Creating an Embedded Product with Support for UEFI Secure Boot

Insyde Software

Martin O. Nicholes, Ph.D., Engineer, Insyde Software
December 2012
Abstract:

Embedded system designers want to control what software runs on their system. This enhances security of a system, and makes it much more difficult for unauthorized software to run during the boot of a system. However, to ship a system with the secure boot feature enabled, fundamental changes need to be made to the way the system is designed, manufactured, deployed, and maintained. Many companies have used signing services for operating system drivers, but supporting the secure boot infrastructure on a product line is a much more difficult proposition. This paper discusses aspects of this problem and reviews resources that can help solve it. The focus is on embedded systems, but the principles are applicable to any computer system. Remember that releasing a system with secure boot peaks hacker interest and makes the system a target, so every effort must be made to minimize security holes from design phase to field deployment.
Introduction

System designers have always wanted to control what software will run on their embedded system. In order to provide this control, some way to check each software module before execution is required. Cryptographic techniques to verify a software module are readily available. However, these techniques require a way to manage critical cryptographic elements, such as keys. The secure boot concept has been around for quite some time, but the Unified Extensible Firmware Interface (UEFI) Forum has recently taken steps to standardize the technique. If UEFI Secure Boot is implemented correctly, a system can now claim that it was booted with the “correct” software. This trusted software stack can provide increased protection of the system at runtime and make it very difficult to install root kits.

In cryptographic systems, such as UEFI Secure Boot, the underlying cryptographic algorithms are well understood. As long as the security algorithm is correctly implemented, a system will have some security properties that can be demonstrated. Algorithms are chosen for a certain lifetime. For example, if you have a computer model that will only be in operation for three years and then becomes obsolete, a less secure algorithm can be selected. However, if you are creating products with a ten year lifetime, then a much more secure algorithm should be selected. Time span is only one aspect of the selection process. Another important aspect is the value that is being protected. For example, if hacking a security solution could take down a product line with millions of installed systems, a more secure solution is desirable. Another consideration, especially for embedded systems, is the suitability of the algorithm for running on the target processor. Some processors have instructions designed to speed up certain cryptographic algorithms. However, selecting the right algorithms is the easy part of the problem. How can other cryptographic elements be managed, such as keys?

The keys used in the UEFI secure boot solution are asymmetric. There are public and private keys in each key pair. Typically, the public key is freely available, and can be used to verify that some data came from the owner of the private key and the data has not been changed in transit. The difficult problem is how the keys are handled. How can a system decide if it trusts a particular public key? How can a company distribute its’ public key? This is an old and difficult problem called Public Key Infrastructure (PKI). Another very important aspect is how can a company protect its’ private key? If it is lost or stolen, then someone else can send data that looks like it came from that company. Protecting the private key presents serious risks. Many companies have had private keys compromised with costly results both in lost reputation and money spent to cleanup afterwards.

The UEFI Secure Boot solution helps in many of these areas. It provides a standard way to process software modules, so that any UEFI Secure Boot system can evaluate them. In addition, it specifies cryptographic algorithms that are safe to use in today’s products. Finally, it provides a basic PKI that a platform uses to store public key information and to evaluate public keys. It is basic, since the firmware environment is limited compared with a normal operating system. The UEFI standard also provides guidelines on protecting the database of keys and other cryptographic elements required for secure boot operation.

Companies that develop UEFI platforms, including embedded, have a great foundation for a secure boot solution.

This paper will discuss various aspects of UEFI secure boot in the embedded environment, but the principles are applicable to any UEFI system.

**Secure Boot Background**

There have been several excellent papers published to explain the secure boot process, so this paper will provide a very brief review of UEFI Secure Boot to establish terms for the rest of the paper. See the list of references for works that provide more details on secure boot.

UEFI Secure Boot defines an infrastructure that enables the computer system to validate software before execution. This means that only trusted software will be run during a boot process. UEFI firmware will contain the necessary cryptographic subsystem to validate the software, including cryptographic functions and the necessary public keys (may be held in a certificate). The store of public keys has controlled access, so that only authorized changes can be made. The stores of cryptographic keys and image information are designed to allow various levels of access.

There are four stores (PK or Platform Key, KEK or Key Exchange Keys, db or execute database, dbx or do-not-execute database) defined in the UEFI Secure Boot specification. The owner of the platform (either a person or a company) can make changes to the secure boot stores, for example adding the measurement of a piece of software, clearing all the secure boot stores or setting the stores to factory defaults. Figure 2 shows how authority to change these databases is hierarchical. The owner of the PK may make changes to the PK and KEK. The owner of a KEK key may make changes to the db and dbx stores. The complexity of the secure boot stores makes it a requirement to have an excellent field support plan for secure boot.

Once the secure boot stores are configured correctly, the platform will use the information in the stores to validate every piece of software that is loaded as the operating system boots. In general, the system will load files prior to execution and validate the signature. If the signature is present and verified using a key from the db, or the signing key chains to a key in the db, the software module can be run. If the hash (cryptographic measurement) of the file is found in the db, the file is approved to execute as well. Of course, if a piece of software is signed by an untrusted key or has its measurement in the dbx (do-not-execute) store, it will not be run.
The UEFI specification recommends SHA 256 (for hashing) and RSA 2048 (for encryption) as minimum cryptographic algorithms for secure boot. These are proper minimums for use in other system security, such as validating firmware updates.

Some of the secure boot decisions can be based on policy alone. For example, any file loaded from onboard flash is trusted explicitly. These are the basic mechanisms of secure boot.

After OS boot, verification of executable files, if supported, relies on OS dependent security systems and a separate key store.

At this point, we would like to emphasize that the services of a qualified security professional will help a company make informed and security conscious decisions on their secure boot design. This paper reviews the requirements at a high level, but practical design of the full solution is left to each company. If a secure boot solution is built upon weak security principles, it could lead to costly rework and security issues in the installed base of systems, affecting a particular model, product line, or every computer shipped with this feature. Finally, although the UEFI Secure Boot design is based on sound cryptographic principles, particular implementations will have security vulnerabilities. Use of secure system development techniques will limit these vulnerabilities.

**Embedded System Considerations**

Embedded systems have special characteristics that should be taken into account when designing a secure boot system. Embedded systems generally experience very few hardware changes in the field, perhaps even to the point of restricting all changes after platform shipment. This helps ease the issues with keeping the secure boot information “current” over the lifetime of the system. Some embedded systems must allow hardware changes, for example, network storage solutions need to support addition and replacement of hard drives.

Changes to the software shipped with the system are more likely, although this could be restricted as well. Often in embedded systems, customers are not provided with the means to add software to the system. Still, it is poor practice to design a system so that software changes are difficult for field support. Changes to an embedded system may be necessary due to software updates required for fixing a defect or a security issue.

Embedded systems are likely to have a longer lifetime in the field than general systems, and a smaller installed base. System lifetime and installed base size are important factors to help determine requirements for key strength and certificate lifetime.

Selection of appropriate cryptographic algorithms is critical for embedded systems. There are several questions to consider during selection. First, does the target processor have any special hardware support for a particular algorithm? For example, Intel® processors in the Westmere family (containing the 2010® core) have six Advanced Encryption Standard (AES) instructions providing full hardware support for AES. These instructions speed up encryption and decryption and provide for key expansion.
Second, what is the flash footprint of the algorithm? Third, what is the lifetime of the product that will use the algorithm? Considering the answers to these will help with the selection process. Finally, selecting the library to support the cryptographic algorithms is another crucial decision. Of course, open source alternatives such as OpenSSL are available, but many vendors have cryptographic libraries for sale, that offer faster execution and smaller flash footprint. In addition, choosing a vendor who offers field support for a cryptographic library is an added benefit. This can help smaller companies by providing external resources to track security issues in a library and implement patches to resolve those issues.

User Interface Design

The main issues with user interfaces revolve around what capabilities are provided to the user, and how those capabilities are protected against unauthorized use. The following table lists common capabilities as well as notes regarding that capability.

<table>
<thead>
<tr>
<th>Secure Boot Capability</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Disable Secure Boot</td>
<td>Administrator only, possibly requiring physical presence</td>
</tr>
<tr>
<td>Clear Secure Boot Databases</td>
<td>Administrator only, possibly requiring physical presence</td>
</tr>
<tr>
<td>Add unsigned update to Secure Boot Databases</td>
<td>Dangerous operation, possibly disallow</td>
</tr>
<tr>
<td>Add a file to the approved (db) database</td>
<td>Administrator only, possibly requiring physical presence</td>
</tr>
<tr>
<td>Restore Secure Boot Databases to Factory Default</td>
<td>Administrator only</td>
</tr>
</tbody>
</table>

*Table 1 – Secure Boot Capabilities*

Protecting against unauthorized access to secure boot state leads the designer to evaluate the security properties of user credentials used on the system. Does the system enforce password complexity or aging? Does the system force the default password to be changed? Does the system support 2-factor authentication, adding the use of a smart card or biometric device? These methods will raise the security level of the system, and protect the secure boot stores from tampering.

Hardware Design for Secure Boot

The system must include hardware protection mechanisms for the firmware image, as well as secure boot databases. Firmware should only be changeable during a secure update process. Hardware support will be necessary to achieve this goal, as covered in NIST SP800-147. The requirements for the secure boot databases are very similar, and are called out in the UEFI Firmware Specification. For example, the platform “public key must be stored in non-volatile storage which is tamper and delete resistant.” The Key Exchange Keys requirement is similar, the “public key must be stored in non-volatile storage which is tamper resistant.” Some operating systems may have even more strict requirements for storage protection, and require some amount of storage for the secure boot databases. For example, there may be a requirement to be able to restore factory settings for secure boot, including the state of the
cryptographic databases (PK, KEK, db, and dbx). This may require a sizeable amount of non-volatile storage on the system.

Hardware mechanisms that provide protection of the flash include: locking blocks of the flash before running untrusted code and locking the whole flash part for untrusted code. Some chipsets provide support for locking the flash during normal operation and unlocking in a special mode available only to trusted code.

**System Design for Secure Boot**

In addition to hardware changes, a system designed for secure boot must have a solid overall security design. Since the critical trusted code is in the firmware system, the design must require manufacturer control of firmware updates. The system must also protect the secure boot databases from unauthorized modification. Normally, a full system security design will employ 4 keys, one for the PK, one for the KEK, one for the db, and one for validating signed firmware updates.

An embedded system has very few driver changes, since the hardware environment is very stable. However, if any drivers are obtained or added into the system outside of the main firmware image, then it is important to work with the driver vendor on a process to deliver signed drivers. Of course, the corresponding public key must be in the secure boot database.

Similarly, any utilities needed to maintain or repair the system once secure boot is enabled must also be signed. This requires the development of a release process where utilities or drivers can be signed by authorized members of the company. Control of the signing key (the private key) is a critical element of this process. The signing key should not be protected by software alone, but in some type of hardware device, ranging from a smart card to a Hardware Security Module. In addition, the public half of this signing key must be present in the db, so that authorized utilities and drivers will run. The company manufacturing the system can make sure this happens.

Protection of the other private keys is also crucial. The private part of the PK is very powerful, as the owner of this key can change the secure boot databases on any platform using that PK. In the same way, the private half of a company’s KEK is important to protect, since the owner of this key can apply updates to the db and dbx of any system using the same KEK. This could be used to add a bad signing certificate to the db, allowing unauthorized software to execute. Finally, the private half of a company’s signing key, if lost or stolen, could be used to sign any software, enabling it to run on the associated company’s systems.
Typically, a company will have a cryptographically stronger key that is used to sign or authorize a second level signing key. The primary signing key is held in a certificate that is valid for a longer time period than the second level signing key. The second level signing key is valid for a shorter time, more in line with the useful lifetime of the system. UEFI Secure Boot allows a lower level key to be used for signing software. The lower level key can be included in the db, or an ancestor key that authorized that key could be included. Figure 3 shows how the top level certificate authorizes the second level certificate.

In order to make sure that only authenticated firmware is installed on the system, NIST SP800-147 recommends that the update process validate the firmware using a signature, which can be verified with an embedded key. In addition, it recommends planning how firmware will be modified when repairing or re-purposing a system. The strength of the key used to verify the signature should be appropriate for the lifetime of the system and the value of data handled by the system. A company should also consider the breadth of the embedded key, meaning is there one key for all products from the company, all units from a product line, or some other level. Each company must evaluate the risk of a lost or stolen key, determining the number of installed systems that will be exposed and will need special remediation.

### Secure Boot Design Resources

This section will review the resources available to a company in the area of secure boot. If a company develops boot firmware in-house, then the firmware team must review and understand the UEFI Firmware Specification. This specification will detail the additional support needed in UEFI services to support secure boot. In addition, operating systems have requirements that must be met. Open source UEFI code is available on the Sourceforge Tianocore project. This will provide a framework for development of a UEFI system supporting secure boot.

Recent versions of Linux and Windows have support for various secure boot features. Several white papers and specifications have been released detailing operating system support. These resources can help your company tune the design for a target operating system. Windows 8 has a standard set of public keys that must be installed in the KEK and db on systems shipping with secure boot. Some Linux distributions rely on the single Microsoft signing service key and hold their own set of keys in the loader itself.

In order to help create signed drivers and utilities, Microsoft has created a driver signing service as part of the Windows Developer Center site. This allows smaller companies, who do not want to create their own driver signing solution, to ship drivers that will be able to run on a secure boot system. This also helps solve the problem of requiring too many keys in the system db. If the public key for the Microsoft driver signing service is present, then any vendor’s software that was signed by that signing service will be authorized to run on the system.
Summary

Embedded systems, although not as open as traditional laptops, desktops and servers, still present a challenge to secure boot design. Care must be taken in system design to ensure hardware support is available as needed to protect secure boot databases, and operations affecting secure boot databases are authorized. In addition, the company must develop security processes to protect various signing keys, sign approved payloads, control firmware updates, and maintain deployed systems that are protected by secure boot.

References


Bobzin, Jeff. “Implementing a Secure Boot Path with UEFI 2.3.1.”

Cooper, David. NIST Special Publication 800-147, “BIOS Protection Guidelines.”

Freescale Semiconductor. “Secure Boot.”

Garret, Matthew. “UEFI Secure Boot Cases and Linux.”
<http://www.uefi.org/learning_center/4_-_UEFI_summit_RedHat.pdf>

Pavlik, Vojtěch. “Suse and Secure Boot, the Details.”
<https://www.suse.com/blogs/uefi-secure-boot-details/>

<http://focus.ti.com/pdfs/vf/wireless/platformsecuritywp.pdf>

van der Hoeven, Arie. “Deploying Secure Boot: Key Creation and Management.”

Insyde is a registered trademark of Insyde Software. Other names and brands may be claimed as the property of others.