the OS kernel, and important data, such as configuration files, combined with some hardware based *root-of-trust* and a secure boot process, which ensures that software components are loaded and executed only if their signature is valid, can help protecting the integrity of software, but does not entirely solve the problem. In particular, signed code and verified boot ensure that the device runs intact code right after a reset, but software can also be compromised at run-time by exploiting design and implementation level vulnerabilities in it. For instance, an attacker may be able to execute arbitrary injected code on a device by exploiting software bugs, such as not checking the amount of data copied into a limited size buffer or using dangling pointers, leading to memory corruption [1].

When software vulnerabilities are discovered, they need to be fixed, and embedded devices need to be updated with the fixed software. This applies to the OS and the firmware too. In addition, due to the potentially large number of embedded devices used in IoT applications and their often special operating environment, it is preferable that the update process can be carried out remotely, without the need to physically approach each and every device. Remote firmware and OS update is sometimes also called *over-the-air* (OTA) update, because the update may be downloaded by the devices over wireless communication links.

Security of the remote update process itself is of paramount importance [2], as we would like to avoid that attackers exploit an insecure update mechanism to install a compromised OS or firmware remotely at large scale. Potentially, such compromised updates may prevent any further legitimate update, leaving the control of all compromised devices in the hand of

[1]All authors are with the Laboratory of Cryptography and System Security (CrySyS Lab), Department of Networked Systems and Services, Budapest University of Technology and Economics

the attacker. Recovering from such a situation would require manual update of every device, which would be time consuming and expensive.

Besides security, the update process must be reliable and *fail-safe*, by which we mean that an unsuccessful update should not leave the devices in a state where they cannot boot and operate properly, but it should be possible to detect if the update failed and to load the last stable version of the updated software. At the same time, attackers should not be able to force a version rollback when the devices run a stable version of the software, because if that was possible, then they could force the devices to re-install an old, potentially vulnerable version of the software through which they can compromise the devices again.

In this extended abstract, we introduce a remote firmware and OS update system and mechanism for embedded IoT devices that satisfy the above requirements on security, reliability, and rollback protection.

2 Architecture

In this section, we give a high level overview on our update system architecture, which consists in partitions on the persistent storage (e.g., flash disk) of the embedded device, different images stored on those partitions, and a few log files.

We use 3 partitions with different access restrictions. The device boots from the *boot* partition, which holds the firmware image of the device and the kernel images of 2 OSs, the MainOS (typically some embedded Linux) and the UpdateOS (a trusted OS with minimal functionality, e.g., a stripped down Linux or some formally verified microkernel such as seL4[2]), as well as the root file system of the MainOS and 3 log files, called *updatelog_Firmware, updatelog_UpdateOS, updatelog_MainOS*, where different update events are logged and a control file, called *nextOStoboot*, indicating which OS to boot. Application data is stored on a separate *data* partition. Firmware and OS kernel images, as well as root file system images for the MainOS, downloaded from an update server are logged in the *downloadlog* on the *images* partition and stored on the *images* partition, from which they are copied on the *boot* partition during the secure update process. Another log file, called *selftestlog*, can also be found on the *images* partition that stores information about the result of self-testing by a freshly updated MainOS.

In order to protect the update log files (i.e., *updatelog_Firmware*, *updatelog_UpdateOS*, and *updatelog_MainOS*) from being updated by a potentially compromised MainOS, the *boot* partition can be written only by the firmware and the UpdateOS. The MainOS can write on the *data* and *images* partitions; the latter is needed in order for the MainOS to be able to download and store updates.

3 Boot process

Our architecture supports a secure boot process. After reset, code in a boot ROM verifies the digital signature of the firmware image on the *boot* partition, and on success, it loads and executes the firmware. The hash of the signature verification public key used by the boot ROM code is stored in a special, one-time programmable memory, which is written during device customization, after which this signature verification public key can no longer be modified. The firmware performs low level system verification, initializes trusted software components, such as a Trusted Execution Environment, and eventually executes the OS boot loader (e.g., U-Boot[3]).

The OS boot loader has another signature verification public key, which is used to verify the digital signatures of the OS kernel images on the *boot* partition. The OS boot loader always

[2] `https://sel4.systems`

[3] `https://www.denx.de/wiki/U-Boot/`

checks the *nextOStoboot* control file, and acts according to what is indicated in that file. When the MainOS is about to be booted, the OS boot loader writes in the *nextOStoboot* that the UpdateOS should be loaded next time, makes the *boot* partition write protected, and gives control to the MainOS kernel. Finally, the MainOS kernel verifies the integrity of the root file system image on the *boot* partition, and on success, it mounts the root file system, after which the device is up and running. When the UpdateOS is about to be booted, the OS boot loader writes in the *nextOStoboot* file that the MainOS should be loaded next time, and gives control to the UpdateOS kernel.

4 Update process

Updates are downloaded and written on the *images* partition by an update service running on the MainOS. Upon the next boot, the UpdateOS detects the update image from the *downloadlog*, verifies its digital signature, and on success, it places the update image on the *boot* partition. In case of an update of the firmware or the UpdateOS, the update image replaces the old version, as we assume that these are thoroughly tested images that function properly. However, in case of a MainOS update, both the update image and the old image are kept on the *boot* partition. If the digital signature verification fails, the update image is deleted by the UpdateOS. In any case, an appropriate log entry is created in the corresponding *updatelog* file and the device is rebooted.

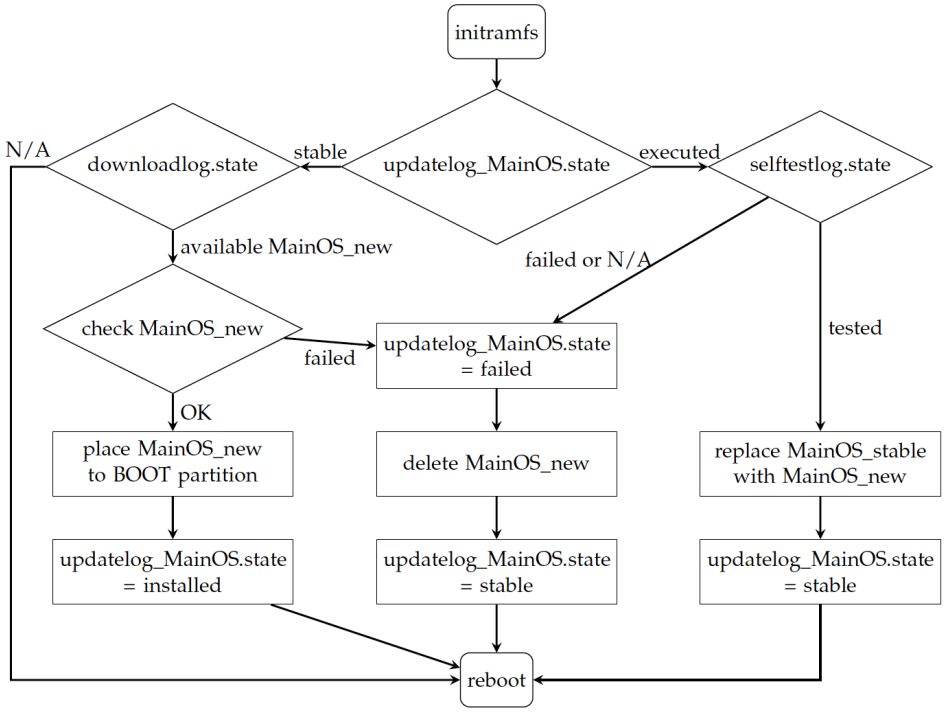

Figure 1: Simplified flowchart of updating the MainOS

In the sequel, we explain how the the MainOS is updated. The process is illustrated in Figure 1. When the device executes the boot process next time, the OS boot loader detects from the *updatelog_MainOS* that it should load and start an updated MainOS for the first time. It writes in the *updatelog_MainOS* the version to be booted, makes the *boot* partition write protected, and transfers control to the MainOS kernel. The MainOS mounts the root file system and performs self-testing. If everything goes well, the result of the self-testing is written in the

selftestlog. Upon next boot, the UpdateOS detects that the update was successful by observing the *updatelog_MainOS* and the *selftestlog*, so it deletes the old MainOS image from the device and logs in the *updatelog_MainOS* that the update was successful.

However, the self-testing may fail or the new version of the MainOS may hang or crash. Such hangs or crashes are handled with a watchdog mechanism that reboots the device. In this case, the UpdateOS detects the failed self-testing of the updated MainOS by observing in the *updatelog_MainOS* that an update was booted, while missing any indication of a successful self-test in the *selftestlog*. When this happens, the UpdateOS deletes the failing update from the device, logs the failure in *updatelog_MainOS*, and reboots the device. After the reboot, the latest stable MainOS is loaded and executed.

Security is achieved by installing only properly signed updates by the trusted UpdateOS and logging all relevant events to the *updatelog* files that cannot be modified by the potentially compromised MainOS or any applications running on it. Trust in the UpdateOS is based on the following factors: (1) the UpdateOS is signed and its signature is verified before loading and executing it; (2) the UpdateOS executes only for a limited amount of time; (3) the UpdateOS has a reduced functionality (e.g., all unnecessary features and services are disabled, including even network access); and (3) the UpdateOS can potentially be formally verified due to its stripped functionality.

Fail-safety is achieved by using a watchdog mechanism that reboots the device upon failures and by using log files in order to detect a failed self-test after an update. Moreover, the latest stable version of an updated component is kept on the device until the success of the update can be verified, so in case of failure, the device can still boot the latest stable version.

Finally, rollback protection is achieved by keeping information about updates in the *updatelog* files, which cannot be modified by the potentially compromised MainOS or the applications running on it, and by removing old versions from the device after a successful update.

The described architecture and update process are complex enough to warrant for a thorough verification. For this reason, we used the UPPAAL[4] model checker to model the update process and to formally verify its correctness. We checked the following two properties:

- **Update is possible:** When a given version of the MainOS is running and there is a functioning update available, it is possible to reach a state where this update is successfully installed.
- **Rollback is impossible:** It can never occur that a given version of the MainOS is successfully installed when a newer version was running and marked as stable in the past.

Our update process described above satisfies both properties.

Acknowledgment

The presented work was carried out within the SETIT Project (2018-1.2.1-NKP-2018-00004), which has been implemented with the support provided from the National Research, Development and Innovation Fund of Hungary, financed under the 2018-1.2.1-NKP funding scheme.

References

[1] L. Szekeres, M. Payer, L. T. Wei, and R. Sekar. Eternal war in memory. *IEEE Security and Privacy Magazine*, 12, May-June 2014.

[2] TCG IoT-SG. TCG Guidance for Secure Update of Software and Firmware on Embedded Systems – version 1.0, revision 64. TCG Reference Document – Draft, July 2019.

[4] http://www.uppaal.org/