Prevent edge node attacks by securing your firmware

Configuring Kinetis® MCU capabilities with ARM® mbed™ TLS for a secure boot

***Donnie Garcia**, Solutions Architect for Secure Transactions, NXP*
***Diya Soubra**, Senior Product Marketing Manager, ARM*

Abstract

The reality of a world filled with smart and aware devices is that there is a world of attack possibilities versus the technology our society is reliant upon. Just consider the scenario where an IoT edge node is attacked by replacing firmware to allow access to a trusted network. In today's Internet of Things (IoT) world of connected devices, phishing scams perpetrated by re-purposing edge nodes is a real threat. Therefore, a plan for the development, manufacturing and deployment of IoT edge node devices must be made. The complexities of life cycle management create a demanding environment where the end developers must make use of a range of hardware security features, software components and partnership to achieve their security goals and prevent malicious firmware from being installed onto IoT edge node devices. Essential to sustaining end to end security is a secure and trusted boot, which can be achieved with the right MCU hardware capabilities and ARM® mbed™ TLS. This paper will introduce a life cycle management model and detail the steps for how to achieve a secure boot with a lightweight implementation leveraging NXP® ARM Cortex®-M-based microcontrollers with mbed TLS cryptography support.

Secure designs begin with a security model consisting of policies, an understanding of the threat landscape and the methods used to enforce physical and logical security. In order to protect firmware execution within today's threat landscape, there must be a policy to only allow execution of authenticated firmware. The methods used to enforce this policy rely on MCU security technology to create a protected boot flow. The boot firmware can contain public key cryptography to authenticate application code. In addition to these components integrated in the end device, there are tools and steps that must be taken in the manufacturing environment using manufacturing hardware for code signing and host programs for provisioning.

Table of Contents

Introduction: Internet of Things Phishing

From connected villages to intelligent vehicles, the value of connecting various sensors to improve efficiency is spurring visionaries to develop Internet of Things (IoT) edge nodes. Innovation around smart and connected electronic devices is pushing technology into scenarios that were unimaginable just a short time ago. The way we go about our daily lives is changing because of technology adoption around the IoT. This technology adoption is a contributing factor to the security threat of focused attacks using IoT edge devices as a platform for phishing. As user acceptance and trust in our technology increases, people are more willing to provide credentials to their smart watch, sprinkler system or home camera.

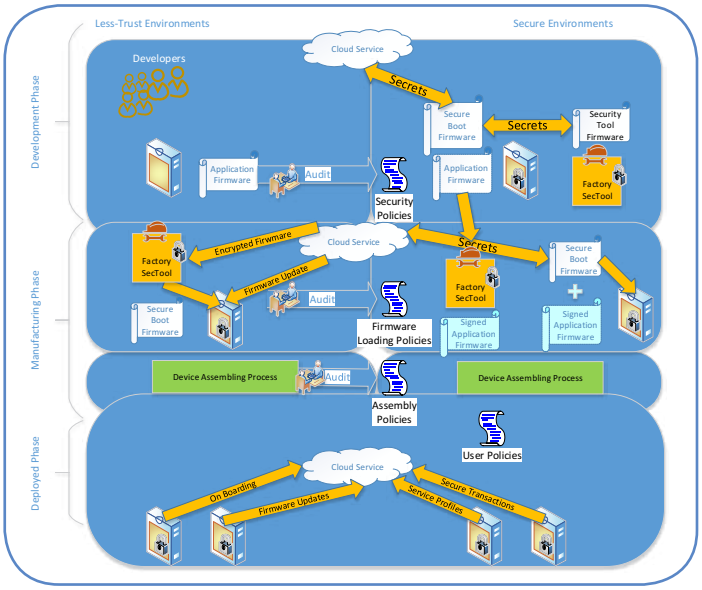
Just as email phishing scams that come with malicious attachments or links that lure users to malware, a determined attacker can take an un-protected IoT edge node device, repurpose it with malicious firmware and use it to access protected data or services. For the case of an IoT edge node phishing scheme, this would be providing access to user networks leading to data breaches or even distributed denial of service attacks.

At the root of this vulnerability is that too many embedded designs do not protect their firmware with a trusted boot flow including application authentication before execution. Implementing this level of security is a challenge, but with the proper planning, the right hardware, software and tools, firmware can be protected from such attacks. This paper outlines the design and development of an implementation that can be done using NXP Kinetis® MCUs along with ARM mbed TLS support. It will also outline the future of embedded security with microcontroller enhancements such as TrustZone® for ARMv8-M to provide a foundation for a more efficient and developer-friendly security solution in the future.

Lifecycle View

The challenge with regards to making a secure embedded device can be represented in the lifecycle view shown in *Figure 1*. The IoT edge node device flows through several stages. These are shown on the left of the diagram as *Development, Manufacturing and Deployment*. Within these stages of the lifecycle the product could be in *Secure Environments* or *Less-Trust Environments* as shown at the top of the diagram. For example, in the development stage, application code could be developed by external developers which would be in a Less-Trust Environment. Alternatively, if the firmware development is handled by trusted internal developers then this would be in the more *Secure Environment*.

Figure 1: Lifecycle View for Secure IoT Edge Node



Throughout the lifecycle, there are important policies that govern how the device should be handled. These are detailed below as the Security Policies, Firmware Loading Policies, Assembly Policies and User Policies. Some examples for the contents for these essential documents are in the following *Table 1*.

Table 1: Policies for Lifecycle Management

Policy	Examples
Security policies ensure that the application code maintains the security of the end device.	• No prompts for sensitive data such as Enter PIN or password • A list of words that the end device should not say
Firmware loading policies ensure that the proper steps are taken and controls are in place to protect the programming of the end device.	• Password control for firmware source binaries • Upon receiving the microcontroller, the device should be completely erased to ensure that it is in a known state (no un-wanted firmware)
Assembly policies ensure that only approved components are used.	• All components should be inspected for expected for proper markings during assembly
User policies provide guidelines for the end user to maintain the security of the device.	• Visual inspection of the device for tampering • Device should be physically protected behind locked doors

In the development phase, the product owner develops a factory security tool and security tool firmware. This tool is used to generate public key/private key pairs, sign application firmware and interface securely to a cloud service provider. The product owner also develops the root of trust firmware such as the secure bootloader. This firmware performs secure boot and secure boot loading. This stage is where sensitive data such as product IDs and service IDs are generated. These secrets can be passed to the cloud service provider in the development phase.

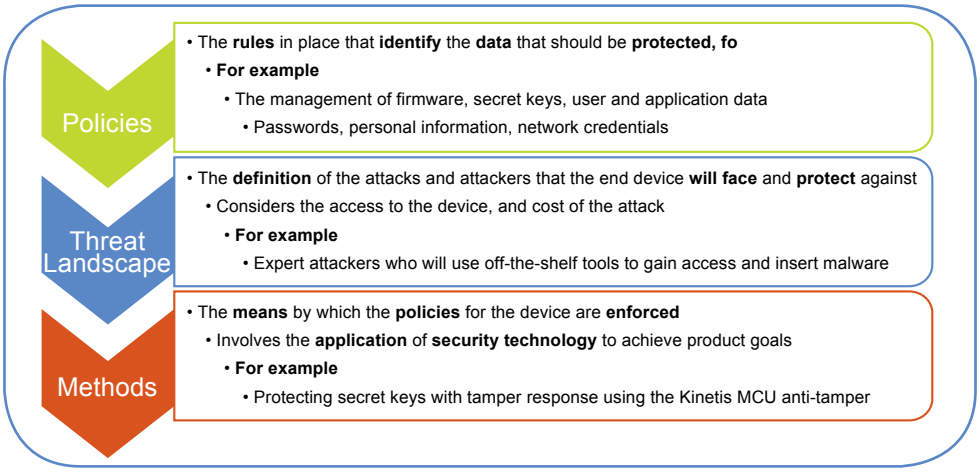
For the case of a controlled manufacturing site that is in a secure environment, the factory security tool is used only to sign application firmware. Then standard tools can be used to place the root of trust firmware and signed application firmware. Microcontroller security mechanisms are used to protect the root of trust firmware. For the scenario where a less-trusted manufacturing site is used, then the factory security tool could be deployed there. The factory security tool can interface to the cloud service provider securely to get the root of trust firmware. The foot of trust firmware must be securely placed on to the end device. Once the secure bootloader is on the end device, then the device will only accept and execute signed application code.

To implement such a lifecycle requires preset agreements with multiple parties, such as application code developers, external manufacturing sites, cloud service providers and component manufacturers. There are policies and audits which need to be in place. The complexities of lifecycle management create a demanding environment where the end developers must make use of a range of hardware security features, software components and partnership to achieve their security goals and prevent malicious firmware from being installed onto IoT edge node devices. The following sections provide details on the design for the factory security tool and root of trust firmware providing a secure bootloader which are central components to this view.

Security Model

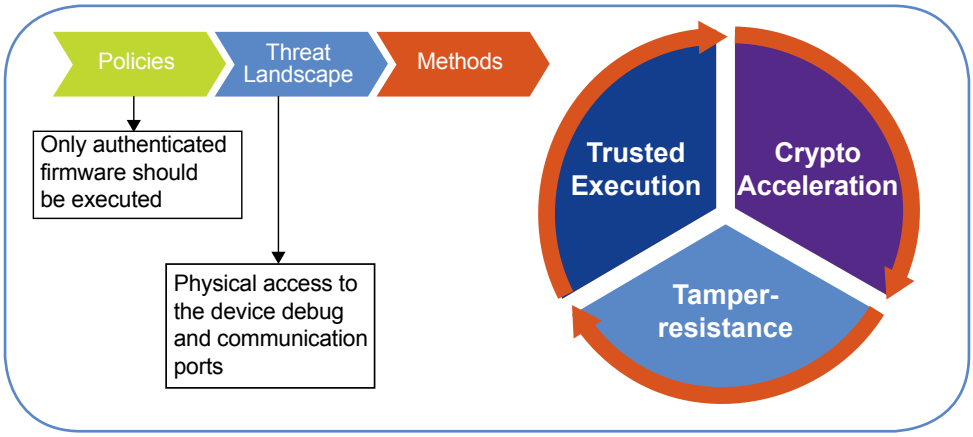
When designing a secure system, it is important to apply a security model. A security model is built from policies, the threat landscape and methods as shown in *Figure 2*. This model provides a framework for understanding and designing to the security goals of the device. The methods, or how the security policies are enforced to achieve product goals, are made possible by the security technology that is integrated into the embedded controllers such as NXP's Kinetis ARM Cortex-M MCUs.

Figure 2: Security Model



As an example, for the case of protecting firmware, a security model would be represented by what is shown in *Figure 3*. There is a policy that only authenticated firmware should ever be executed. The threat landscape typical for an IoT edge node is attackers will have physical access to the device and so its communication and debug ports could be exploited. Lastly, the methods that make use of microcontroller security technology supporting trust, cryptography and anti-tamper will be employed to enforce the security policy to the levels demanded by the threat landscape.

Figure 3: Secure Boot Security Model

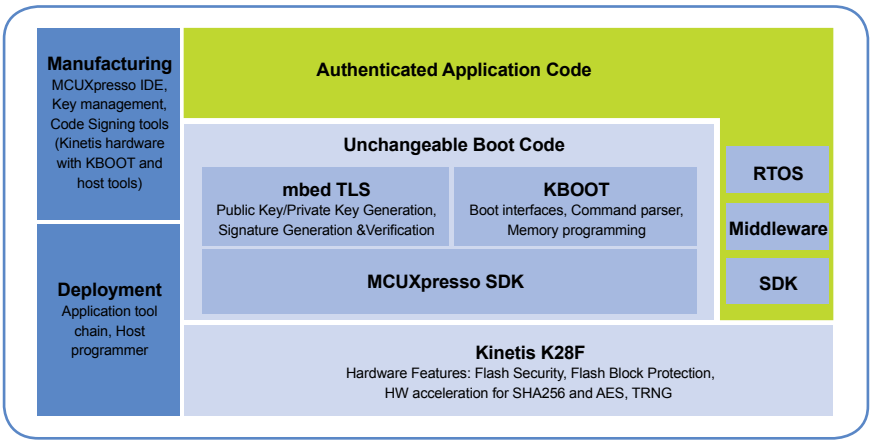


Devices will have different polices and face various threats as these aspects are application specific, but the underlying technology and the methods employed can be leveraged for many cases. The secure boot implementation addresses a wide range of security policies. As detailed in the following sections with the right hardware, software and tools, it can be achieved with limited impact to product cost and performance goals.

Secure Boot System Architecture

With a security model in place, the next stage of development is creating the secure boot system architecture. This provides the details of the components needed to achieve the secure boot with authentication of application firmware. *Figure 4* represents this system level view of the components and how they interact with one another.

Figure 4: Secure Boot System Architecture



At the base of *Figure 4* there is the hardware providing physical and logical security. This is where microcontroller capabilities are necessary to protect data, perform cryptography and monitor access to memories and peripherals. Sitting above the hardware must be unchangeable boot code. This code must always run when the device is powered. This boot code contains low level drivers to set up relevant security peripherals, a cryptography stack for authentication and or confidentiality of data and a way to load application code (a bootloader).

With the unchangeable boot code present on the right hardware, application code that is loaded on the device is authenticated upon every boot. Application code can be changed but the cryptographic authentication applied to the code by the secure boot ensures that the changes are only and always provided by a trusted entity. Application code can make use of all or a portion of the microcontroller resources as determined by the secure boot code. This is because upon boot, the secure boot code is always executed first, ensuring proper resource management.

Represented on the left of *Figure 4* are tools used in the manufacturing and deployment of the device. The microcontroller must be programmed, so tools for key management, creating firmware files and connecting and downloading firmware into the device are needed to implement the secure boot design. With these components considered the goal of authenticating application firmware upon every boot is achievable.

Applying NXP Kinetis MCU Security Technology with ARM mbed TLS

Kinetis ARM Cortex-M-based microcontrollers have been architected to support high security applications. As detailed in the **Kinetis MCUs security brochure**, numerous hardware features are supported across the hundreds of Kinetis MCU products, which can be enabled to support cryptography, trust and tamper resistance in the end device. For the purposes of the secure boot implementation, we will utilize features that are available on most Kinetis MCU devices. To assist the developer, the following sections will reference the **Kinetis K28_150MHz device**; a higher memory integration device offering 2MB of embedded flash and 1MB of embedded SRAM. The following sections will highlight the specific capabilities and configurations needed to achieve the secure boot with regards to hardware, software and tools.

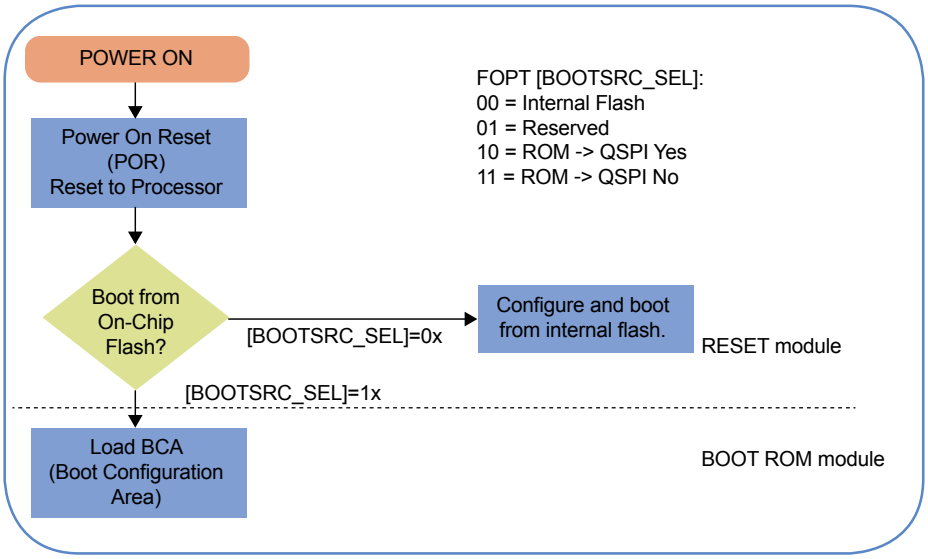
The Necessary Hardware Features

At the hardware level, there are several functions the microcontroller must support. These are controlling the boot flow of the device, protecting memory resources and making firmware immutable. The following section will detail how this is achieved for the Kinetis K28 150MHz device.

Control of Boot Flow

Kinetis MCUs are architected to boot up from internal memory. This protects against the threat of hijacking an embedded application by changing an external memory device. Some Kinetis devices like the K28 150MHz MCU have an internal ROM. For this implementation, the internal ROM is bypassed so that the trusted secure boot code can be customized using internal flash. This is done by setting non-volatile control register bits [BOOTSRC_SEL] as highlighted in *Figure 5* from **reference manual** section *7.3.4 Boot Sequence*. Once configured this way, the RESET module state machine of the K28_150MHz device will ensure that internal flash will be fetched and the secure boot code will always run.

Figure 5: Control of Boot Flow

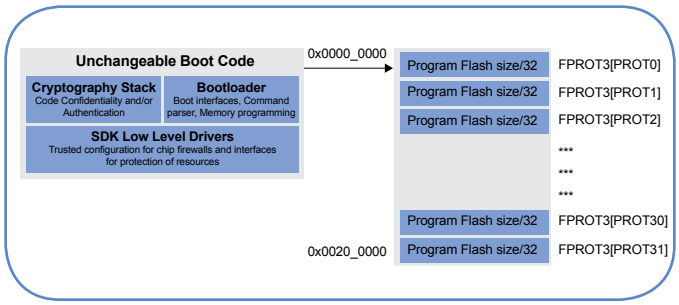


Flash Block Protection

As detailed in section 33.3.3.6 of the K28_150MHz **reference manual**, *"The FPROT registers define which program flash regions are protected from program and erase operations. Protected flash regions cannot have their content changed; that is, these regions cannot be programmed and cannot be erased…"*

The protected region size is chip specific as they are defined as program flash size divided by 32. In the case of a 2MB flash like the K28_150 device, these are 64KB blocks. This is substantial space for this secure boot implementation, but for smaller flash size devices, multiple blocks could be configured. As shown in *Figure 6*, the FPROT3[PROT0] control bit must be set and the unchangeable boot code placed at memory map location 0x0000_0000 to protect the secure boot code.

Figure 6: Using Flash Block Protection



Chip Security Setting

Once development of the secure boot code is completed, the chip security setting can be set to disable access from JTAG/SWD port and restrict data accesses to internal memory. See **reference manual** section *9.2 Flash Security*. The only allowable flash command once the security is enabled is the mass erase operation. This ensures that the data residing inside the chip cannot be read. Furthermore, the mass erase operation can also be disabled if the MEEN bit in the FSEC register is set to %01. See **reference manual** section *33.3.3.3 Flash Security Register (FTFE_FSEC)*.

Flash Configuration Field

The control registers for controlling boot flow, setting flash block protect and chip security settings are all part of a block of non-volatile registers as detailed in **section 33.3.1** *Flash Configuration Field Configuration*. As detailed in *Figure 7*, these registers are physically located in the memory map starting at address 0x0_400. These registers are also mirrored into peripheral registers to represent the settings that have been pre-configured. For the case of flash block protection (FPROT), the settings can be changed during run time to increase areas of protection, but never decrease protection. This allows the secure boot code to dynamically protect regions of flash by increasing areas of protection.

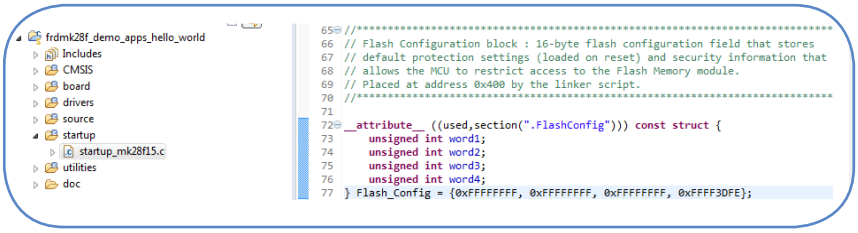
Figure 7: Flash Configuration Field

Flash Configuration Field Offset Address	Size (Bytes)	Field Description
0x0_0400 - 0x0_0407	8	Backdoor comparison key.
0x0_0408 - 0x0_040B	4	Program flash protection bytes. Refer to the description of the Program Flash Protection Registers (FPROT0-3).
0x0_040F	1	Reserved
0x0_040E	1	Reserved
0x0_040D	1	Flash nonvolatile option byte. Refer to the description of the Flash Option Register (FOPT).
0x0_040C	1	Flash nonvolatile option byte. Refer to the description of the Flash Security Register (FSEC).

Tools Interaction with Flash Configuration Field

When using the flash configuration field, it is possible to completely block any debug access and programming of the K28_150MHz device. The default configuration of integrated development environments and debugger tools is different across different tool chains. So, to achieve the hardware configuration needed, the specific toolchain must be configured to allow access. *Figure 8* represents where these settings can be done in the MCUXpresso, which is available at no cost, IDE using the **FRDM-K28F Freedom development board** with an integrated **CMSIS-DAP** debugger tool. Extreme care must be taken when using these fields. Extreme care must be taken when using these fields because the chip can be locked out in flash programming if the program image does not have these fields setup correctly.

Figure 8: Using MCUXpresso IDE



Software and Tools Resources

ARM mbed TLS

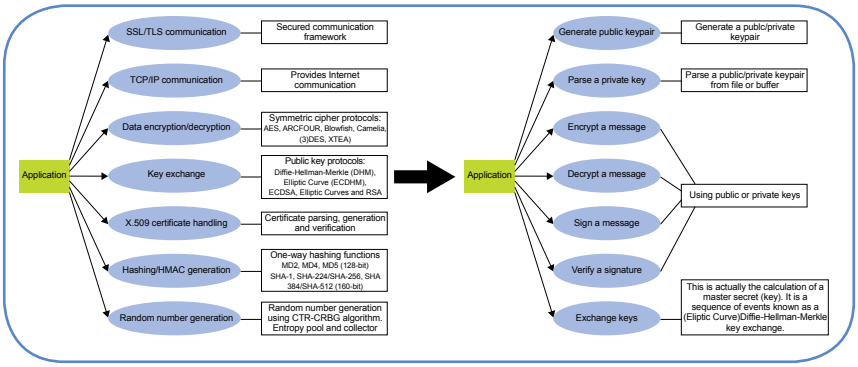
To satisfy the cryptography needed for the secure boot implementation, the solution uses the **MCUXpresso Software Development Kit (SDK)** configured with ARM mbed TLS support. The MCUXpresso SDK software abstracts the interface to the available hardware peripherals with a package consisting of peripheral drivers, middleware, board specific configurations and application code. Within the package there are many demo applications. For ARM mbed TLS, which is available at no cost, there are two demo applications that can be leveraged to gain a working knowledge of the software library. These are the test and benchmark applications.

When ARM mbed TLS support is ported onto Kinetis devices, the software is configured to make use of available microcontroller hardware resources. In the case of the K28_150MHz MCU, this is using the MMCAU cryptographic accelerator block that assist with AES, DES and hash operations.

Formerly Polar SSL, the ARM mbed TLS library is perfectly aligned to the needs of the secure boot development. The library is well documented and supported with numerous discussion forum post and code examples. The library is available as opens source under the Apache 2.0 license, which allows the code to be used in closed source projects. In addition, the library was created to be modular and with the consideration of the constraints of embedded systems allowing developers to fine tune their use of the library for the needs of specific applications.

As a representation of the alignment to our needs for secure boot, *Figure 9* details the main use cases for the library. As shown on the left, the library has modules related to key exchange. The specific capabilities provided by the public key module are represented on the right. Here we see the functions which we have introduced in the system architecture diagram (*Figure 4*) for generating a public key pair, signing a message, and verifying signatures. The hardware abstraction provided by these functions greatly eases the burden on the end developer for completing the necessary cryptographic operations.

Figure 9: ARM mbed TLS Design

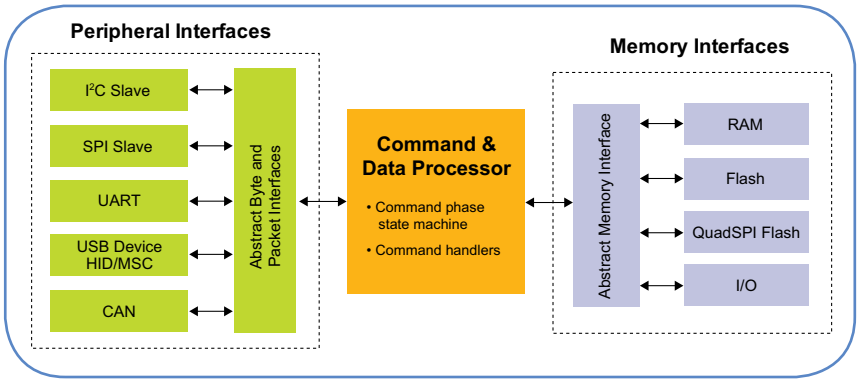


The ARM mbed TLS source files which are critical for an ECDSA implementation of the secure boot are: ec_curve.h, eccurve_config.h, ecdsa.h, ecdsa.c and ec_curve.c. Importing these files allows the end developer to make use of the ecdsa context structure defining the key information and the supporting APIs related to the ecdsa operations. Specifically, these APIs include ecdsa_genkey for public key generation. In addition, for transferring curve information ec_use_known_curve_param API is used. Depending on the lifecycle stage of the device, the ecdsa_sign and ecdsa_verify APIs are used. The curve selection is made in the eccurve_config.h file. Here you can see the options for a scalable security level based on the curves supported by mbed TLS. There is support for ECDSA curves ranging from SECP192 to SECP521.

KBOOT the Kinetis Bootloader

Providing the bootloader functions is the NXP Kinetis Bootloader product known as KBOOT. As shown in *Figure 10*, KBOOT embedded software consist of peripheral interfaces, a command and data processor and memory interfaces. KBOOT is provided as full source code and can be modified for end user needs.

Figure 10: KBOOT Block Diagram



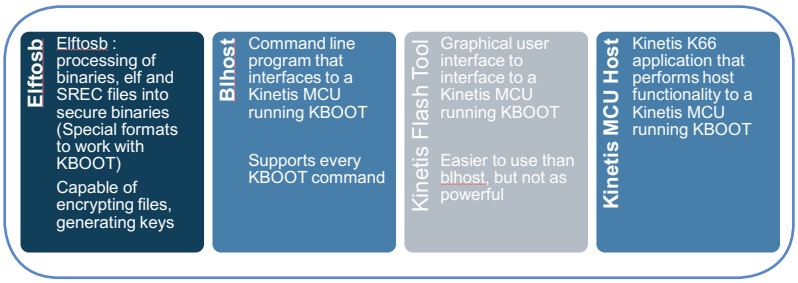
There are processor defines for configuring which peripheral interfaces should are enabled. This serves as a dual purpose as it allows for a way to optimize for code size and addresses security because it disables interfaces to the bootloader functions from unsupported peripheral interfaces. An example of how to use these defines is shown in the KBOOT **reference manual** section *11.6 Modifying a Peripheral Configuration Macro*. The reference manual also details the command API that is supported by the command and data processor block. In addition to the base commands for downloading firmware, the command API includes the ability to direct the device to execute firmware. This functionality is used in the factory setting to execute specific functions and extract signature and key data.

Depending on the end device, KBOOT supports provisioning for all available memory interfaces. For example, on the K28_150MHz MCU, in addition to RAM and Flash, KBOOT can manage the placement of data into external serial NOR flash via the QuadSPI interface.

KBOOT Tools

In addition to the KBOOT software which runs on the device, KBOOT also includes other tools packages that run on Linux®, Mac® or Windows® host machines. These are shown below in *Figure 11*.

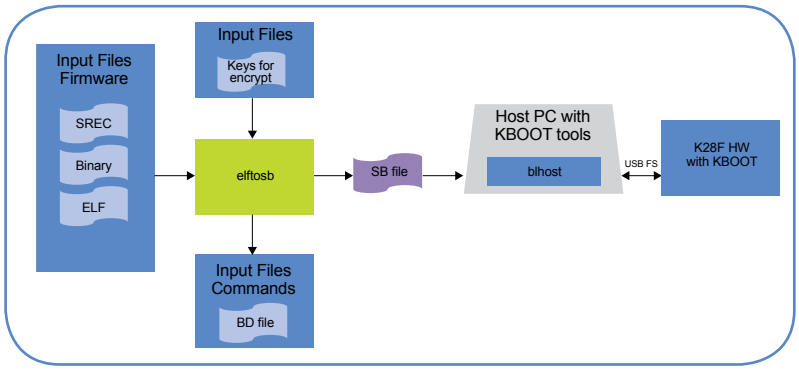
Figure 11: KBOOT Tools



For the processing of binaries, elf files and srecords there is a tool named *elftosb*. The elftosb tool takes commands from BD files. BD, short for boot descriptor file is an input command file used by elftosb to create secure binary files (sb file). The sb file contains commands and firmware data that is sent to the device that is running the KBOOT bootloader. The *blhost* tool is what is used to process the sb files and interface to the devices running KBOOT. Also, worth mentioning is the Kinetis Flash Tool and the Kinetis MCU host application but these are not used in this implementation.

Both elftosb and blhost are provided as source code and can be built for different operating systems. *Figure 12* shows a typical workflow for using the KBOOT tools. Moving from left to right, first the elftosb tool is used. Based on commands passed by a BD file, the elftosb tool takes input firmware files and creates the secure binary. With a secure binary, at a different time and place, a host PC running blhost tool can be used to provision a Kinetis microcontroller like the K28_150MHz device that is running KBOOT.

Figure 12: Typical KBOOT Workflow



Lifecycle with KBOOT Tools and Secure Boot

The following section relates the secure boot implementation and KBOOT tools to the lifecycle view introduced in *Figure 1*.

Development Stage

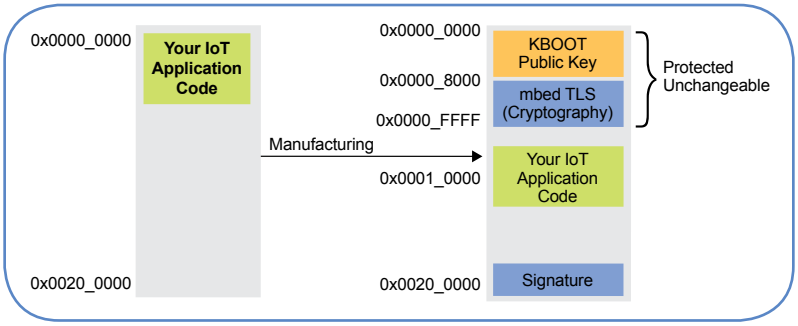
During the product development stage, there are two separate firmware developments which are done in the secure environment (please refer to *Figure 1*). Both developments are based on the software described in the previous sections, KBOOT and ARM mbed TLS.

The two developments are:

- Factory Security Tool Firmware
  - This bootloader application is for use in a secure manufacturing environment. The main security functions in addition to bootloader functions are to generate a PUB/PRIV key pair and to generate the signature for application code using the **private key**.
- Secure Boot Firmware
  - This bootloader application is for use in a deployed device. The main security functions in addition to bootloader functions are to check the signature of application code using the **public key**, and only allow execution of the application code if the signature is **authentic**.

The firmware for the Factory SecTool and Secure Boot is completely independent of application code development. Application code development can occur on a different target device, by different developers. As shown in *Figure 13* below, memory mapping on the left, this development can follow a traditional development flow for microcontrollers. During the manufacturing stage, the resulting firmware files can be relocated as shown on the memory mapping on the right to work with the secure boot firmware, which includes KBOOT and mbed TLS cryptography.

Figure 13: Memory Map for Application Development



Manufacturing Stage

After the application code has been audited versus security policy guidelines as shown in *Figure 1*, the following steps can be taken to complete the manufacturing of end devices that use a secure boot. Steps 1 and 2 are represented at the top of *Figure 14*, and you'll find steps 3 and 4 at the bottom.

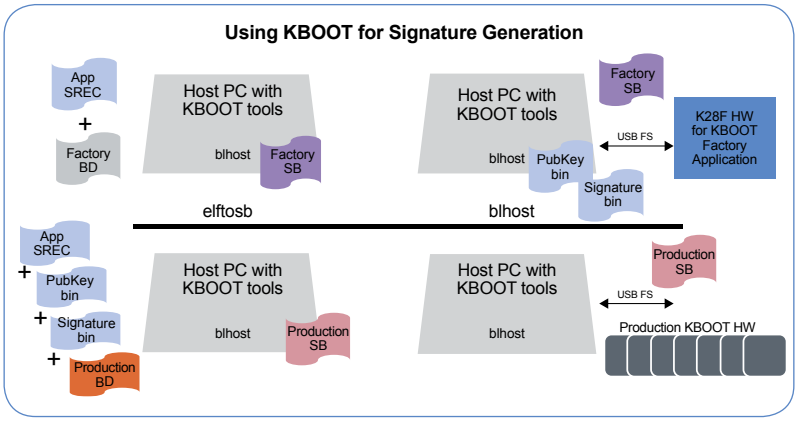
1) Application SREC is combined with Factory BD file to create the Factory Secure Binary (Factory.SB)

2) Using HW with the Factory Security Tool firmware, the Factory.sb is downloaded and blhost commands are used to extract binaries for signature and public keys.

3) Application SREC is combined with signature binary to make the Production secure binary (Production.sb)

4) Production secure binary is used to program final hardware

Figure 14: Manufacturing with KBOOT



Once a public key, private key pair is generated in steps 1 and 2, the programming of the production image can occur on all devices that will be protected by the same private key. Variations of this implementation can be made to address multiple key pairs and roll back protections. For example, multiple public key/private key pairs can be generated and stored onto the device during the manufacturing stage and then selected based on version settings.

A Foundation for Future Secure Embedded Systems

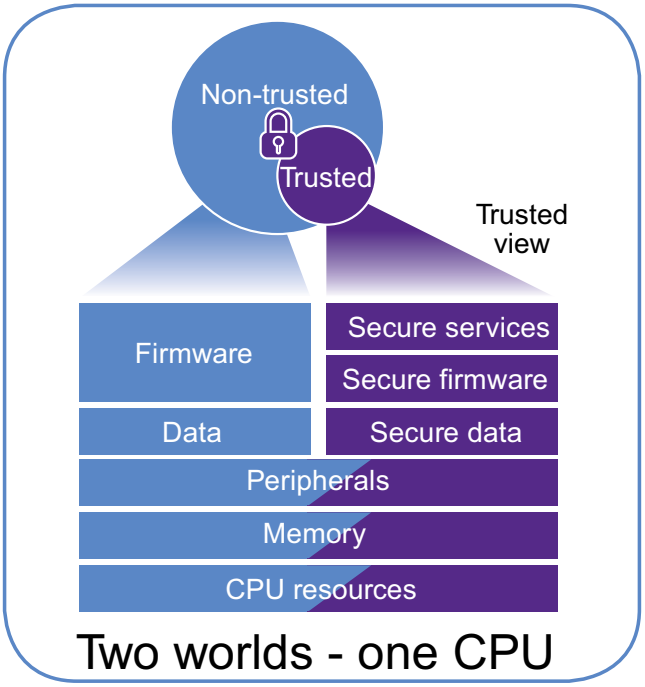
The principles and guidelines introduced in the previous sections provide a basis for secure embedded designs, but are not without challenges. Firstly, microcontroller resources (embedded flash memory) must be used. Future microcontroller devices could be improved. Integration of the secure boot capabilities into the ROM of the device would allow optimizations with regards to memory and development time. In addition, hardware acceleration for public key cryptography such as the ECDSA algorithm would reduce code size and boot times. Finally, most exciting is the availability of new processor architectures such as ARMv8-M.

ARM® TrustZone® for ARMv8-M Overview

TrustZone is a technology used by **ARM Cortex® processors** to implement system-wide isolation of assets in a system on chip (SoC). It is a widely-adopted technology that has been deployed in the market for over 10 years providing SoC security. TrustZone is the basis for protecting high-value applications such as mobile payments and digital rights management for media content. With the release of the **ARMv8-M architecture**, TrustZone was introduced into the ultra-low power, small area, real-time Cortex-M processors that power the **IoT**.

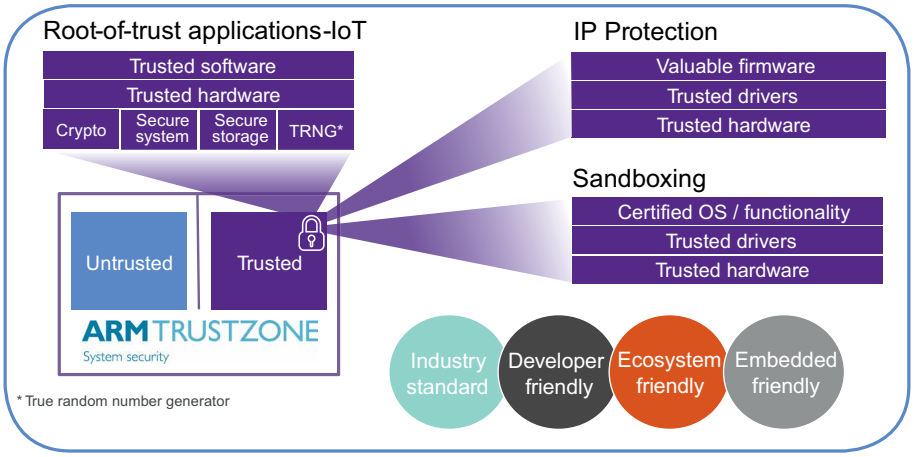
As represented in *Figure 15*, isolation in TrustZone for ARMv8-M is implemented in hardware by the processor, thus removing the need for a run-time software isolation layer. TrustZone for ARMv8-M reduces the number of functions or API calls required to make a transition from non-trusted firmware to trusted firmware. This dramatically improves the system efficiency.

Figure 15: Isolation of Resources



Using TrustZone for ARMv8-M, the system is split into two parts, one for the trusted firmware and the other for the user application. The trusted part cannot be accessed by the user side due to the hardware isolation. In the case that the user application is hacked via a software attack, then the trusted code is expected to recognize the unauthorized behavior and restore the user application code to a proper state. Given that these devices will live in the field for many years, and future attack methods are not known as of today, this is the best method to ensure a long, healthy life of an embedded solution.

Figure 16: TrustZone for ARMv8-M Use Cases



TrustZone® for ARMv8-M Use Cases

1. **Root of trust implementation**
   Connected devices with authentication requirements need a root of trust in the system architecture. This is particularly important for devices that can be updated over the air. In a system with TrustZone technology, code for firmware-update support and associated authentication can be placed in secure memory space, and hence, protected. Even if a device is compromised at the application level, the Root of Trust functionality cannot be altered and replaced with spurious firmware.
2. **Asset confidentiality management in IoT devices**
   Many IoT devices need to handle security-sensitive information, such as user details and security keys. TrustZone technology allows this information and associated firmware (that can have direct access to this data) to be stored in protected, secure memory space. The architecture design enables the application code running in non-secure state access to the secure information via predefined APIs only (and if provided in the secure software, via an authentication process).
3. **Firmware protection**
   Firmware shipped with the device includes valuable IP that needs to be protected. TrustZone technology enables IP protection by allowing the supplier to put their firmware in protected, secure memory space, while still allowing users to use the firmware via predefined API calls.
4. **Sandboxing for devices with certified software**
   Many devices, such as a wireless chipset, contain preloaded software, yet also allow developers to augment functionality by adding their own application software components. Using TrustZone technology, the preloaded firmware can be placed in the secure side and its behavior prevented from being altered or compromised by applications running on the non-secure side. This helps ensure that certified firmware fulfills its mission as originally certified. In addition, placing the firmware in the secure side helps protect it from being reverse-engineered.

The hi-tech industry never sleeps. The drive towards lower power and higher performance efficiency continues. For future embedded solutions, TrustZone for ARMv8-M is the foundation for a more efficient and developer-friendly security solution. Find out more here: **https://community.arm.com/processors/trustzone-for-armv8-m/**.

Conclusions

In today's connected world, the protection of firmware is an essential component to delivering solutions that safeguard device manufacturers and their customers. Essential to sustaining end-to-end security is a secure and trusted boot, which can be achieved with the right MCU hardware capabilities and ARM mbed TLS. NXP's microcontrollers contain the hardware features and software enablement that can be integrated to strengthen end device security and protect value. As the drive towards lower power and higher performance efficiency for IoT edge nodes continues, future capabilities in embedded controllers and ARM processors will provide the basis for future security solutions for the IoT.

References

http://www.nxp.com/docs/en/reference-manual/KBTLDR200RM.pdf
http://www.nxp.com/docs/en/reference-manual/K28P210M150SF5RM.pdf
https://tls.mbed.org/high-level-design
https://tls.mbed.org/module-level-design-public-key
https://community.arm.com/processors/trustzone-for-armv8-m/

Document Number: SECURITYWP REV 0