

## IoT Security

[Using IoT Security Features](#)

[Getting Started](#)

[IoT Endpoint Security Fundamentals](#)

[Introduction](#)

[Overview](#)

[No Universal Passwords](#)

[Secured Interfaces](#)

[Proven Cryptography](#)

[Security by Default](#)

[Signed Software Updates](#)

[Automatically Applied Updates](#)

[Vulnerability Reporting Program](#)

[Security Expiration Date](#)

[Next Steps](#)

[Series 2 Device Security Features](#)

[Developer's Guide](#)

[Overview](#)

[Series 2 Secure Debug](#)

[Introduction](#)

[Series 2 Device Security Features](#)

[Introduction To Secure Debug](#)

[Secure Engine Subsystem](#)

[Debug Lock](#)

[Debug Unlock](#)

[Examples](#)

[Precautions](#)

[Failure Analysis](#)

[Series 2 TrustZone](#)

[Introduction](#)

[Series 2 Device Security Features](#)

[TrustZone Basics](#)

[Bus Level Security \(BLS\)](#)

[Secure And Privileged Programming Model](#)

[TrustZone Implementation](#)

[Upgrade Existing Application To TrustZone](#)

[TrustZone Platform Examples](#)

[Production Programming of Series 2 Devices \(PDF\)](#)

[Anti-Tamper Protection Configuration and Use](#)

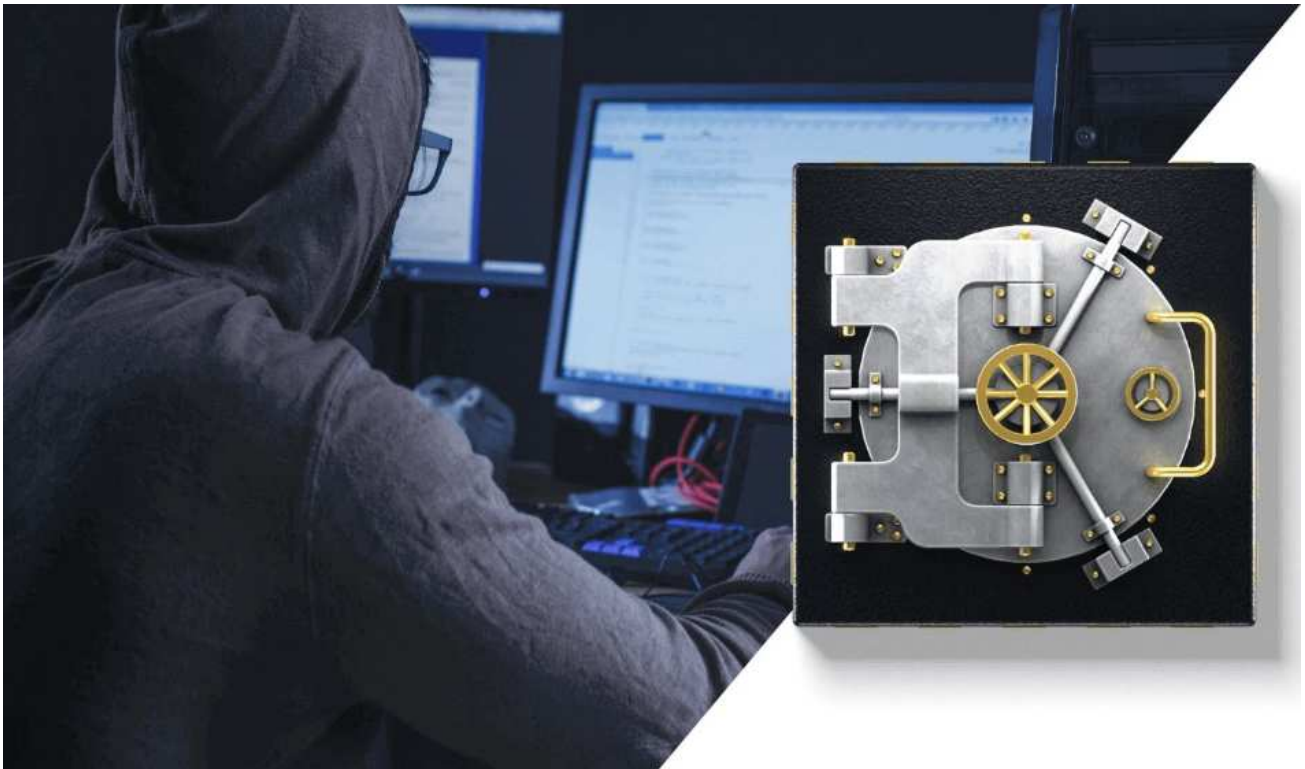
[Overview](#)[Series 2 Device Security Features](#)[Introduction](#)[Secure Engine Manager](#)[Tamper Responses](#)[Tamper Sources](#)[Anti-Tamper Configuration](#)[Usage Example](#)[Tamper Disable](#)[Examples](#)[Authenticating Silicon Labs Devices using Device Certificates](#)[Overview](#)[Series 2 Device Security Features](#)[Introduction](#)[Secure Identification on HSE-SVH Devices](#)[Device Certificate Options](#)[Entity Attestation Token \(EAT\)](#)[Remote Authentication Process](#)[Secure Engine Manager](#)[Examples](#)[Secure Key Storage](#)[Overview](#)[Series 2 Device Security Features](#)[Introduction](#)[HSE Secure Key Storage](#)[TrustZone Secure Key Storage](#)[Secure Key Storage Implementations](#)[Examples](#)[Programming Series 2 Devices Using the DCI and SWD \(PDF\)](#)[Integrating Crypto Functionality with PSA Crypto vs. Mbed TLS \(PDF\)](#)[Protocol-Specific Information](#)[Production Guide](#)[Overview](#)[Custom Part Manufacturing Service](#)[Overview](#)[SE Firmware Version](#)[Debug Lock Settings](#)[Secure Boot with RTSL Settings](#)[Tamper Response](#)[Standard Security Keys](#)[Additional Custom Keys](#)[Custom Certificates](#)[Configure Device for Untrusted Environment Example](#)[Import Custom Wrapped Keys Example](#)

PKI Recommendations

## Using IoT Security Features

# Using Silicon Labs IoT Security Features

Silicon Labs offers a range of security features depending on the part you are using and your application and production needs.



The content on these pages is intended for those who want to implement security features as part of your IoT device management. If you are looking for an introduction to Silicon Labs Security features and to security issues that confront those implementing IoT systems, see the [Silabs.com Security page](#).

**For details about this release:** Links to release notes are available on the [silabs.com Gecko SDK](#) page as part of the Gecko Platform release notes.

**For background on security issues in general:** [IoT Security Fundamentals](#) explains some security basics.

**To get started with implementing security:** See the [Getting Started page](#) for help determining what features you want to implement based on the series 2 part you are working with. Series 2 devices are the preferred choice for secure system implementation.

**If you are already in development:** See the [Developer's Guide](#) for details. Security APIs are documented in the [Gecko Platform API Reference](#).

**For detailed information about implementing some security features with specific protocols:** See the [protocol-specific pages](#). An extensive body of other protocol-specific content can be accessed through the [docs.silabs.com homepage](#).



## Getting Started

# Getting Started with Silicon Labs IoT Security Features on Series 2 Devices

Protecting IoT devices against security threats is central to a quality product. Silicon Labs offers several security options to help developers build secure devices, secure application software, and secure paths of communication to manage those devices. Silicon Labs' security offerings were significantly enhanced by the introduction of the Series 2 products that included a Secure Engine. The Secure Engine is a tamper-resistant component used to securely store sensitive data and keys, and to execute cryptographic functions and secure services.

On Series 1 devices, the security features are implemented by the TRNG (if available) and CRYPTO peripherals.

On Series 2 devices, the security features are implemented by the Secure Engine and CRYPTOACC (if available). The Secure Engine may be hardware-based or virtual (software-based). Here the following abbreviations are used:

- HSE - Hardware Secure Engine
- VSE - Virtual Secure Engine
- SE - Secure Engine (either HSE or VSE)

Additional security features are provided by Secure Vault. Three levels of Secure Vault feature support are available, depending on the part and SE implementation, as reflected in the following table:

Security Level (1)	SE Support	MCU	Wireless SoC (2)
Secure Vault Base (SVB)	N/A	EFM32JG1, EFM32PG1, EFM32JG12, EFM32PG12, EFM32GG11, EFM32GG12, EFM32TG11	EFR32xG1, EFR32xG12, EFR32xG13, EFR32xG14
Secure Vault Mid (SVM)	VSE (VSE-SVM)	EFM32PG22	EFR32xG22
"	HSE (HSE-SVM)	EFM32PG23A	EFR32xG21A, EFR32xG23A, EFR32xG24A
Secure Vault High (SVH)	HSE only (HSE-SVH)	EFM32PG23B	EFR32xG21B, EFR32xG23B, EFR32xG24B

Note:

1. The features of different Secure Vault levels can be found in <https://www.silabs.com/security>.
2. The x is a letter B, F, M, or Z.

Secure Vault Mid consists of two core security functions:

- Secure Boot: Process where the initial boot phase is executed from an immutable memory (such as ROM) and where code is authenticated before being authorized for execution.
- Secure Debug access control: The ability to lock access to the debug ports for operational security, and to securely unlock them when access is required by an authorized entity.

Secure Vault High offers additional security options:

- Secure Key Storage: Protects cryptographic keys by "wrapping" or encrypting the keys using a root key known only to the HSE-SVH.
- Anti-Tamper protection: A configurable module to protect the device against tamper attacks.
- Device authentication: Functionality that uses a secure device identity certificate along with digital signatures to verify the source or target of device communications.

A Secure Engine Manager and other tools allow users to configure and control their devices both in-house during testing and manufacturing, and after the device is in the field.

Silicon Labs strongly recommends installing the latest SE firmware on Series 2 devices to support the required security features. The latest SE firmware image (.seu and .hex) and release notes can be found in these Windows folders of the GSDK.

```
C:\Users\<UserName>\SimplicityStudio\SDKs\gecko_sdk\util\se_release\public
```

If you have not already installed the GSDK, instructions for doing so with Simplicity Studio are available in the [Getting Started section of the Simplicity Studio 5 User's Guide](#).

Refer to *AN1222: Production Programming of Series 2 Devices* for guidance on the SE firmware upgrade procedure. The latest SE firmware shipped with Series 2 devices and modules (if available) at the time of this writing are listed in the following table:

MCU Series 2 and Wireless SoC Series 2	SE	Shipped SE Firmware Version (Device and Module)
EFR32xG21A	HSE-SVM	1.2.13
EFM32PG23A	HSE-SVM	2.1.7
EFR32xG23A	HSE-SVM	2.1.2 (Rev B), 2.1.7 (Rev C)
EFR32xG24A	HSE-SVM	2.1.7
EFR32xG21B	HSE-SVH	1.2.13
EFM32PG23B	HSE-SVH	2.1.7
EFR32xG23B	HSE-SVH	2.1.2 (Rev B), 2.1.7 (Rev C)
EFR32xG24B	HSE-SVH	2.1.7
EFM32PG22 and EFR32xG22	VSE-SVM	1.2.12

In support of these products Silicon Labs offers whitepapers, webinars, and documentation. The following table summarizes the key security documents:

Document	Summary	Applicability
<a href="#">Series 2 Secure Debug</a>	How to lock and unlock Series 2 debug access, including background information about the Secure Engine	Series 2
<a href="#">Series 2 Secure Boot with RTSL</a>	Describes the secure boot process on Series 2 devices using Secure Engine. For information on bootloading with Silicon Labs products, see <a href="#">Silicon Labs Gecko Bootloader User's Guide for GSDK 4.0 and Higher (series 1 and 2 devices)</a>	Series 2
<a href="#">Anti-Tamper Protection Configuration and Use</a>	How to program, provision, and configure the anti-tamper module	Series 2 with SVH
<a href="#">Authenticating Silicon Labs Devices using Device Certificates</a>	How to authenticate a device using secure device certificates and signatures, at any time during the life of the product	Series 2 with SVH
<a href="#">Secure Key Storage</a>	How to securely “wrap” keys so they can be stored in non-volatile storage	Series 2 with SVH
AN1222: Production Programming of Series 2 Devices	How to program, provision, and configure security information using Secure Engine during device production	Series 2
AN1303: Programming Series 2 Devices Using the Debug Challenge Interface (DCI) and Serial Wire Debug (SWD)	How to provision and configure Series 2 devices through the DCI and how to program their internal flash memory through the SWD	Series 2
AN1311: Integrating Crypto Functionality Using PSA Crypto Compared to Mbed TLS	How to integrate crypto functionality into applications using Silicon Labs implementation of PSA Crypto compared to Mbed TLS	Series 1 and Series 2



## IoT Endpoint Security Fundamentals

# IoT Endpoint Security Fundamentals

NOTE: This section replaces *UG103.05: IoT Endpoint Security Fundamentals*. Further updates to this user guide will be provided here.

This guide introduces the security concepts that must be considered when implementing an Internet of Things (IoT) system. Using the ioXt Alliance's eight security principles as a structure, this guide clearly delineates the solutions Silicon Labs provides to support endpoint security and what you must do outside of the Silicon Labs framework. Where appropriate, Silicon Labs' approach to our own security is offered as an example. This guide is designed for product developers and managers.

Silicon Labs' *Fundamentals* series covers topics that project managers, application designers, and developers should understand before beginning to work on an embedded networking solution using Silicon Labs chips, networking stacks such as EmberZNet PRO or Silicon Labs Bluetooth, and associated development tools. These guides can be used as a starting place for anyone needing an introduction to developing wireless networking applications, or who is new to the Silicon Labs development environment.

## Overview

# Overview

Securing the IoT is challenging. It is also mission-critical. Threats are continuously evolving, and the demand on product developers to keep up can be burdensome – particularly in low-cost, resource-constrained IoT products. Protecting your product in a connected world is a necessity, as customer data and modern online business models are increasingly targets for costly hacks that jeopardize end-user privacy and corporate brand damage. Silicon Labs is committed to working with the security community, customers, and other experts to bring state-of-the-art technology to help protect your connected portfolio.

Silicon Labs is a member of the ioXt (Internet of Secure Things) Alliance. The ioXt Alliance was formed to bring together wireless carriers, leading consumer product manufacturers, standards groups, compliance labs and government organizations to align baseline security requirements, to set the stage for testing and compatibility certification, and to work together building global standards for the IoT world.



The ioXt alliance has produced the ioXt Security Pledge (<https://www.ioxtalliance.org/s/ioXt-SecurityPledge-booklet-final.pdf>) The pledge covers eight principles in the areas of Security, Upgradability, and Transparency. Silicon Labs has adopted these principles in our own operations as well as in the products we provide. Our approach to these principles is described in this document.



The above image and all pledge language is reproduced from *The ioXt Security Pledge: 8 Principles for Consumer Product Design and Manufacturing to Ensure Security, Upgradability & Transparency (2019)*.

## No Universal Passwords

# No Universal Passwords

*The product shall not have a universal password; unique security credentials will be required for operation. Universal passwords allow an attacker to easily gain access to any device. Therefore, products shall either have a unique password or require the user to enter a new password immediately upon first use.*

It is your responsibility to ensure that your product enforces the creation of a unique password before activation.

Silicon Labs' products are designed to be configured by the manufacturer before being delivered to customers, and therefore passwords are outside of our scope. However, Silicon Labs tools are designed to support the various levels of security provided by the protocol in question. Most protocols offer different security levels, with tradeoffs between security level and other features such as ease of network formation. You need to review and decide on the level required by your application. For example:

- The EmberZNet Pro SDK supports a highly secure centralized trust-center-controlled method that replaces a device's factory-programmed link key with a key that is unique to each device on the network.
- Z-Wave 700 products come with a factory-programmed unique S2 keypair on first power-up, and support SmartStart commissioning through a package QR code containing the public key.
- Bluetooth options range from an unsecured "Just Works" approach to a LE Secure Connections Pairing model. Application designers can implement additional device authentication methods, such as through the companion smartphone app, to help ensure secure pairing even for devices without a user interface.

## Secured Interfaces

# Secured Interfaces

*All product interfaces shall be appropriately secured by the manufacturer.*

The interfaces to be secured will vary by product configuration. For example, in an NCP topology the NCP interface must be secured. Debug interfaces should always be locked. Wireless interfaces should be secured by using strong pairing and commissioning methods and by enabling encrypted and authenticated transmissions.

While securing the interfaces is in the end your responsibility, Silicon Labs provides the tools to enable that security.

Both Series 1 and Series 2 devices are designed to support securing debug access. For Series 1 devices, that functionality is provided through writing a Debug Lock word to the device. Unlocking the device erases the main application and the key material stored in the Lockbits page. For Series 2 devices, securing debug access is done through the device's Secure Engine. Both allow the developer to lock the debug port itself. See [Silicon Labs Gecko Bootloader User's Guide for Series 3 and Higher](#), [Silicon Labs Gecko Bootloader User's Guide for GSDK 4.0 and Higher \(series 1 and 2 devices\)](#), or [UG266: Silicon Labs Gecko Bootloader User's Guide for GSDK 3.2 and Lower](#) for an overview of securing debug access, and [Series 2 Secure Debug](#) for details on the Series 2 implementation. [UG104: Testing and Debugging Applications for the Silicon Labs EFR32MG Platforms](#) provides an overview of the various application testing stages and the debug access (hardware and software) required in each.

For more information on Wireless interface security in the different protocols, see the following:

- [Zigbee Security](#)
- [Bluetooth LE Fundamentals](#) and relevant KBAs
- [AN1037: Apple HomeKit Over Bluetooth®](#)
- [UG235.03: Architecture of the Silicon Labs Connect Stack v2.x](#)
- [UG435.03: Architecture of the Silicon Labs Connect Stack v3.x](#)



## Proven Cryptography

# Proven Cryptography

*Product security shall use strong, proven, updatable cryptography using open, peer-reviewed methods and algorithms.*

An important aspect of any IoT device is how secure the device is when it communicates with other devices, gateways, or the cloud. This standard mandates using proven cryptographic methods rather than attempting to implement your own.

Developers commonly secure communications such as TCP/IP connections, Bluetooth, Zigbee, or Z-Wave using the standardized and proven cryptographic methods native to the protocol. However, if a microcontroller sends sensitive information over a simple interface such as a UART to another microcontroller, it is important to realize that data should also be secured to prevent someone from snooping the UART line.

Silicon Labs offers a hardware CRYPTO module that provides an efficient acceleration of common cryptographic operations and allows these to be used efficiently with low CPU overhead. The CRYPTO module includes hardware accelerators for the Advanced Encryption Standard (AES), Secure Hash Algorithm SHA-1 and SHA-2 (SHA-224 and SHA-256), and modular multiplication used in ECC (Elliptic Curve Cryptography) and GCM (Galois Counter Mode). The CRYPTO module can autonomously execute and iterate a sequence of instructions to aid software and speed up complex cryptographic functions like ECC, GCM, and CCM (Counter with CBC-MAC).

In addition to the CRYPTO module, Silicon Labs includes mbed TLS as part of the Gecko Platform SDK. mbed TLS is open source software licensed by ARM Limited. It provides an SSL library that makes it easy to use cryptography and SSL/TLS in applications. mbed TLS supports software implementations of all crypto algorithms that are supported by TLS 1.2 as well as a build API that allows hardware drivers to replace the software implementations when cipher accelerators are supported by the platform. Its modular framework allows for subcomponents like the crypto libraries to be incorporated into a design independently of the SSL/TLS components, saving valuable code space and runtime RAM. mbed TLS supports SSLv3 up to TLSv1.2 communication by providing the following:

- TCP/IP communication functions: listen, connect, accept, read/write.
- SSL/TLS communication functions: init, handshake, read/write.
- X.509 functions: CRT, CRL and key handling
- Random number generation
- Hashing
- Encryption/decryption

These functions are split up into logical interfaces. They can be used separately to provide any of the above functions or to mix-and-match into an SSL server/client solution that utilizes a X.509 PKI. Examples of such implementations are provided with the source code. Components or plugins and APIs provide configuration interfaces accessible through the various SDK installations.

For more information, see the latest MCU and Peripheral Software Documentation for the target part at <https://docs.silabs.com>.

## Security by Default

# Security by Default

*Product security shall be appropriately enabled by default by the manufacturer.*

The state in which a product is shipped is up to the manufacturer. This standard mandates that any security features provided with the product be enabled before shipping. Customers should not have to turn security on; rather they should actively have to disable it. For example, Silicon Labs Z-Wave end-nodes and gateway SDKs ship with S2 cryptography and SmartStart network formation enabled by default.

Silicon Labs believes that product security should be considered during product design, and not as an afterthought. Within development environments, all Silicon Labs application security features may be enabled or disabled as appropriate during application development. Security must also be considered during device design and testing. [Bringing Up Custom Devices for the EFR32MG and EFR32FG Families](#) describes the security tokens (keys, certificates, and so on) that can be programmed into a custom device to support various types of security, including that provided by the Gecko Bootloader (see [Signed Software Updates](#)).

## Signed Software Updates

# Signed Software Updates

*The product shall only support signed software updates. While it is critical that all products be updatable, it is just as critical that these update images be secured. A manufacturer must cryptographically sign update images to prevent tampering during deployment. The product must not use unsigned updates, as they could be fraudulent.*

Silicon Labs development tools support building signed upgrade images and securely updating devices in the field, through the Silicon Labs Gecko Bootloader. The Gecko Bootloader can be configured to perform a variety of functions, from device initialization to firmware upgrades. Key features of the bootloader are:

- Useable across Silicon Labs Gecko microcontroller and wireless microcontroller families
- In-field upgradeable
- Configurable
- Enhanced security features, including:
  - Secure Boot: When Secure Boot is enabled, the bootloader enforces cryptographic signature verification of the application image on every boot, using asymmetric cryptography. This ensures that the application was created and signed by a trusted party.
  - Signed upgrade image file: The Gecko Bootloader supports enforcing cryptographic signature verification of the upgrade image file. This allows the bootloader and application to verify that the application or bootloader upgrade comes from a trusted source before starting the upgrade process, ensuring that the image file was created and signed by a trusted party.
  - Encrypted upgrade image file: The image file can also be encrypted to prevent eavesdroppers from acquiring the plaintext firmware image.

On Series 1 devices, the Gecko Bootloader has a two-stage design, first stage and main stage, where a minimal first stage bootloader is used to upgrade the main bootloader. The first stage bootloader only contains functionality to read from and write to fixed addresses in internal flash. To perform a main bootloader upgrade, the running main bootloader verifies the integrity and authenticity of the bootloader upgrade image file. The running main bootloader then writes the upgrade image to a fixed location in internal flash and issues a reboot into the first stage bootloader. The first stage bootloader verifies the integrity of the main bootloader firmware upgrade image, by computing a CRC32 checksum before copying the upgrade image to the main bootloader location.

On Series 2 devices, the Gecko Bootloader consists only of the main stage bootloader. The main bootloader is upgradable through the hardware peripheral Secure Engine. The Secure Engine provides functionality to install an image to address 0x0 in internal flash, by copying from a configurable location in internal flash. To perform a main bootloader upgrade, the running main bootloader verifies the integrity and authenticity of the bootloader upgrade image file. The running main bootloader then writes the upgrade image to the upgrade location in flash and requests that the Secure Engine install it. The Secure Engine is also capable of verifying the authenticity of the main bootloader update image against a root of trust. The Secure Engine itself is upgradable using the same mechanism.

In summary, Series 2 devices support a hardware root of trust and a Secure Boot process that verifies the authenticity and integrity of Gecko Bootloader, whereas in Series 1 devices, the authenticity and integrity of Gecko Bootloader are assumed trusted and are not explicitly checked.

The Gecko Bootloader can enforce application image security on two levels:

- Secure Boot refers to the verification of the authenticity of the application image in main flash on every boot of the device. When Secure Boot is enabled, the cryptographic signature of the application image in flash is verified on every boot, before the application is allowed to run. Secure Boot is not enabled by default in the example configurations provided by Silicon Labs, but enabling it is highly recommended to ensure the validity and integrity of firmware images.
- Secure Firmware Upgrade refers to the verification of the authenticity of an upgrade image before performing a bootload, and optionally enforcing that upgrade images are encrypted. The Secure Firmware Upgrade process uses symmetric

encryption to encrypt the upgrade image, and asymmetric cryptography to sign the upgrade image in order to ensure its integrity and authenticity.

For more information on Silicon Labs' support for software update security, refer to the following:

Bootloaders in general: [Bootloader Fundamentals](#)

The Gecko Bootloader in general: [Silicon Labs Gecko Bootloader User's Guide for Series 3 and Higher](#), [Silicon Labs Gecko Bootloader User's Guide for GSDK 4.0 and Higher \(series 1 and 2 devices\)](#), or *UG266: Silicon Labs Gecko Bootloader User's Guide for GSDK 3.2 and Lower*.

Using the Gecko Bootloader with specific protocols:

- [Using the Gecko Bootloader with EmberZNet](#)
- [Using the Gecko Bootloader with Silicon Labs Connect](#)
- [Using the Gecko Bootloader with Silicon Labs Bluetooth Applications](#)

Secure Boot on Series 2 devices: [Series 2 Secure Boot with RTSL](#)

# Automatically Applied Updates

## Automatically Applied Updates

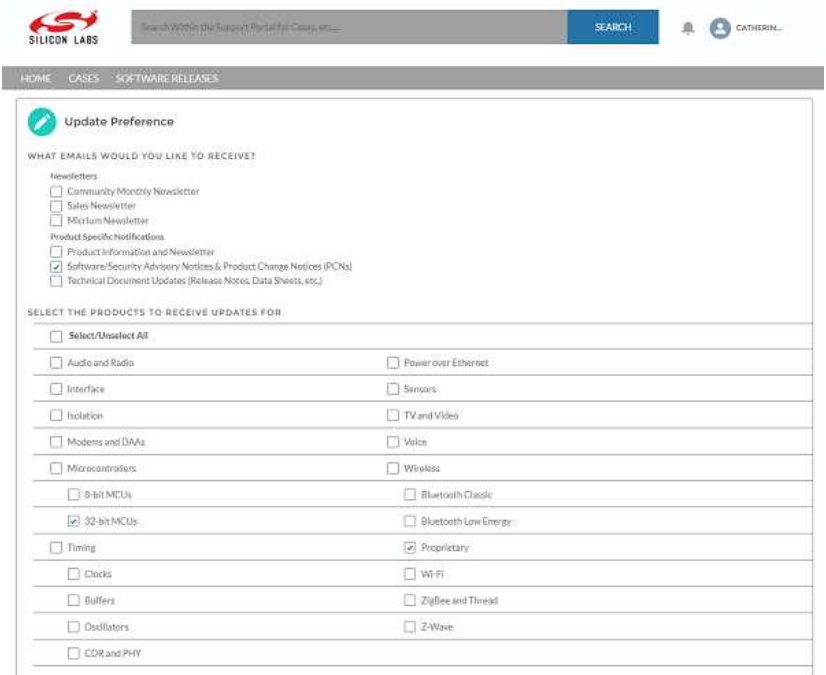
*The manufacturer will act quickly to apply timely security updates. Whenever a security vulnerability is detected, the manufacturer will automatically apply a patch to the product. No user intervention will be required.*

It is the manufacturer's responsibility to develop and implement automatic security updates. The design and methodology of such systems, for example through a Cloud-connected infrastructure or by direct intervention by a service representative, is up to you.

Silicon Labs will notify you of any security-related updates, as described in [Vulnerability Reporting Program](#). Your responsibility is to evaluate the level of risk that vulnerability poses for your particular product and to integrate the update into your platform as appropriate so that your end users are protected. Updated components might include the protocol libraries, Secure Engine firmware inside the Series 2 family, or an SDK module such as the Gecko Bootloader that enforces secure OTA updates and secure boot functionality.

Silicon Labs recommends the following:

- Subscribe to security updates through our Salesforce portal. To review or change your subscriptions, log in to the portal, click **HOME** to go to the portal home page and then click the **Manage Notifications** tile. Make sure that **Software/Security Advisory Notices & Product Change Notices (PCNs)** is checked, and that you are subscribed at minimum for your platform and protocol. Click **Save** to save any changes.



The screenshot shows the 'Update Preference' form in the Silicon Labs Salesforce portal. The form is titled 'Update Preference' and has a sub-header 'WHAT EMAILS WOULD YOU LIKE TO RECEIVE?'. It contains two main sections: 'Newsletters' and 'Product Specific Notifications'. Under 'Newsletters', there are three checkboxes: 'Community Monthly Newsletter', 'Sales Newsletter', and 'Mixium Newsletter'. Under 'Product Specific Notifications', there are three checkboxes: 'Product Information and Newsletter', 'Software/Security Advisory Notices & Product Change Notices (PCNs)' (which is checked), and 'Technical Document Updates (Release Notes, Data Sheets, etc.)'. Below these sections is a section titled 'SELECT THE PRODUCTS TO RECEIVE UPDATES FOR:'. It contains a table with two columns of checkboxes for various product categories. The categories are: Audio and Radio, Interface, Isolation, Modems and DAAs, Microcontrollers, 8-bit MCUs, 32-bit MCUs (checked), Timing, Clocks, Buffers, Oscillators, CDR and PHY, Power over Ethernet, Sensors, TV and Video, Voice, Wireless, Bluetooth Classic, Bluetooth Low Energy (checked), Proprietary (checked), Wi-Fi, ZigBee and Thread, and Z-Wave.

- Do not turn off Simplicity Studio's update notification. Within Simplicity Studio, you can download updates and easily access product release notes.

## Vulnerability Reporting Program

# Vulnerability Reporting Program

*The manufacturer shall implement a vulnerability reporting program, which will be addressed in a timely manner. All companies that offer Internet-connected devices and services shall provide a public point of contact as part of a vulnerability disclosure policy in order that security researchers and others are able to report issues. Disclosed vulnerabilities should be acted on in a timely manner.*

Manufacturers are responsible for implementing their own program. For any individual vulnerability, you will need to weigh the value of transparency with your customers against the risk of malicious use of the information to exploit a vulnerability before it can be addressed. Silicon Labs makes similar decisions about how broadly to report security vulnerabilities discovered in our products.

Silicon Labs customers and security researchers can report security vulnerabilities in Silicon Labs hardware and software products on the Silicon Labs website: <https://www.silabs.com/security/product-security>.


Silicon Labs' Security Vulnerability Disclosure Policy may be found here:

[https://www.silabs.com/documents/public/miscellaneous/PS1012-Security\\_Vulnerability\\_Disclosure\\_Policy.pdf](https://www.silabs.com/documents/public/miscellaneous/PS1012-Security_Vulnerability_Disclosure_Policy.pdf)


Silicon Labs has a Product Security Incident Response Team (PSIRT) that is dedicated to the case management of reported security vulnerabilities. The PSIRT works with other Silicon Labs groups including Applications, Developers, Sales, and Marketing to assess reported vulnerabilities, perform technical analysis and determine an appropriate response. The key processes for addressing vulnerabilities include:

- **Triage:** Determines what is needed to reproduce the vulnerability.
- **Technical Analysis and Disposition:** Confirms the validity of the security vulnerability, its scope, and its impact, and provides a resolution or disposition decision. Silicon Labs scores incidents according to CVSS 3.1 (Common Vulnerability Scoring System): low, medium, high, critical.
- **Output:** Communicates with our customers. The level and method of disclosure beyond the reporting entity depends on the severity and scope of the vulnerability.

Silicon Labs' provides broad vulnerability reporting to customers subscribed through our Salesforce portal (see [Automatically Applied Updates](#) for information on how to subscribe). A subscribed customer will see Security Advisory notifications something like the following:

 Notification Assignments







New Notifications






2 Items • Sorted by Created

Date • Filtered by my notification assignments - Read by Owner • Updated a minute ago

Q Search this list...

	CREATED DATE	SUBJECT	TYPE	
1	6/18/2019 5:01 PM	New Security Advisory	Security Advisory	
2	6/18/2019 5:10 PM	New Security Advisory	Security Advisory	

 Notification

A-00000013

Subject

Test 7/14/2019

Type

Security Advisory

Email Summary


Email Summary


Product Category

Audio and Radio;Interface

Body

Notification Body

 Files (1)



Notification File

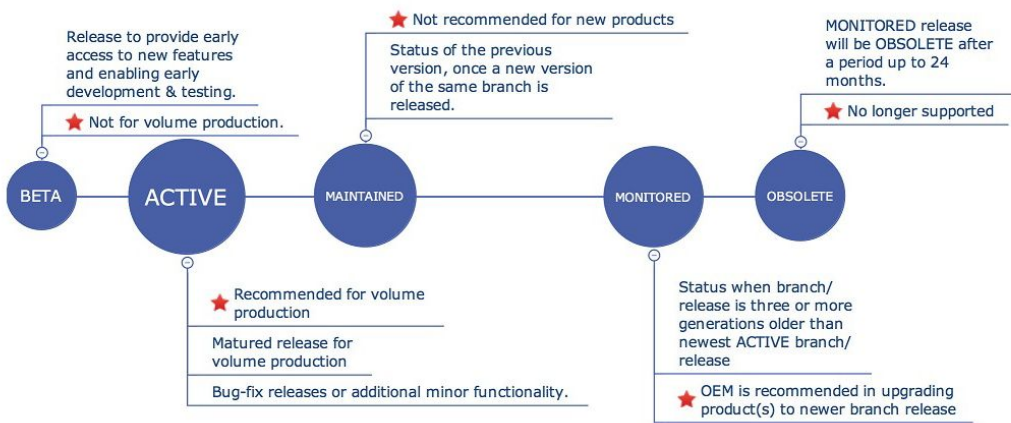
Jul 15, 2019 • 37KB • pdf

# Security Expiration Date

## Security Expiration Date

The manufacturer shall be transparent about the period of time that security updates will be provided. Like a manufacturer's product warranty, there shall be transparency around the support period of security updates.

Manufacturers should provide details about product support at various stages and publish security expiration dates. Z-Wave's Protocol Lifecycle provides an example.



The Lifecycle details in what phases updates will be applied, and to what product branch. For details on the various phases and how the lifecycle is implemented for specific Z-Wave products, see:

<https://www.silabs.com/products/development-tools/software/z-wave/embedded-sdk/life-cycle>



## Next Steps

# Next Steps

The Silicon Labs Security web page (<https://www.silabs.com/security>) contains links to a variety of general security-related resources. You may wish to bookmark the page, as it will be continually updated with new content, new tools, and new flows.

If you are already in development, we strongly recommend that you implement the standards described here as you develop, test, and release your product to customers.

If you are in the early stages of your product design and have not already selected a device or development environment, we recommend that you include security considerations in your decision. Silicon Labs provides information about the security features of our devices and development environments. Section [EFR32 Series 2 Device Security Features](#) highlights the features and their documentation references. In addition, protocol-specific security information is available in the following documents.

- [Zigbee Security](#)
- *AN1302: Bluetooth® Low Energy Application Security Design Considerations in SDK v3.x and Higher*
- *AN1329: Using Silicon Labs Secure Vault Features with OpenThread*
- [Bluetooth LE Fundamentals](#) and relevant Knowledge Base Articles (KBAs)
- *UG235.03: Architecture of the Silicon Labs Connect Stack v2.x*
- *UG435.03: Architecture of the Silicon Labs Connect Stack v3.x*

## Series 2 Device Security Features

# Series 2 Device Security Features

Protecting IoT devices against security threats is central to a quality product. Silicon Labs offers several security options to help developers build secure devices, secure application software, and secure paths of communication to manage those devices. Silicon Labs' security offerings were significantly enhanced by the introduction of the Series 2 products that included a Secure Engine. The Secure Engine is a tamper-resistant component used to securely store sensitive data and keys, and to execute cryptographic functions and secure services.

On Series 1 devices, the security features are implemented by the TRNG (if available) and CRYPTO peripherals.

On Series 2 devices, the security features are implemented by the Secure Engine and CRYPTOACC (if available). The Secure Engine may be hardware-based, or virtual (software-based). Throughout this document, the following abbreviations are used:

- HSE: Hardware Secure Engine
- VSE: Virtual Secure Engine
- SE: Secure Engine (either HSE or VSE)

Additional security features are provided by Secure Vault. Three levels of Secure Vault feature support are available, depending on the part and SE implementation, as reflected in the following table:

Security Level (1)	SE Support	MCU	Wireless SoC (2)
Secure Vault Base (SVB)	N/A	EFM32JG1, EFM32PG1, EFM32JG12, EFM32PG12, EFM32GG11, EFM32GG12, EFM32TG11	EFR32xG1, EFR32xG12, EFR32xG13, EFR32xG14
Secure Vault Mid (SVM)	VSE (VSE-SVM)	EFM32PG22C	EFR32xG22C, EFR32xG27C (3)
Secure Vault Mid (SVM)	HSE (HSE-SVM)	-	EFR32xG21A, EFR32xG21A (Rev C), EFR32MR21A (Rev C), EFR32xG23A, EFR32xG24A, EFR32xG25A, EFR32xG28A
Secure Vault High (SVH)	HSE only (HSE-SVH)	EFM32PG23B, EFM32PG28B	EFR32xG21B, EFR32xG12B (Rev C) EFR32xG23B, EFR32xG24B, EFR32xG25B, EFR32xG28B

Notes:

1. The features of different Secure Vault levels can be found in <https://www.silabs.com/security>.
2. The x is a letter B, F, M, or Z.
3. Unlike other VSE-SVM parts, the EFR32xG27C device has a built-in PUF.

Secure Vault Mid consists of two core security functions:

- **Secure Boot:** Process where the initial boot phase is executed from an immutable memory (such as ROM) and where code is authenticated before being authorized for execution.
- **Secure Debug access control:** The ability to lock access to the debug ports for operational security, and to securely unlock them when access is required by an authorized entity.

Secure Vault High offers additional security options:

- **Secure Key Storage:** Protects cryptographic keys by "wrapping" or encrypting the keys using a root key known only to the HSE-SVH.
- **Anti-Tamper protection:** A configurable module to protect the device against tamper attacks.

**Device authentication:** Functionality that uses a secure device identity certificate along with digital signatures to verify the source or target of device communications.

A Secure Engine Manager and other tools allow users to configure and control their devices both in-house during testing and manufacturing, and after the device is in the field.

Silicon Labs strongly recommends installing the latest SE FW image on Series 2 devices and updating to the latest GSDK to mitigate security vulnerabilities. The latest SE FW image can be found in this Windows folder for GSDK v4.x:

C:\Users\<PC USER NAME>\SimplicityStudio\SDKs\gecko\_sdk\util\se\_release\public

Refer to *AN1222: Production Programming of Series 2 Devices* for guidance on the SE firmware upgrade procedure. The latest SE firmware shipped with Series 2 devices and modules (if available) at the time of this writing are listed in the following table:

:::custom-table{30%,}

Series 2 MCU and Wireless SoC VSE - SVM	Shipped SE Firmware Version
EFM32PG22C	1.2.12
EFR32xG22C	1.2.12
EFR32xG22C (Rev D)	1.2.14
EFR32xG27C	2.2.1
...	

:::custom-table{30%,}

Series 2 Wireless SoC HSE - SVM	Shipped SE Firmware Version
EFR32xG21A	1.2.13
EFR32MR21A (Rev C)	1.2.16
EFR32xG21A (Rev C)	1.2.16
EFR32xG23A	2.1.7
EFR32xG24A	2.1.7
EFR32xG25A	2.2.1
EFR32xG28A	2.2.2
...	

:::custom-table{30%,}

Series 2 MCU and Wireless SoC HSE - SVH	Shipped SE Firmware Version
EFR32xG21B	1.2.13
EFR32xG21B (Rev C)	1.2.16
EFM32PG23B	2.1.7
EFR32xG23B	2.1.7
EFR32xG24B	2.1.7
EFR32xG25B	2.2.1
EFM32PG28B	2.2.2
EFR32xG28B	2.2.2
...	

In support of these products Silicon Labs offers whitepapers, webinars, and documentation. The following table summarizes the key security documents:

:::custom-table{30%,50%,20%}

Document	Summary	Applicability
Document	Summary	Applicability
<a href="#">Series 2 Secure Debug</a>	How to lock and unlock Series 2 debug access, including background information about the Secure Engine	Series 2
<a href="#">Series 2 Secure Boot with RTSL</a>	Describes the secure boot process on Series 2 devices using Secure Engine.	Series 2
<a href="#">Anti-Tamper Protection Configuration and Use</a>	How to program, provision, and configure the anti-tamper module	Series 2 with SVH
<a href="#">Authenticating Silicon Labs Devices using Device Certificates</a>	How to authenticate a device using secure device certificates and signatures, at any time during the life of the product	Series 2 with SVH
<a href="#">Secure Key Storage</a>	How to securely “wrap” keys so they can be stored in non-volatile storage	Series 2 with SVH
AN1222: Production Programming of Series 2 Devices	How to program, provision, and configure security information using Secure Engine during device production	Series 2
AN1303: Programming Series 2 Devices Using the Debug Challenge Interface (DCI) and Serial Wire Debug (SWD)	How to provision and configure Series 2 devices through the DCI and how to program their internal flash memory through the SWD	Series 2
AN1311: Integrating Crypto Functionality Using PSA Crypto Compared to Mbed TLS	How to integrate crypto functionality into applications using Silicon Labs implementation of PSA Crypto compared to Mbed TLS	Series 1 and Series 2
<a href="#">Series 2 TrustZone</a>	Describes the basics of TrustZone, secure and privileged programming model, and shows how to upgrade existing application to TrustZone.	Series 2

## Overview

# Silicon Labs IoT Security Developer's Guide

The IoT Security Developer's Guide offers detailed information on how to implement each of the device security features. This content is applicable to any protocol that supports the feature described. Additional protocol-specific information for Bluetooth, Bluetooth Mesh, OpenThread, and Zigbee is available in the [protocol-specific section](#).

- **Series 2 Secure Debug:** Describes how to lock and unlock the debug access of EFR32 Gecko Series 2 devices. Many aspects of the debug access, including the secure debug unlock are described. The Debug Challenge Interface (DCI) and Secure Engine (SE) Mailbox Interface for locking and unlocking debug access are also included.
- **Series 2 TrustZone:** Covers the basics of ARMv8-M TrustZone, describes how TrustZone is implemented on Series 2 devices, and provides application examples.
- **Production Programming of Series 2 Devices (PDF):** Provides details on programming, provisioning, and configuring Series 2 devices in production environments. Covers Secure Engine Subsystem of Series 2 devices, which runs easily upgradeable Secure Engine (SE) or Virtual Secure Engine (VSE) firmware.
- **Anti-Tamper Protection Configuration and Use:** Shows how to program, provision, and configure the anti-tamper module on EFR32 Series 2 devices with Secure Vault.
- **Authenticating Silicon Labs Devices using Device Certificates:** Describes how to authenticate an EFR32 Series 2 device with Secure Vault, using secure device certificates and signatures.
- **Secure Key Storage:** Explains how to securely "wrap" keys in EFR32 Series 2 devices with Secure Vault, so they can be stored in non-volatile storage.
- **Programming Series 2 Devices Using the Debug Challenge Interface (DCI) and Serial Wire Debug (SWD) (PDF):** Describes how to provision and configure Series 2 devices through the DCI and SWD.
- **Integrating Crypto Functionality Using PSA Crypto Compared to Mbed TLS (PDF):** Describes how to integrate crypto functionality into applications using PSA Crypto compared to Mbed TLS.

## Series 2 Secure Debug

# Series 2 Secure Debug

NOTE: This section replaces *AN1190: Series 2 Secure Debug*. Further updates to this application note will be provided [here](#).

This application note describes how to lock and unlock the debug access of Series 2 devices. Many aspects of the debug access, including the secure debug unlock, are discussed. The Debug Challenge Interface (DCI) and Mailbox Interface for locking and unlocking debug access are also included.

The debug locks and unlocks for the Cortex-M33 debug interface are implemented through the Secure Engine on Series 2 devices.

## Key Points

- Basic overview of the Secure Engine.
- Debug port access by Debug Challenge Interface (DCI) or Mailbox Interface.
- New locking and unlocking features for Series 2 devices.
- Examples for provisioning and Secure Debug Unlock.

Series 2 Device Security Features

# Series 2 Device Security Features

Protecting IoT devices against security threats is central to a quality product. Silicon Labs offers several security options to help developers build secure devices, secure application software, and secure paths of communication to manage those devices. Silicon Labs’ security offerings were significantly enhanced by the introduction of the Series 2 products that included a Secure Engine. The Secure Engine is a tamper-resistant component used to securely store sensitive data and keys and to execute cryptographic functions and secure services.

On Series 1 devices, the security features are implemented by the TRNG (if available) and CRYPTO peripherals.

On Series 2 devices, the security features are implemented by the Secure Engine and CRYPTOACC (if available). The Secure Engine may be hardware-based, or virtual (software-based). Throughout this document, the following abbreviations are used:

- HSE - Hardware Secure Engine
- VSE - Virtual Secure Engine
- SE - Secure Engine (either HSE or VSE)

Additional security features are provided by Secure Vault. Three levels of Secure Vault feature support are available, depending on the part and SE implementation, as reflected in the following table:

Level (1)	SE Support	Part (2)
Secure Vault High (SVH)	HSE only (HSE-SVH)	Refer to <a href="#">IoT Endpoint Security Fundamentals</a> for details on supporting devices.
Secure Vault Mid (SVM)	HSE (HSE-SVM)	"
"	VSE (VSE-SVM)	"
Secure Vault Base (SVB)	N/A	"

Notes:

1. The features of different Secure Vault levels can be found in <https://www.silabs.com/security>.
2. [IoT Endpoint Security Fundamentals](#).

Secure Vault Mid consists of two core security functions:

- Secure Boot: Process where the initial boot phase is executed from an immutable memory (such as ROM) and where code is authenticated before being authorized for execution.
- Secure Debug access control: The ability to lock access to the debug ports for operational security, and to securely unlock them when access is required by an authorized entity.

Secure Vault High offers additional security options:

- Secure Key Storage: Protects cryptographic keys by "wrapping" or encrypting the keys using a root key known only to the HSE-SVH.
- Anti-Tamper protection: A configurable module to protect the device against tamper attacks.
- Device authentication: Functionality that uses a secure device identity certificate along with digital signatures to verify the source or target of device communications.

A Secure Engine Manager and other tools allow users to configure and control their devices both in-house during testing and manufacturing, and after the device is in the field.

## User Assistance

In support of these products, Silicon Labs offers whitepapers, webinars, and documentation. The following table summarizes the key security documents:

Document	Summary	Applicability
Series 2 Secure Debug (this application note)	How to lock and unlock Series 2 debug access, including background information about the SE	Secure Vault Mid and High
<a href="#">Series 2 Secure Boot with RTSL</a>	Describes the secure boot process on Series 2 devices using SE	Secure Vault Mid and High
<a href="#">Anti-Tamper Protection Configuration and Use</a>	How to program, provision, and configure the anti-tamper module	Secure Vault High
<a href="#">Authenticating Silicon Labs Devices using Device Certificates</a>	How to authenticate a device using secure device certificates and signatures, at any time during the life of the product	Secure Vault High
<a href="#">Secure Key Storage</a>	How to securely 'wrap' keys so they can be stored in non-volatile storage.	Secure Vault High
AN1222: Production Programming of Series 2 Devices	How to program, provision, and configure security information using SE during device production	Secure Vault Mid and High

## Key Reference

Public/Private keypairs along with other keys are used throughout Silicon Labs security implementations. Because terminology can sometimes be confusing, the following table lists the key names, their applicability, and the documentation where they are used.

Key Name	Customer Programmed	Purpose
Public Sign key (Sign Key Public)	Yes	Secure Boot binary authentication and/or OTA upgrade payload authentication
Public Command key (Command Key Public)	Yes	Secure Debug Unlock or Disable Tamper command authentication
OTA Decryption key (GBL Decryption key) aka AES-128 Key	Yes	Decrypting GBL payloads used for firmware upgrades
Attestation key aka Private Device Key	No	Device authentication for secure identity

## SE Firmware

Silicon Labs strongly recommends installing the latest SE firmware on Series 2 devices to support the required security features. Refer to [AN1222](#) for the procedure to upgrade the SE firmware and [IoT Endpoint Security Fundamentals](#) for the latest SE Firmware shipped with Series 2 devices and modules.



## Introduction To Secure Debug

# Introduction to Secure Debug

## Debug Lock

All devices require the capability to lock out debug access to the device. This prevents attackers from using the debug interface to perform the following illegal operations:

- Reprogramming the device
- Interrogating the device
- Interfering with the operation of the device

A fairly standard practice during the board-level test in production is to program, test, and lock the parts.

Three different locks can be enabled on the Series 2 debug interface:

- [Standard-debug-lock](#)
- [Permanent-debug-lock](#)
- [Secure-debug-lock](#)

Silicon Labs provides [Custom Part Manufacturing Service \(CPMS\)](#) to securely configure the debug port of the chip to one of the three possible locks before the devices leave the factory.

## Debug Unlock

Users need to unlock parts under a number of circumstances:

- Code development
- Field failure diagnosis
- Product field service
- Existing inventory reprogramming

Two different unlocks can run on the Series 2 debug interface:

- [Standard-debug-unlock](#)
- [Secure-debug-unlock](#)

Secure Engine Subsystem

# Secure Engine Subsystem

## Overview

The HSE refers to a separate security co-processor that provides hardware isolation between security functions and the host processor.

The VSE refers to a collection of security functions available to the host processor in Root mode if a separate security co-processor is not provided.

The SE is used to perform a series of cryptographic operations and other secure system operations as described in the following table.

Operation	VSE-SVM	HSE-SVM	HSE-SVH	Description
Unique ID	Y	Y	Y	Software can identify every device.
Secure Boot with RTSL	Y	Y	Y	Only boot authenticated firmware.
Secure Debug	Y	Y	Y	Allow enhanced failure analysis.
Crypto Engine (1)	-	Y	Y	Up to 256-bit ciphers and elliptic curves.
TRNG (1)	-	Y	Y	Generate keys for cryptography.
DPA Countermeasures	-	Y	Y	Resist side channel attacks.
Secure Key Storage	-	-	Y	Protected by PUF technology.
Secure Key Management	-	-	Y	Isolate encrypted keys from application code.
Secure Attestation	-	-	Y	Ensure integrity and authenticity.
Anti-Tamper	-	-	Y	Detect tamper and protect keys/data.
Advanced Crypto	-	-	Y	Up to 512-bit ciphers and 521-bit elliptic curves.

Note:

1. On VSE-SVM devices, the crypto engine and TRNG (True Random Number Generator) are implemented by the CRYPTOACC (Cryptographic Accelerator) peripheral.

To start using the [secure debug unlock](#) functionality, the device needs to be [provisioned](#). These steps include writing one-time-programmable (OTP) settings to the SE to determine which functionality is enabled, and uploading the Public Command Key to validate a secure debug attempt.

This application note describes how the different device debug locks and unlocks are implemented through the SE on Series 2 devices.

The secure debug feature is implemented by Root code executed by the HSE Core or by the Cortex-M33 operating in VSE (Root mode).

Silicon Labs strongly recommends installing the latest SE firmware on Series 2 devices to support the required security features. The latest SE firmware image (.seu and .hex) and release notes can be found in the Windows folder below.

For GSDK v3.2 and lower:

```
C:\SiliconLabs\SimplicityStudio\v5\developer\sdk\gecko_sdk_suite\<GSDK VERSION>\util\se_release\public
```

For GSDK v4.0 and higher:

```
C:\Users\<PC USER NAME>\SimplicityStudio\SDKs\gecko_sdk\util\se_release\public
```

## Command Interface

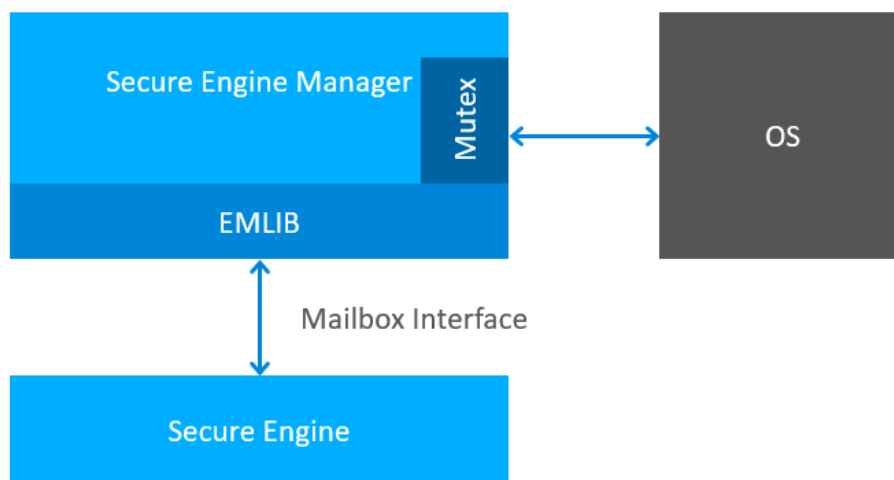
Interaction with the SE is performed over a command interface. The command interface is available through a dedicated Debug Challenge Interface (DCI) as well as through a mailbox interface from the Cortex-M33.

Some commands may not be available at all times and may not be accessible over both interfaces. The DCI typically only contains operations for setting up a new device and for locking it down (meant for production processes), while the mailbox interface also contains commands to support cryptographic operations in HSE.

## Mailbox

Mailbox operations should not be performed directly, but rather should be executed through the appropriate functions in `em_se.c` of `emlib`. The `em_se.c` provides an abstraction of the mailbox interface, allowing message construction and DMA transfer setup.

On top of `emlib`, the Secure Engine Manager (SE Manager) provides an abstraction of the Secure Engine's command set. The SE Manager also provides APIs for cryptographic operations and thread synchronization. The SE Manager is available in GSDK v3.0 or later.



**Note:** Some functions in `em_se.c` of `emlib` are deprecated in GSDK v3.0 and will be removed in a future version of `emlib`. All high-level functionality has been moved to the SE Manager.

## Debug Challenge Interface (DCI)

The Debug Challenge Interface (DCI) is made available through commands in Simplicity Studio and Simplicity Commander. This is the easiest way to access and set up the different security options.

For more information about DCI, see [AN1303: Programming Series 2 Devices using the Debug Challenge Interface \(DCI\) and Serial Wire Debug \(SWD\)](#).

Debug Lock

# Debug Lock

## Overview

The debug access port connected to the Series 2 device's Cortex-M33 processor can be closed by issuing commands to the SE, either from a debugger over DCI or through the mailbox interface. These three debug lock properties govern the behavior of the debug lock.

Property	Description If Set	Default Value
Debug Lock	The debug port is kept locked on boot.	False (Disabled)
Device Erase	The Erase Device command is available.	True (Enabled)
Secure Debug	Secure debug unlock is available.	False (Disabled)

The following sections describe how to interact with these properties and how to enable debug locks using the SE command interface either over DCI or the mailbox interface. The status of the debug lock can be inspected using the [Read Lock Status](#) command.

## Standard Debug Unlock

The device is in standard debug unlock state if the debug lock properties are in default values.

Secure Debug	Device Erase	Debug Lock	Description
Disabled	Enabled	Disabled (Unlock)	All debug operations are allowed.

## Standard Debug Lock

With the default properties in the table above, the device can be locked using the [Apply Lock](#) command. The typical flow for this configuration is simply to issue the `Apply Lock` command after the device has been programmed, either using a DCI command from the [programming debugger](#) or through the [mailbox interface](#).

Secure Debug	Device Erase	Debug Lock	Description
Disabled	Enabled	Enabled (Standard)	The Erase Device command will wipe the main flash and RAM, and then a reset will yield an unlocked device.

The standard debug lock behaves similarly to Series 1 devices. The access port can be closed, but issuing a [device erase](#) wipes the device and opens the debug port again.

## Permanent Debug Lock

The [Erase Device](#) command can be disabled, which permanently enables the debug lock. This can be done at any time by issuing the [Disable Device Erase](#) command, even after the debug lock has been enabled.

Secure Debug	Device Erase	Debug Lock	Description
Disabled	Disabled	Enabled (Permanent)	The part cannot be unlocked. Devices with Permanent Debug Lock engaged cannot be returned for failure analysis.

## Secure Debug Lock

For secure debug lock, the debug interface can be temporarily enabled by answering a challenge if the [Secure debug](#) property is enabled before locking.

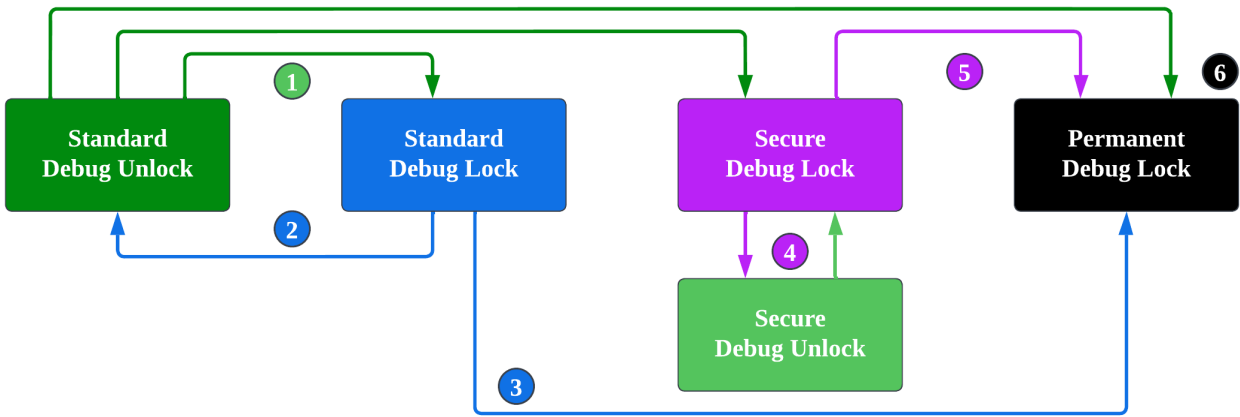
Secure Debug	Device Erase	Debug Lock	Description
Enabled (1)	Disabled (2)	Enabled (Secure)	Secure debug unlock is enabled, which makes it possible to securely open the debug lock temporarily to reprogram or debug a locked device.

Note:

- 1. Secure debug is enabled in two steps before the debug lock is enabled:
  - a. Install the Public Command Key using [Simplicity Studio](#) or [Simplicity Commander](#) or directly through the [SE Manager API](#).
  - b. Enable secure debug using Simplicity Studio or Simplicity Commander or directly through the SE Manager API.
- 2. Disable the device erase using Simplicity Studio or Simplicity Commander or directly through the SE Manager API. This is an **IRREVERSIBLE** action and should be disabled **AFTER** the secure debug is enabled.

### Debug Lock State Transition

The following figure describes the transitions between different debug lock states.



- 1. [Standard debug unlock](#) can transit to any debug lock state.
- 2. [Standard debug lock](#) can revert to standard debug unlock via an [Erase Device](#) command (erase the main flash and RAM). After the device is reset, debug port remains unlocked until it is explicitly locked again.
- 3. Standard debug lock can transit to permanent debug lock by disabling the [Device Erase](#) property but cannot transit to secure debug lock.
- 4. [Secure debug lock](#) can use [Debug Unlock Token](#) to temporary transit to [secure debug unlock](#), which does not erase the main flash and RAM but enables debug operations. The device reverts to the secure debug lock through a power-on or pin reset.
- 5. Secure debug lock can transit to permanent debug lock by disabling the [Secure Debug](#) property but cannot transit to standard debug lock.
- 6. [Permanent debug lock](#) is a terminal state and cannot transit to any debug lock or unlock state.

### Debug Lock Command Reference

The commands for debug lock are described in the following table.

DCI Command (1)	Mailbox (SE Manager) API (2)	Description	Availability
Apply Lock	sl_se_apply_debug_lock	Enables the debug lock for the part.	While debug is unlocked.

DCI Command (1)	Mailbox (SE Manager) API (2)	Description	Availability
Read Lock Status	sl_se_get_debug_lock_status	Returns the current debug lock status and configuration.	Always.
Disable Device Erase	sl_se_disable_device_erase	Disables the Erase Device command. This command does not lock the debug interface to the part, but it is an IRREVERSIBLE action for the part.	Always.
Disable Secure Debug	sl_se_disable_secure_debug	Disables the secure debug functionality that can be used to open a locked debug port.	While secure debug is enabled.
Enable Secure Debug	sl_se_enable_secure_debug	Enables the secure debug functionality that can be used to open a locked debug port.	While debug is unlocked and Public Command Key is uploaded.
Set debug options	sl_se_set_debug_options	Configures the TrustZone access permissions of the debug interface. (3)	While debug is unlocked.
Init Pub Key	sl_se_init_otp_key	Used to provision a single public key during device initialization. The public key cannot be changed once written, and the command will be unavailable for that key.	Available once for each key.
Read Pub Key	sl_se_read_pubkey	Reads the stored public key.	Always.
Get Challenge	sl_se_roll_challenge	Used to roll the current challenge value (16 bytes) to revoke secure debug access. (4)	While Public Command Key is uploaded.

**Notes:**

1. Performing these commands over DCI is implemented in Simplicity Studio and Simplicity Commander.
2. The `sl_se_apply_debug_lock`, `sl_se_get_debug_lock_status`, `sl_se_init_otp_key`, and `sl_se_read_pubkey` are available on all Series 2 devices. Other APIs are only available on HSE devices. The SE Manager API document can be found at <https://docs.silabs.com/gecko-platform/latest/service/api/group-sl-se-manager>.
3. For more information about debug options, see [TrustZone Debug Authentication](#).
4. A new challenge will only be generated if the current one has been successfully used at least once.

## Debug Unlock

# Debug Unlock

## Overview

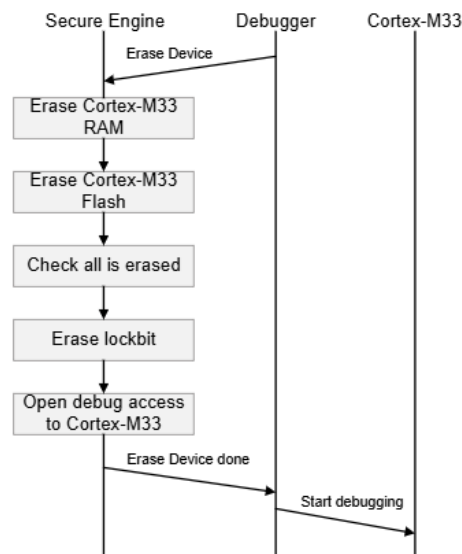
The debug access port connected to the Series 2 device's Cortex-M33 processor can be opened by issuing commands to the SE, either from a debugger over DCI or through the mailbox interface.

New on the Series 2 devices is the addition of [secure debug unlock](#) functionality. When enabled, it is possible to request a challenge from the device and, by answering the challenge, disable the debug lock until the next power-on or pin reset.

The status of the debug lock can be inspected using the [Read Lock Status](#) command.

## Standard Debug Unlock

With the properties of the [standard debug lock](#) or [secure debug lock with Device Erase enabled](#), the device can be returned to the [standard debug unlock](#) state using the [Erase Device](#) command. This command will wipe the main flash and RAM and verify they are empty before opening the debug lock. It will not wipe user data and provisioned SE settings.



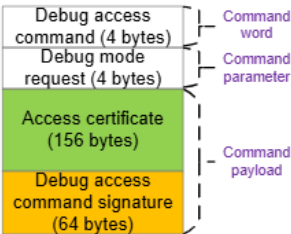
## Secure Debug Unlock

In a secure debug unlock scenario, the customer, who has control over the Private Command Key for a SE, has programmed a Public Command Key into the device. The Public Command Key is used to verify the signature on a certificate, telling the SE what authorization has been given by the owner of the key (customer) to the one issuing the command (customer or delegate). Authorization can be granted, for example, to unlock only the debug port on the Cortex-M33, or to restore only specific tamper signals on HSE-SVH devices.

This mode is particularly useful in failure analysis scenarios because it allows devices to be unlocked without losing flash and RAM contents.

Debug Unlock Token

The elements of the Debug Unlock Token are described in the following figures and table.



Element	Value	Description
Debug access command	0xfd010001	The command word of the Debug Unlock Token.
Debug mode request	Device-dependent	The command parameter of the debug access command.
Access certificate (1)	Device-dependent	See section Access Certificate.
Debug access command signature (1)	Device-dependent	See section Challenge Response.

Note:

1. The debug access command payload consists of an [access certificate](#) and a [debug access command signature](#).

Debug Mode Request																																
Bit	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Name	Reserved	Reserved	Reserved	Reserved	Reserved	Reserved	Reserved	Reserved	Reserved	Reserved	Reserved	Reserved	Reserved	Reserved	Reserved	Reserved	Reserved	Reserved	Reserved	Reserved	Reserved	Reserved	Reserved	Reserved	Reserved	Reserved	SPIDLOCK	SPIDLOCK	NIDLOCK	DBGLOCK	Enable debug port	Reserved

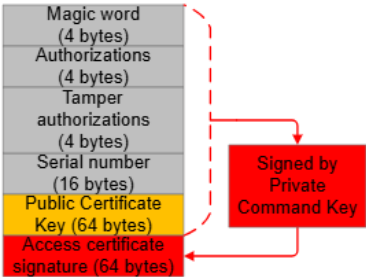
Notes:

- Enable debug port - Debug port enabled if set.
- **DBGLOCK** (Non-secure, Invasive debug lock) - The Invasive debug features for the Non-secure state are unlocked if set.
- **NIDLOCK** (Non-secure, Non-invasive debug lock) - The Non-invasive debug features for the Non-secure state are unlocked if set.
- **SPIDLOCK** (Secure, Invasive debug lock) - The Invasive debug features for the Secure state are unlocked if set.
- **SPNIDLOCK** (Secure, Non-Invasive debug lock) - The Non-invasive debug features for the Secure state are unlocked if set.
- All reserved bits should be 0, and bit 1 must be 1 to access the debug port.
- For the TrustZone-unaware debugging, bits 2 to 5 are irrelevant, so bits 1 to 5 are usually set ( 0x0000003e ) to match with the [Authorizations](#) in the access certificate.
- For the TrustZone-aware debugging, bits 2 to 5 are relevant. Refer to [Trust Zone Debug Authentication](#) for details about these debug options.



Access Certificate

The elements of the access certificate are described in the following figures and table.



Element	Value	Description
Magic word	0xe5ecce01	A constant value used to identify the access certificate.
Authorizations	0x0000003e (1)	A value used to authorize which bit in the debug mode request can be enabled for secure debug.
Tamper Authorizations	0x00000000 or 0xfffffb6 (2)	A value used to authorize which bit in the tamper disable mask can be set to disable the tamper response.
Serial number	Device-dependent	A number used to compare against the on-chip serial number for secure debug or tamper disable.
Public Certificate Key (3)	Device-dependent	The public key corresponding to the Private Certificate Key (3) used to generate the signature (ECDSA-P256-SHA256) in a challenge response.
Access certificate signature	Device-dependent	All the content above is signed (ECDSA-P256-SHA256) by the Private Command Key corresponding to the Public Command Key in the SE OTP.

Notes:

- 1. This value allows all [debug options](#) to be reset for secure debug.
- 2. Value that sets available bits in the tamper disable mask for tamper disable (HSE-SVH device only).
- 3. The Private/Public Certificate Key is a randomly generated key pair. It can be ephemeral or retainable.

The Private Certificate Key can be used repeatedly to generate the signature in a [challenge response](#) on one device until the Private/Public Certificate Key pair is discarded. This can reduce the frequency of access to the Private Command Key, allowing more restrictive access control on that key.

For more information about tamper disable, see [Anti-Tamper Protection Configuration and Use](#).

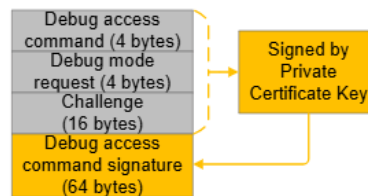
Authorizations		
Name		Bit
Reserved		31
Reserved		30
Reserved		29
Reserved		28
Reserved		27
Reserved		26
Reserved		25
Reserved		24
Reserved		23
Reserved		22
Reserved		21
Reserved		20
Reserved		19
Reserved		18
Reserved		17
Reserved		16
Reserved		15
Reserved		14
Reserved		13
Reserved		12
Reserved		11
Reserved		10
Reserved		9
Reserved		8
Reserved		7
Reserved		6
SPNIDLOCK request mask		5
SPIDLOCK request mask		4
NIDLOCK request mask		3
DBGLOCK request mask		2
Enable debug port request mask		1
Reserved		0

Notes:

- Set the bit to enable the corresponding bit in the debug mode request.
- The Debug Unlock Token will reset the corresponding [debug option](#) if the same bit is set in Debug mode request and Authorizations.

## Challenge Response

The elements of the challenge response are described in the following figure and table.



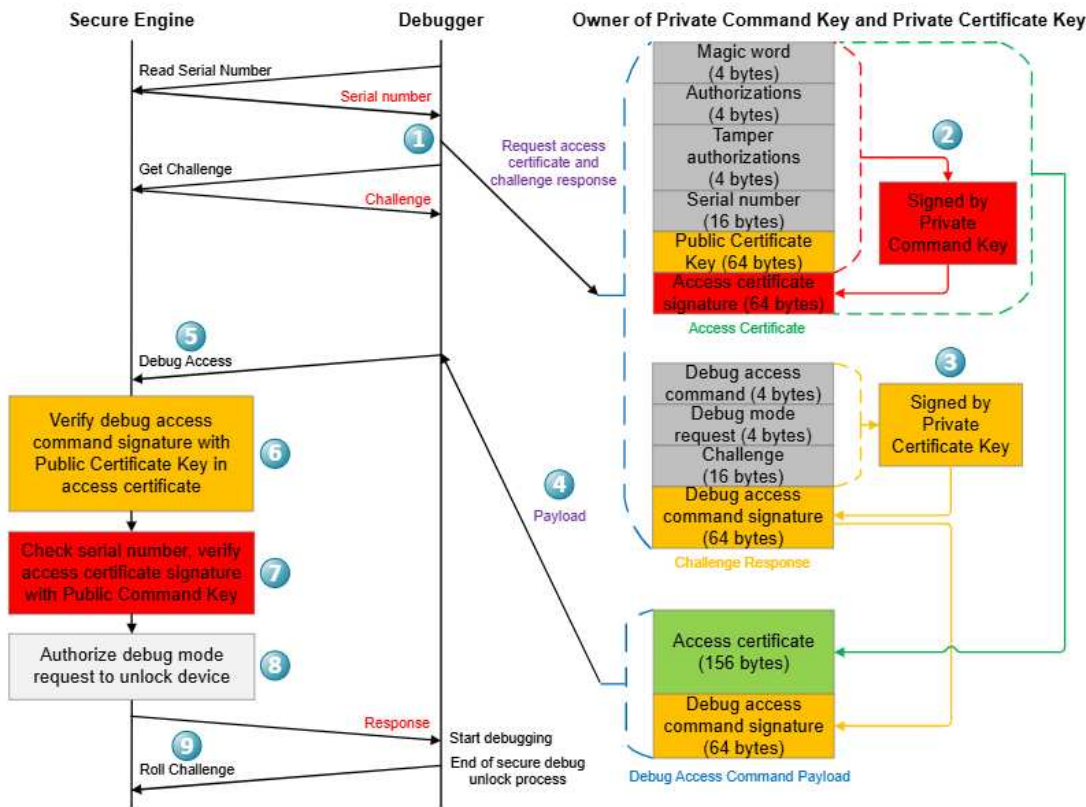
Element	Value	Description
Debug access command	0xfd010001	The command word of the Debug Unlock Token.
Debug mode request	Device-dependent	The command parameter of the debug access command.
Challenge	Device-dependent (1)	A random value generated by the SE.
Debug access command signature	Device-dependent (2)	All the content above is signed (ECDSA-P256-SHA256) by the Private Certificate Key corresponding to the Public Certificate Key in the access certificate.

Notes:

1. The challenge remains unchanged until it is updated to a new random value by [rolling the challenge](#). The Private Certificate Key can be reused for signing when the device challenge is refreshed.
2. This signature is the final argument of the [Debug Unlock Token](#).

## Debug Access Flow

The debug access flow is described in the following figure.



1. Get the serial number and challenge from the SE.
2. Generate the [access certificate](#) with the device serial number.
3. Generate the [challenge response](#) with device challenge.
4. Generate the debug access command payload with access certificate and debug access command signature.
5. Send the [Debug Unlock Token](#) to the SE.
6. Verify the debug access command signature using the Public Certificate Key in the access certificate.
7. Verify the serial number and the access certificate signature using the on-chip serial number and Public Command Key in the SE OTP.
8. Authorize the debug mode request to reset the [debug options](#) until the next power-on or pin reset.
9. Roll the challenge to invalidate the current Debug Unlock Token.

TrustZone Debug Authentication

The debug and trace support in the Cortex-M33 devices are based on the [CoreSight](#) architecture, which can be classified into Invasive and Non-invasive debugging features as described in the following table.

Classification	Debug and Trace Features	Description
Invasive	Core debug (e.g., single stepping), Breakpoints, Data watchpoints, Halt mode debugging	These features halt the Cortex-M33 core and change the program execution flow.
Non-invasive	Embedded Trace Macrocell (ETM), Micro Trace Buffer (MTB), Data trace, Instrumentation Trace Macrocell (ITM), Profiling	These features have a minor or no impact on the program execution flow.

The separation of Invasive and Non-invasive debug and trace operations in CoreSight architecture can apply to TrustZone debug authentication, which defines the permission levels of the debug and trace features on Secure and Non-secure worlds.

The table below describes four debug options in SE to support TrustZone debug authentication. It is possible to restrict the TrustZone access permissions of the debug interface by setting one or more of the following options.

Debug Option	Description
DBGLOCK	Non-secure, Invasive debug lock. If this bit is set, the Invasive debug features for the Non-secure state are locked.
NIDLOCK	Non-secure, Non-invasive debug lock. If this bit is set, the Non-invasive debug features for the Non-secure state are locked.
SPIDLOCK	Secure, Invasive debug lock. If this bit is set, the Invasive debug features for the Secure state are locked.
SPNIDLOCK	Secure, Non-invasive debug lock. If this bit is set, the Non-invasive debug features for the Secure state are locked.

**Notes:**

- Use [Simplicity Commander](#) or the [SE Manager API](#) to set the debug options.
- The state of the debug options is stored permanently in SE and can only be reset to the default value ( 0000 ) through the [Erase Device](#) command (if enabled).
- A secure debug lock device ( [Device Erase](#) was disabled ) can only use the [Debug Unlock Token](#) to [temporarily unlock](#) (reset) the debug options to debug the Secure and Non-secure applications.

The following conditions are recommended (1, 2, and 3) or mandatory (4) when setting up the debug options for secure debug unlock.

- If SPIDLOCK is unlocked, then DBGLOCK should also be unlocked.
- If SPNIDLOCK is unlocked, then NIDLOCK should also be unlocked.
- If DBGLOCK is unlocked, the NIDLOCK should also be unlocked.
- If SPIDLOCK is unlocked, then SPNIDLOCK is automatically unlocked.

The following table lists the recommended combinations of debug options.

SPNIDLOCK	SPIDLOCK	NIDLOCK	DBGLOCK	Description
0	0	0	0	Allows all debug and trace features for both the Secure and the Non-secure world (default setting).
0	1	0	0	Only allows a Non-invasive debug in the Secure world. Allows both Invasive and Non-invasive debugs in the Non-secure world.
0	1	0	1	Only allows a Non-invasive debug in the Secure and the Non-secure world.
1	1	0	0	Only allows debug and trace features in the Non-secure world.
1	1	0	1	Only allows a Non-invasive debug in the Non-secure world.
1	1	1	1	All debug and trace features are disabled.

**Notes:**

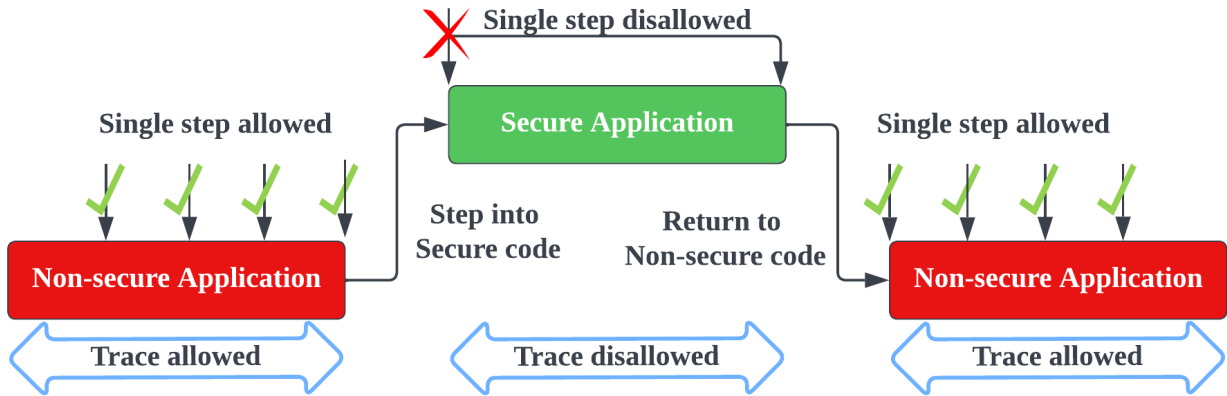
- [Trace Point Interface Unit \(TPIU\) registers' access fault](#) will occur and lock the processor in a security assertion if both NIDLOCK and DBGLOCK in debug option are set ( xx11 ). The device will be unrecoverable if it is in the [permanent debug lock](#) state.
- The workaround is to avoid using the xx11 debug option or avoid accessing the TPIU registers and upgrade to SE firmware  $\geq$  v1.2.14 (xG21 and xG22) or  $\geq$  v2.2.1 (other Series 2 devices) so that the debug options cannot be modified after the device is locked.

The highly recommended setting of debug options is to allow debugging in the Non-secure world while, at the same time, disabling debugging for the Secure world ( 1100 ).

- Secure memories (flash and RAM) are not accessible by the debugger.
- All debug access is blocked from accessing Secure addresses.
-

- The debugger will ignore the vector-catch events generated by the Secure exceptions.
- Trace sources (e.g., ETM) will stop generating instruction/data trace packets when the Cortex-M33 is in a Secure state.
  - The debugger can neither halt a Secure application (e.g., breakpoint) nor single step into it.
  - The Cortex-M33 will not stop when stepping into the Secure application until it returns to the Non-secure state.

The figure below describes the debug scenario of debug options with 1100 configuration.



The following examples describe the relationship between debug options and debug mode request when performing a secure debug unlock on Series 2 devices.

Example 1: All debug and trace features for both the Secure and the Non-secure world are allowed ( 0000 )

Debug Options	Authorizations	Debug Mode Request	Debug options after Secure Debug Unlock	Description
0000	00 1111 10	00 xxxx 10	0000	No action

Example 2: Only debug and trace features in the Non-secure world are allowed ( 1100 )

Debug Options	Authorizations	Debug Mode Request	Debug options after Secure Debug Unlock	Description
1100	00 1111 10	00 00xx 10	1100	No action
1100	00 1111 10	00 10xx 10	0100	Unlock SPNIDLOCK
1100	00 1111 10	00 01xx 10 or 00 11xx 10	0000(reset SPIDLOCK will automatically unlock SPNIDLOCK)	Unlock SPNIDLOCK and SPIDLOCK

Notes:

- The bit order of debug options are SPNIDLOCK (MSB), SPIDLOCK, NIDLOCK, and DBGLOCK (LSB).
- **Debug options:** 0 = Unlocked, 1 = Locked
- **Authorizations** in the access certificate: 0 = Disable, 1 = Enable
- The authorizations in the access certificate are usually set to 00|1111|10 ( 0x3e ), so the corresponding debug options (bits 2 to 5) can be reset (unlocked) by debug mode request during secure debug unlock.
- **Debug mode request** (bits 2 to 5) in the Debug Unlock Token:
  - 0 = No action on the corresponding debug option if it was locked (i.e., 1)
  - 1 = Reset (unlock) the corresponding debug option from 1 to 0 if it was locked (i.e., 1)
  - x = No action (either 0 or 1) on the corresponding debug option if it was unlocked (i.e., 0)
- Debug options return to the original state after power-on or pin reset.

Debug Unlock Command Reference

The commands for debug unlock are described in the following table.

DCI Command (1)	Mailbox (SE Manager) API (2)	Description	Availability
Erase Device	sl_se_erase_device	Performs a device mass erase and resets the debug configuration to its initial unlocked state.	While Device Erase is enabled.
Read Serial Number	sl_se_get_serialnumber	Reads out the serial number (16 bytes) of the Series 2 device.	Always.
Get Challenge	sl_se_get_challenge	Reads out the current challenge value (16 bytes) for Secure debug unlock.	While Public Command Key is uploaded.
Debug Access	sl_se_open_debug	Opens the secure debug access of the Cortex-M33.	Only when Secure Debug is enabled.

**Notes:**

1. Performing these commands over DCI is implemented in Simplicity Studio and Simplicity Commander.
2. These APIs are only available on HSE devices. The SE Manager API document can be found at <https://docs.silabs.com/gecko-platform/latest/service/api/group-sl-se-manager>.

Examples

# Examples

## Overview

The examples for Series 2 debug lock and debug unlock are described in the following table.

Example	Device (Radio Board)	SE Firmware	Tool
Standard debug lock	EFR32MG21A010F1024IM32 (BRD4181A)	Version 1.2.14	SE Manager
Standard debug lock and unlock	EFR32MG21A010F1024IM32 (BRD4181A)	Version 1.2.14	Simplicity Commander
"	EFR32MG21A010F1024IM32 (BRD4181A)	Version 1.2.9	Simplicity Studio 5
Provision Public Command Key	EFR32MG21A010F1024IM32 (BRD4181A)	Version 1.2.14	SE Manager
Secure debug lock	EFR32MG21A010F1024IM32 (BRD4181A)	Version 1.2.14	SE Manager
Provision Public Command Key and secure debug lock	EFR32MG21A010F1024IM32 (BRD4181A)	Version 1.2.14	Simplicity Commander
"	EFR32MG21A010F1024IM32 (BRD4181A)	Version 1.2.9	Simplicity Studio 5
Secure debug unlock and roll challenge	EFR32MG21A010F1024IM32 (BRD4181A)	Version 1.2.14	SE Manager
"	EFR32MG21A010F1024IM32 (BRD4181A)	Version 1.2.14	Simplicity Commander
"	EFR32MG21A010F1024IM32 (BRD4181A)	Version 1.2.9	Simplicity Studio 5
Secure debug unlock	EFR32MG21A010F1024IM32 (BRD4181A)	Version 1.2.9	IAR v8.50.9
Permanent debug lock	EFR32MG21A010F1024IM32 (BRD4181A)	Version 1.2.14	Simplicity Commander
"	EFR32MG21A010F1024IM32 (BRD4181A)	Version 1.2.9	Simplicity Studio 5

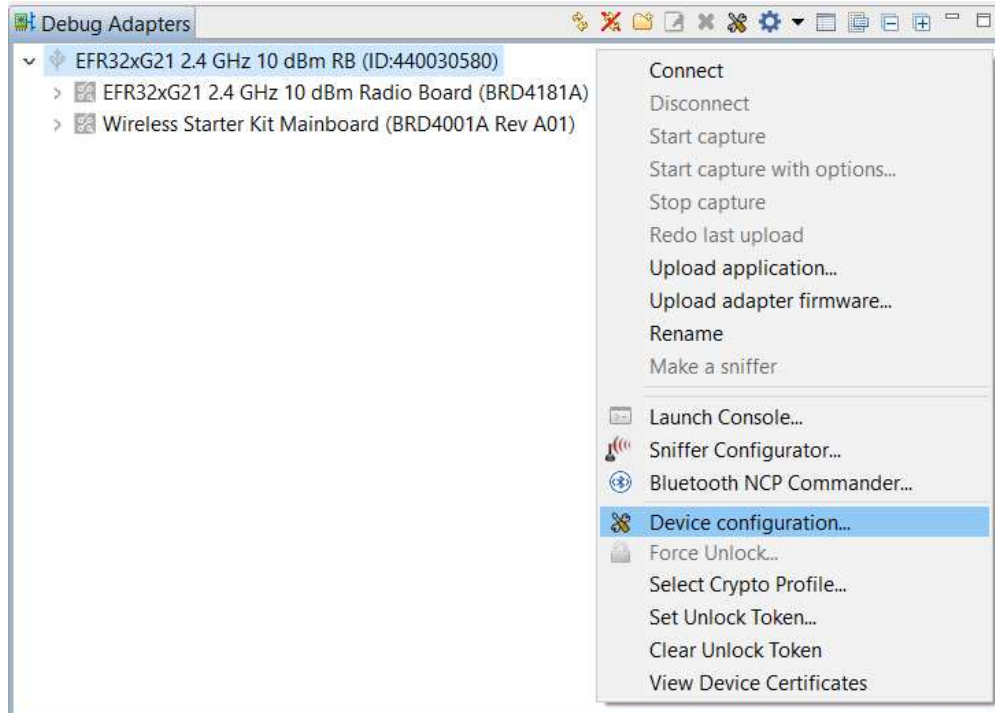
**Note:** Unless specified in the example, these examples can be applied to other Series 2 devices.

## Using Simplicity Studio

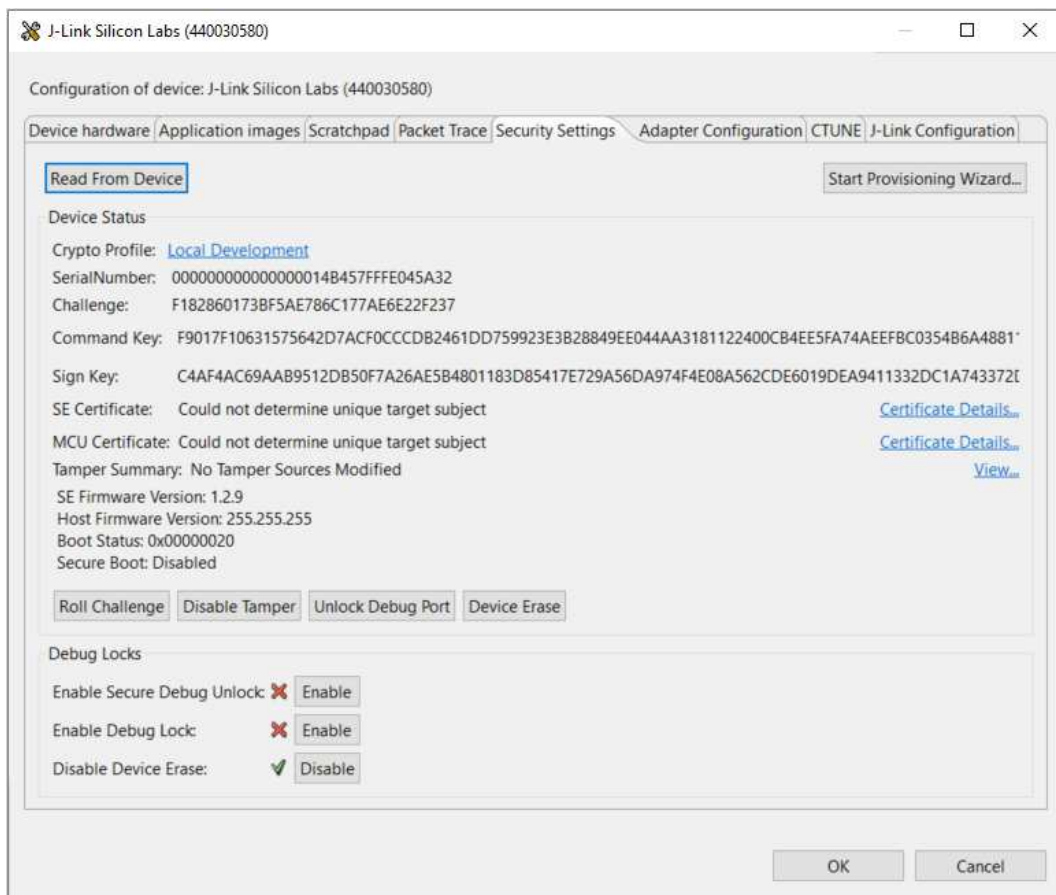
The security operations are performed in the Security Settings of Simplicity Studio. This application note uses Simplicity Studio v5.2.1.1. The procedures and pictures may be different for the other versions of Simplicity Studio 5.

1. Right-click the selected debug adapter **RB (ID:J-Link serial number)** to display the context menu.





- Click **Device configuration...** to open the **Configuration of device: J-Link Silicon Labs (serial number)** dialog box. Click the **Security Settings** tab to get the selected device configuration.





## Using Simplicity Commander

1. This application note uses Simplicity Commander v1.14.2. The procedures and console output may be different for the other versions of Simplicity Commander. The latest version of Simplicity Commander can be downloaded from <https://www.silabs.com/developers/mcu-programming-options>.

```
commander --version
```

```
Simplicity Commander 1v14p2b1232
JLink DLL version: 7.70d
Qt 5.12.10 Copyright (C) 2017 The Qt Company Ltd.
EMDLL Version: 0v18p7b669
mbed TLS version: 2.16.6
Emulator found with SN=440048205 USBAddr=0
DONE
```

2. The Simplicity Commander's Command Line Interface (CLI) is invoked by `commander.exe` in the Simplicity Commander folder. The location for Simplicity Studio 5 in Windows is `C:\SiliconLabs\SimplicityStudio\v5\developer\adapter_packs\commander`. For ease of use, it is highly recommended to add the path of `commander.exe` to the system PATH in Windows.
3. If more than one Wireless Starter Kit (WSTK) is connected via USB, the target WSTK must be specified using the `--serialno \<J-Link serial number>` option.
4. If the WSTK is in debug mode OUT, the target device must be specified using the `--device \<device name>` option.

For more information about Simplicity Commander, see [UG162: Simplicity Commander Reference Guide](#).

## Using External Tools

1. The [secure debug unlock example](#) uses the **OpenSSL** to sign the [access certificate](#) and [challenge response](#). The Windows version of OpenSSL can be downloaded from <https://slproweb.com/products/Win32OpenSSL.html>. This application note uses OpenSSL Version 1.1.1h (Win64).

```
openssl version
```

```
OpenSSL 1.1.1h 22 Sep 2020
```

- The OpenSSL's Command Line Interface (CLI) is invoked by `openssl.exe` in the OpenSSL folder. The location in Windows (Win64) is `C:\Program Files\OpenSSL-Win64\bin`. For ease of use, it is highly recommended to add the path of `openssl.exe` to the system PATH in Windows.
2. The [secure debug unlock example](#) uses the free **Hex Editor Neo** to edit the binary files generated by Simplicity Commander. The Windows version of Hex Editor Neo can be downloaded from <https://www.hhdsoftware.com/free-hex-editor>.

## Using Platform Examples

Simplicity Studio 5 includes the [SE Manager platform examples](#) for debug lock, key provisioning, and secure debug unlock. This application note uses platform examples of GSDK v4.2.1. The console output may be different on other versions of GSDK.

Refer to the corresponding `readme` file for details about each SE Manager platform example. This file also includes the procedures to create the project and run the example.

## Standard Debug Lock and Unlock

### SE Manager - Debug Lock Platform Example

Click the [View Project Documentation](#) link to open the `readme` file.

## Platform - SE Manager Host Firmware Upgrade and Debug Lock

This example project demonstrates the host firmware upgrade and debug lock API of SE Manager.

[CREATE](#)[View Project Documentation](#)

1. Press `SPACE` then `ENTER` to select the debug lock operation.

```
SE Manager Host Firmware Upgrade and Debug Lock Example - Core running at 38000 kHz.
. SE manager initialization... SL_STATUS_OK (cycles: 10 time: 0 us)
. Current selection is HOST FIRMWARE UPGRADE.
+ Press SPACE to select HOST FIRMWARE UPGRADE or DEBUG LOCK, press ENTER to run.
+ Current selection is DEBUG LOCK.
. Get debug lock status... SL_STATUS_OK (cycles: 8788 time: 231 us)
+ Debug lock: Disabled
+ Press ENTER to apply debug lock or press SPACE to exit.
```

2. Press `ENTER` again to lock the device.

```
. Apply the debug lock... SL_STATUS_OK (cycles: 52585 time: 1383 us)
+ Get debug lock status... SL_STATUS_OK (cycles: 8769 time: 230 us)
+ Debug lock: Enabled
. Current selection is DEBUG LOCK.
+ Press SPACE to select HOST FIRMWARE UPGRADE or DEBUG LOCK, press ENTER to run.
```

## Simplicity Commander

1. Run the `security status` command to get the selected device configuration.

```
commander security status --device EFR32MG21A010F1024 --serialno 440048205
```

```
SE Firmware version : 1.2.14
Serial number       : 000000000000000014b457fffe045a93
Debug lock         : Disabled
Device erase       : Enabled
Secure debug unlock : Disabled
Tamper status      : OK
Secure boot        : Disabled
Boot status        : 0x20 - OK
DONE
```

2. Run the `security lock` command to lock the selected device.

```
commander security lock --device EFR32MG21A010F1024 --serialno 440048205
```

```
WARNING: Secure debug unlock is disabled. Only way to regain debug access is to run a device erase.
Device is now locked.
DONE
```

3. Run the `security status` command again to check the device configuration.

```
commander security status --device EFR32MG21A010F1024 --serialno 440048205
```

```
SE Firmware version : 1.2.14
Serial number      : 000000000000000014b457ffe045a93
Debug lock        : Enabled
Device erase      : Enabled
Secure debug unlock : Disabled
Tamper status     : OK
Secure boot       : Disabled
Boot status      : 0x20 - OK
DONE
```

4. Run the `security erasedevice` command to unlock the selected device.

```
commander security erasedevice --device EFR32MG21A010F1024 --serialno 440048205
```

```
Successfully erased device
DONE
```

**Note:** Issue a power-on or pin reset to complete the unlock process.

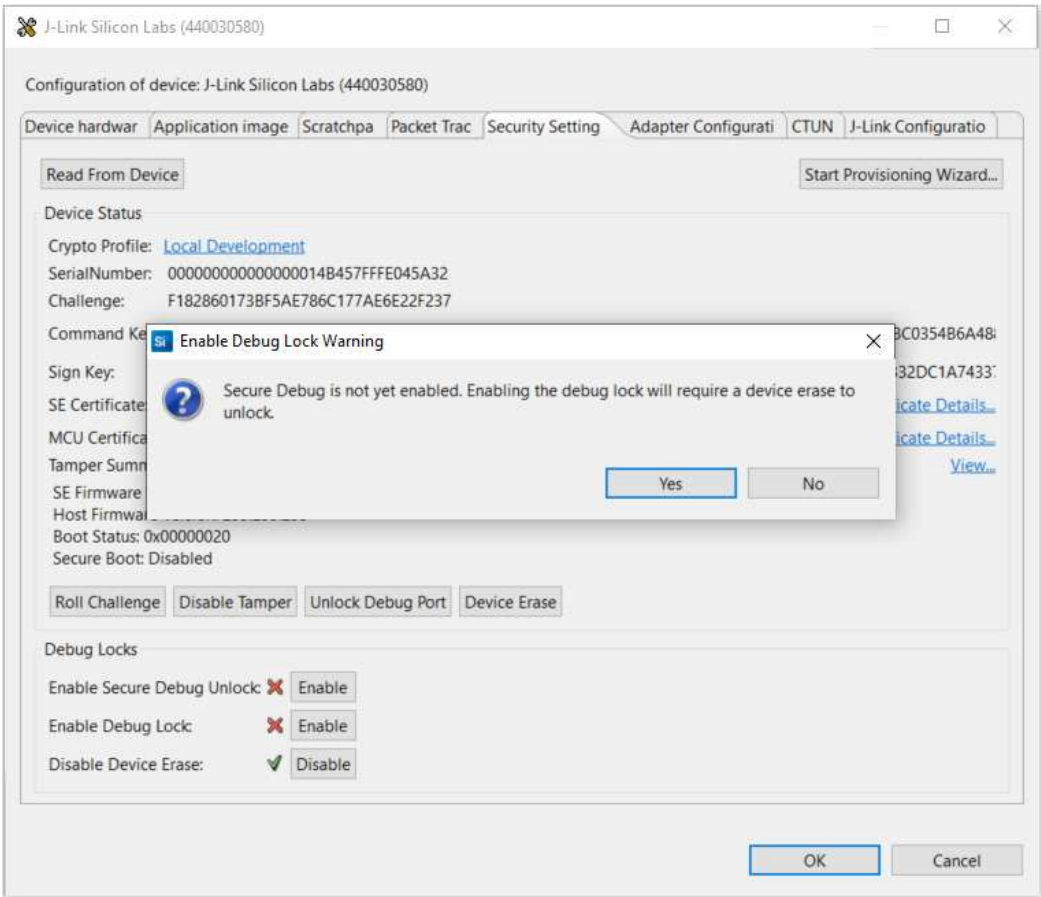
5. Run the `security status` command again to check the device configuration.

```
commander security status --device EFR32MG21A010F1024 --serialno 440048205
```

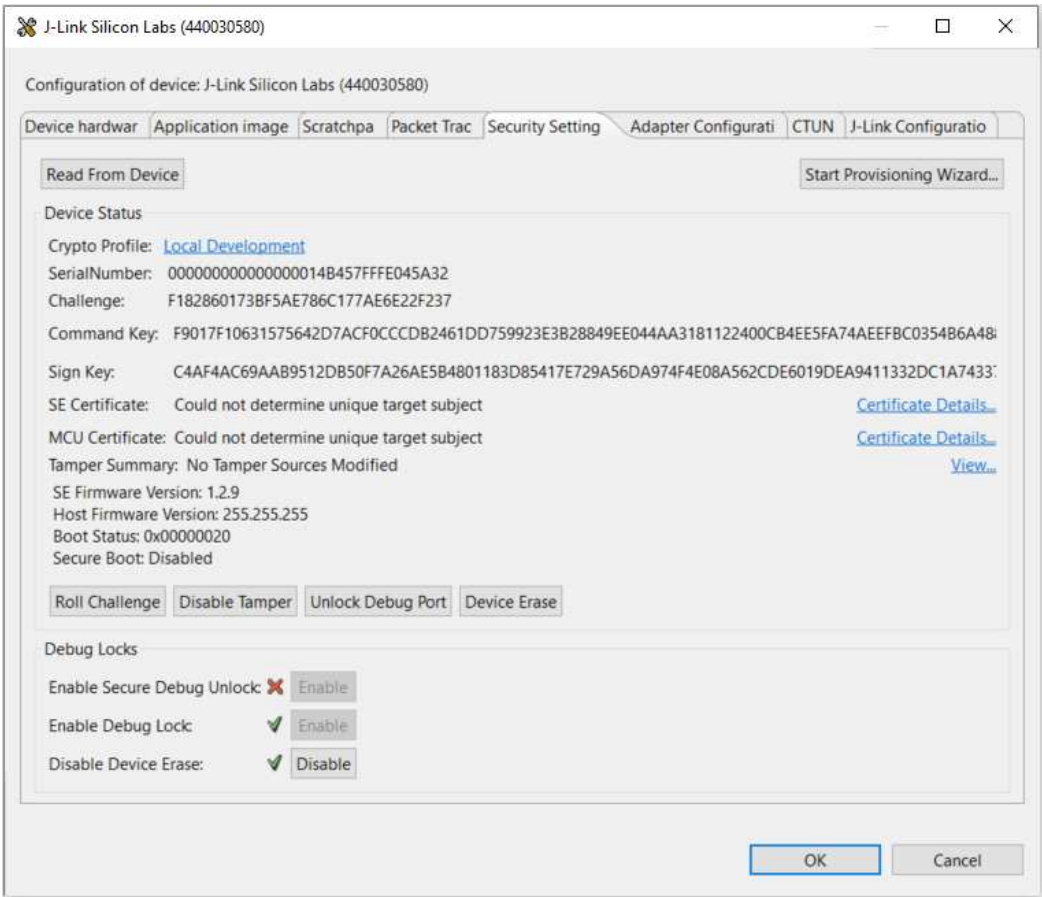
```
SE Firmware version : 1.2.14
Serial number      : 000000000000000014b457ffe045a93
Debug lock        : Disabled
Device erase      : Enabled
Secure debug unlock : Disabled
Tamper status     : OK
Secure boot       : Disabled
Boot status      : 0x20 - OK
DONE
```

## Simplicity Studio

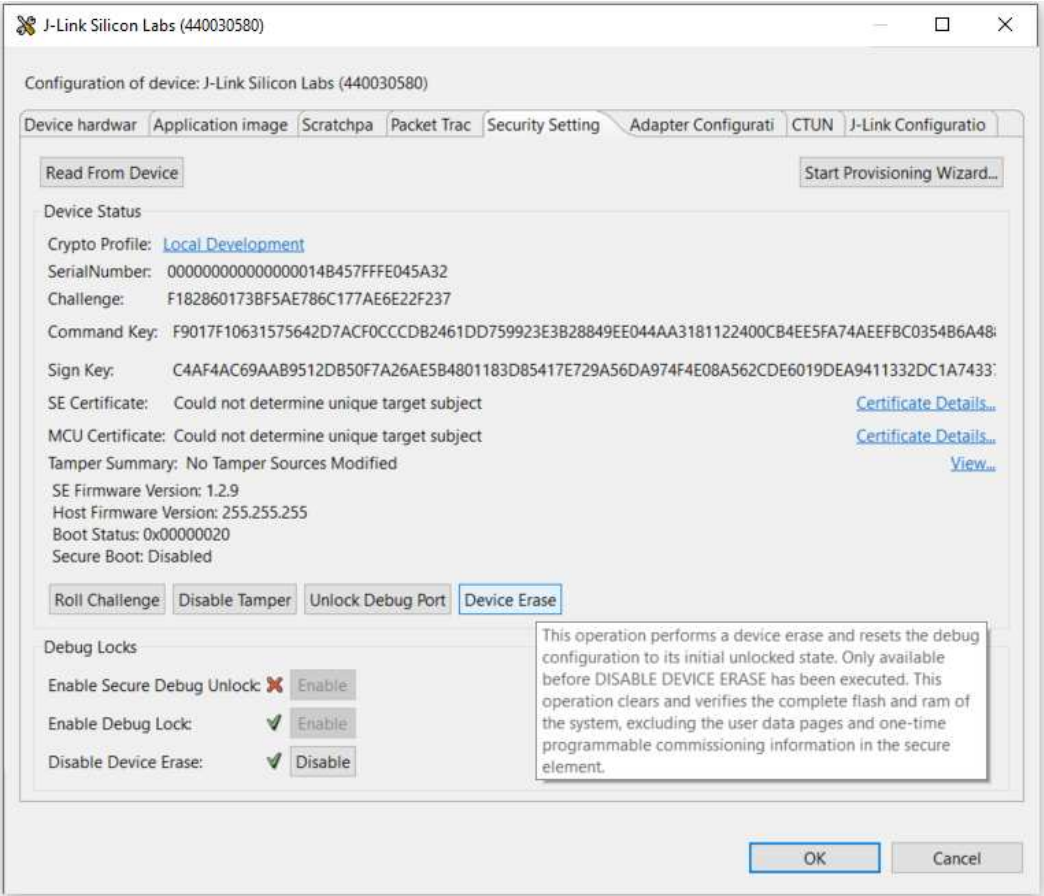
1. Open the **Security Settings** of the selected device as described in [Using Simplicity Studio](#).
2. Click **[Enable]** next to **Enable Debug Lock**: to lock the device. The following **Enable Debug Lock Warning** is displayed. Click **[Yes]** to confirm. This configures standard debug lock.



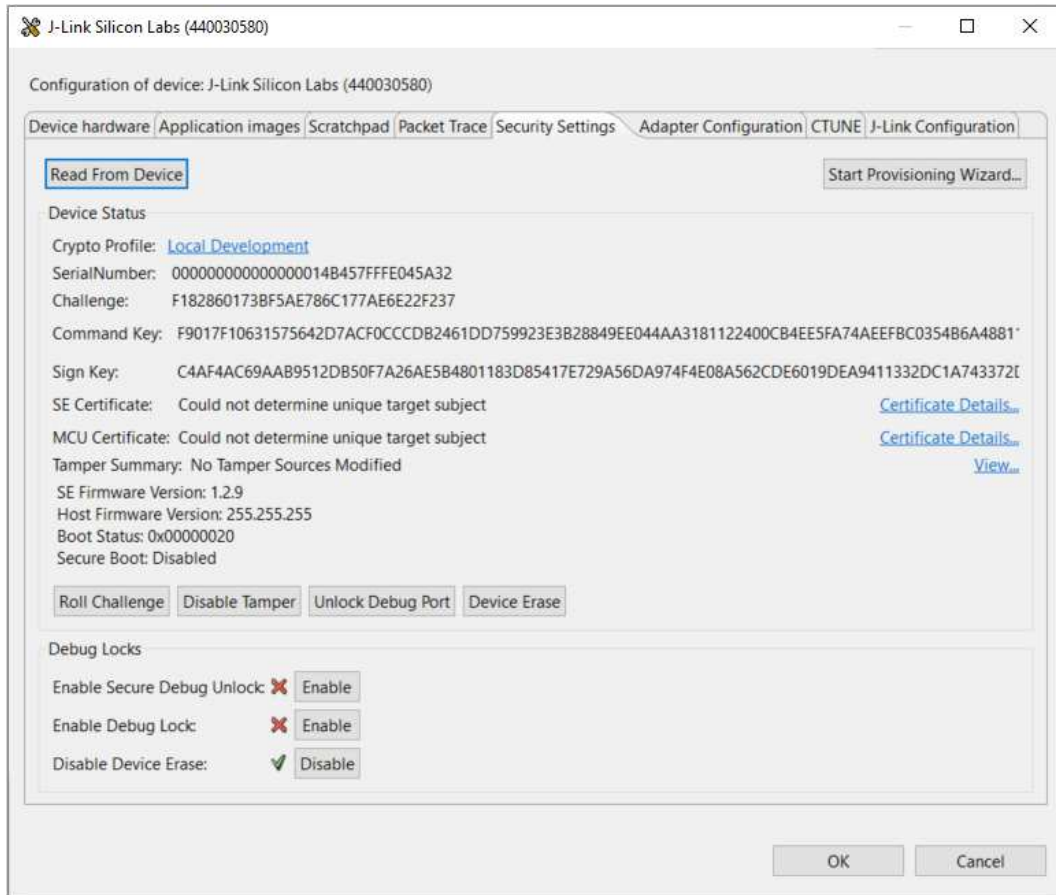
The [Enable] controls next to **Enable Secure Debug Unlock:** and **Enable Debug Lock:** are grayed out after standard debug lock is enabled.



3. Click [Device Erase] to unlock the device.



4. The device will return to the unlock state. Click [OK] to exit.



## Provision Public Command Key and Secure Debug Lock

### SE Manager - Key Provisioning Platform Example

Click the [View Project Documentation](#) link to open the `readme` file.

#### Platform - SE Manager Key Provisioning

This example project demonstrates the key provisioning API of SE Manager.

[View Project Documentation](#)

[CREATE](#)

1. Press `SPACE` to skip the programming of AES-128 key.

```
SE Manager Key Provisioning Example - Core running at 38000 kHz.
. SE manager initialization... SL_STATUS_OK (cycles: 9 time: 0 us)
. Get current SE firmware version... SL_STATUS_OK (cycles: 3578 time: 94 us)
+ Current SE firmware version (MSB..LSB): 00010209
. Read SE OTP configuration... SL_STATUS_COMMAND_IS_INVALID (cycles: 3908 time: 102 us)
. Press ENTER to program 128-bit AES key in SE OTP or press SPACE to skip.
. Encrypt 16 bytes plaintext with 128-bit AES OTP key... SL_STATUS_FAIL (cycles: 4627 time: 121 us)
. Press ENTER to program public sign key in SE OTP or press SPACE to skip.
```

2. Press `SPACE` to skip the programming of Public Sign Key.

```
. Get public sign key... SL_STATUS_FAIL (cycles: 4144 time: 109 us)
. Press ENTER to program public command key in SE OTP or press SPACE to skip.
```

3. Press `ENTER` to program the default Public Command Key in flash to the SE OTP.

```
+ Warning: The public command key in SE OTP cannot be changed once written!
+ Press ENTER to confirm or press SPACE to skip if you are not sure.
```

4. Press `ENTER` to confirm the operation.

```
. Initialize public command key... SL_STATUS_OK (cycles: 56052 time: 1475 us)
. Get public command key... SL_STATUS_OK (cycles: 7135 time: 187 us)
+ The public command key (64 bytes):
B1 BC 6F 6F A5 66 40 ED 52 2B 2E E0 F5 B3 CF 7E
5D 48 F6 0B E8 14 8F 0D C0 84 40 F0 A4 E1 DC A4
7C 04 11 9E D6 A1 BE 31 B7 70 7E 5F 9D 00 1A 65
9A 05 10 03 E9 5E 1B 93 6F 05 C3 7E A7 93 AD 63
. Press ENTER to initialize SE OTP for secure boot configuration or press SPACE to skip.
```

5. Press `SPACE` to skip the secure boot configuration.

```
. SE manager deinitialization... SL_STATUS_OK (cycles: 7 time: 0 us)
```

### SE Manager - Secure Debug Platform Example

Click the `View Project Documentation` link to open the `readme` file.

**Note:** The secure debug platform example can only run on the HSE device.

#### Platform - SE Manager Secure Debug

This example project demonstrates the secure debug API of SE Manager.

[View Project Documentation](#)

CREATE

1. Use a [standard debug unlock](#) device with matched Public Command Key.



```
SE Manager Secure Debug Example - Core running at 38000 kHz.
. SE manager initialization... SL_STATUS_OK (cycles: 9 time: 0 us)
. Get SE status... SL_STATUS_OK (cycles: 8496 time: 223 us)
+ The SE firmware version (MSB..LSB): 0001020E
+ Debug lock: Disabled
+ Device Erase: Enabled
+ Secure debug: Disabled
+ Secure boot: Disabled
+ Debug lock state: Unlocked
+ Non-secure, Invasive debug lock (DBGLOCK) configuration: Unlocked
+ Non-secure, Non-invasive debug lock (NIDLOCK) configuration: Unlocked
+ Secure, Invasive debug lock (SPIDLOCK) configuration: Unlocked
+ Secure, Non-invasive debug lock (SPNIDLOCK) configuration: Unlocked
+ Non-secure, Invasive debug lock (DBGLOCK) current state: Unlocked
+ Non-secure, Non-invasive debug lock (NIDLOCK) current state: Unlocked
+ Secure, Invasive debug lock (SPIDLOCK) current state: Unlocked
+ Secure, Non-invasive debug lock (SPNIDLOCK) current state: Unlocked. The device is in normal state and secure debug is disabled.
+ Exporting a public command key from a hard-coded private command key... SL_STATUS_OK (cycles: 202467 time: 5328 us)
+ Reading the public command key from SE OTP... SL_STATUS_OK (cycles: 7589 time: 199 us)
+ Comparing exported public command key with SE OTP public command key... OK
+ Press ENTER to enable secure debug or press SPACE to exit.
```

2. Press `ENTER` to enable the secure debug.

```
+ Enable the secure debug... SL_STATUS_OK (cycles: 48313 time: 1271 us)
+ Press ENTER to lock the device or press SPACE to disable the secure debug and exit.
```

3. Press `ENTER` to lock the device with `debug options 0x0c`.

```
+ Setting the debug options (0xc)... SL_STATUS_OK (cycles: 51091 time: 1344 us)
+ Locking the device... SL_STATUS_OK (cycles: 89683 time: 2360 us)
+ Device erase is enabled, press ENTER to disable device erase (optional if just for testing) or press SPACE to skip.
```

4. Press `ENTER` to disable device erase.

```
+ Warning: This is a ONE-TIME command which PERMANENTLY disables device erase!
+ Press ENTER to confirm or press SPACE to skip if you are not sure.
```

5. Press `ENTER` to confirm the operation.

```
. Get SE status... SL_STATUS_OK (cycles: 8496 time: 223 us)
+ The SE firmware version (MSB..LSB): 0001020E
+ Debug lock: Enabled
+ Device Erase: Disabled
+ Secure debug: Enabled
+ Secure boot: Disabled
+ Debug lock state: Locked
+ Non-secure, Invasive debug lock (DBGLOCK) configuration: Unlocked
+ Non-secure, Non-invasive debug lock (NIDLOCK) configuration: Unlocked
+ Secure, Invasive debug lock (SPIDLOCK) configuration: Locked
+ Secure, Non-invasive debug lock (SPNIDLOCK) configuration: Locked
+ Non-secure, Invasive debug lock (DBGLOCK) current state: Unlocked
+ Non-secure, Non-invasive debug lock (NIDLOCK) current state: Unlocked
+ Secure, Invasive debug lock (SPIDLOCK) current state: Locked
+ Secure, Non-invasive debug lock (SPNIDLOCK) current state: Locked. The device is in secure debug lock state.
+ Press ENTER to issue a secure debug unlock or press SPACE to exit.
```

6. Press `SPACE` to exit.

```
. SE manager deinitialization... SL_STATUS_OK (cycles: 9 time: 0 us)
```

## Simplicity Commander

1. Run the `security status` command to get the selected device configuration.

```
commander security status --device EFR32MG21A010F1024 --serialno 440048205
```

```
SE Firmware version : 1.2.14
Serial number      : 000000000000000014b457fffe045a93
Debug lock        : Disabled
Device erase      : Enabled
Secure debug unlock : Disabled
Tamper status     : OK
Secure boot       : Disabled
Boot status      : 0x20 - OK
DONE
```

2. Run the `security writekey` command to provision the Public Command Key (e.g., `command_pubkey.pem`).

```
commander security writekey --command command_pubkey.pem --device EFR32MG21A010F1024 --serialno 440048205
```

```
Device has serial number 000000000000000014b457fffe045a93
=====
Please look through any warnings before proceeding.
THIS IS A ONE-TIME command which permanently ties debug and tamper access to certificates signed by this key.
Type 'continue' and hit enter to proceed or Ctrl-C to abort:
=====
continue
DONE
```

**Note:** The Public Command Key cannot be changed once written.

3. Run the `security readkey` command to read the Public Command Key from the SE OTP.

```
commander security readkey --command --device EFR32MG21A010F1024 --serialno 440048205
```

```
B1BC6F6FA56640ED522B2EE0F5B3CF7E5D48F60BE8148F0DC08440F0A4E1DCA4
7C04119ED6A1BE31B7707E5F9D001A659A051003E95E1B936F05C37EA793AD63
DONE
```

4. Run the `security lockconfig` command to enable the secure debug.

```
commander security lockconfig --secure-debug-unlock enable --device EFR32MG21A010F1024 --serialno 440048205
```

```
Secure debug unlock was enabled
DONE
```

5. a. For the **TrustZone-unaware** application, run the `security lockcommand` to lock the selected device.

```
commander security lock --device EFR32MG21A010F1024 --serialno 440048205
```

```
Device is now locked.
DONE
```

- b. For the **TrustZone-aware** application, run the `security lock --trustzone #####` command to set the [debug options](#) (e.g., `1100`) and lock the selected device. The bit order of `#####` is SPNIDLOCK (MSB), SPIDLOCK, NIDLOCK, and DBGLOCK (LSB).

```
commander security lock --trustzone 1100 --device EFR32MG21A010F1024 --serialno 440048205
```

```
Writing debug restriction bits:
DBGLOCK: 0
NIDLOCK: 0
SPIDLOCK: 1
SPNIDLOCK: 1
Device is now locked.
DONE
```

**Notes:**

- The `--trustzone` option for the `security lock` command requires Simplicity Commander  $\geq$  **v1.13.3**.
  - It is strongly recommended to [upgrade](#) to SE firmware  $\geq$  **v1.2.14** (xG21 and xG22) or  $\geq$  **v2.2.1** (other Series 2 devices) so that the debug options cannot be modified after the device is locked.
  - Use `commander security lock` without the `--trustzone #####` option if the default setting of debug options ( `0000` ) is good enough for a TrustZone-aware application.
6. Run the `security disabledeviceerase` command to disable device erase. This is an **IRREVERSIBLE** action, and should be the last step in production.

```
commander security disabledeviceerase --device EFR32MG21A010F1024 --serialno 440048205
```

```
=====
THIS IS A ONE-TIME command which Permanently disables device erase.
If secure debug lock has not been set, there is no way to regain debug access to this device.
Type 'continue' and hit enter to proceed or Ctrl-C to abort:
=====
continue
Disabled device erase successfully
DONE
```

**Note:** The [debug options](#) cannot be reset to the default value `0000` (unlock) if the device erase option is disabled.

7. a. For Simplicity Commander  $<$  **v1.13.3**, run the `security status` command to check the debug lock status of the device.

```
commander security status --device EFR32MG21A010F1024 --serialno 440048205
```

```
SE Firmware version : 1.2.14
Serial number      : 000000000000000014b457fffe045a93
Debug lock        : Enabled
Device erase      : Disabled
Secure debug unlock : Enabled
Tamper status     : OK
Secure boot       : Disabled
Boot status      : 0x20 - OK
DONE
```

- b. For Simplicity Commander  $\geq$  **v1.13.3**, run the `security status --trustzone` command to check the full debug lock status of the device.

```
commander security status --trustzone --device EFR32MG21A010F1024 --serialno 440048205
```

```

SE Firmware version : 1.2.14
Serial number      : 000000000000000014b457ffe045a93
Debug lock        : Enabled
Device erase       : Disabled
Secure debug unlock : Enabled
Debug lock state: Locked
Non-secure, invasive debug lock (DBGLOCK) : Unlocked
Non-secure, non-invasive debug lock (NIDLOCK) : Unlocked
Secure, invasive debug lock (SPIDLOCK) : Locked
Secure, non-invasive debug lock (SPNIDLOCK) : Locked
Non-secure, invasive debug lock state (DBGLOCK) : Unlocked
Non-secure, non-invasive debug lock state (NIDLOCK) : Unlocked
Secure, invasive debug lock state (SPIDLOCK) : Locked
Secure, non-invasive debug lock state (SPNIDLOCK) : Locked
Tamper status      : OK
Secure boot        : Disabled
Boot status        : 0x20 - OK
DONE

```

## Simplicity Studio

1. Run the `util keytotoken` command to convert the Public Command Key file (PEM format) into a text file ( `command_pubkey.txt` ).

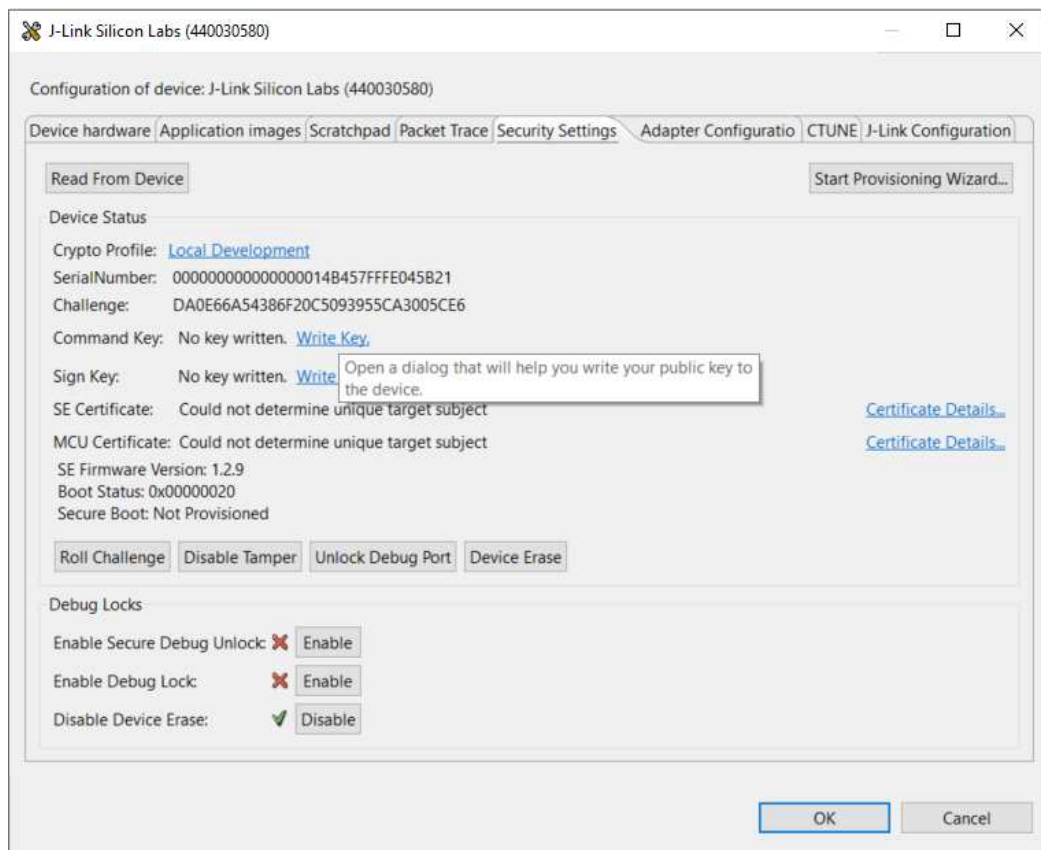
```
commander `util keytotoken` command_pubkey.pem --outfile command_pubkey.txt
```

```

Writing EC tokens to command_pubkey.txt...
DONE

```

2. Open **Security Settings** of the selected device as described in [Using Simplicity Studio](#).
3. Click the **WriteKey** link next to **Command Key**: to open a dialog box.



4. The **Write Command Key** dialog box is displayed.



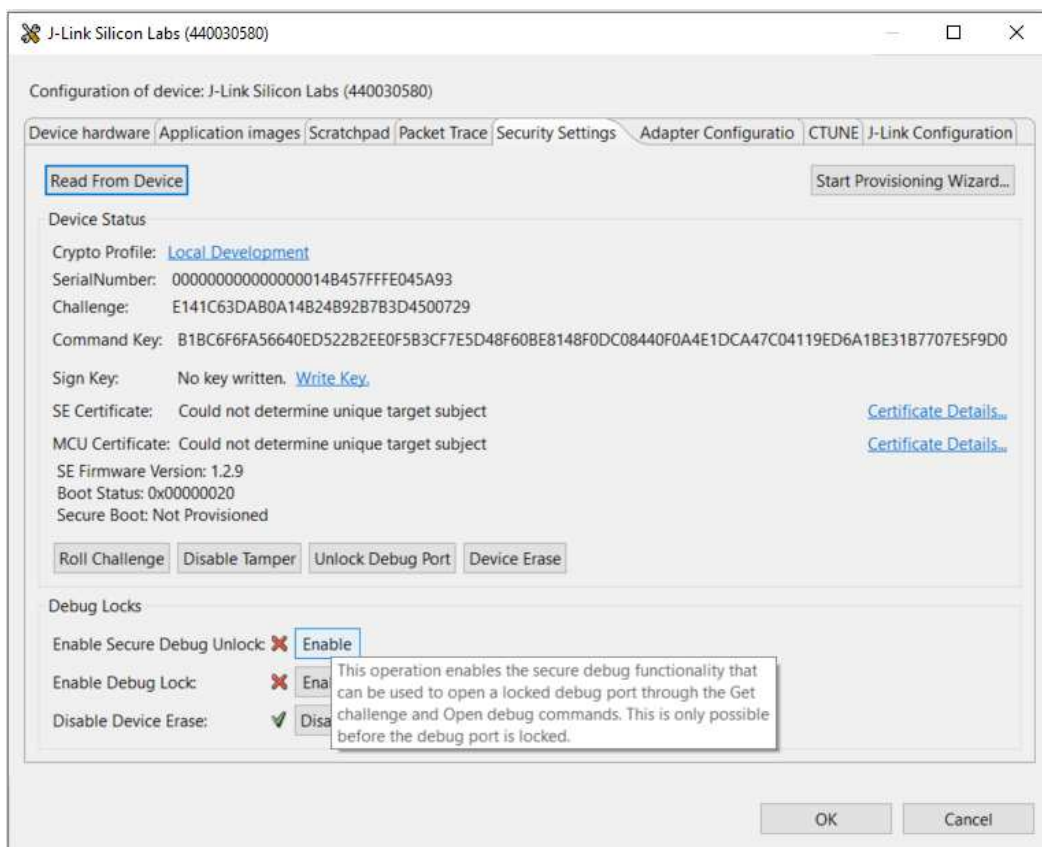
5. Open the `command_pubkey.txt` file generated in step 1.

```
MFG_SIGNED_BOOTLOADER_KEY_X : B1BC6F6FA56640ED522B2EE0F5B3CF7E5D48F60BE8148F0DC08440F0A4E1DCA4
MFG_SIGNED_BOOTLOADER_KEY_Y : 7C04119ED6A1BE31B7707E5F9D001A659A051003E95E1B936F05C37EA793AD63
```

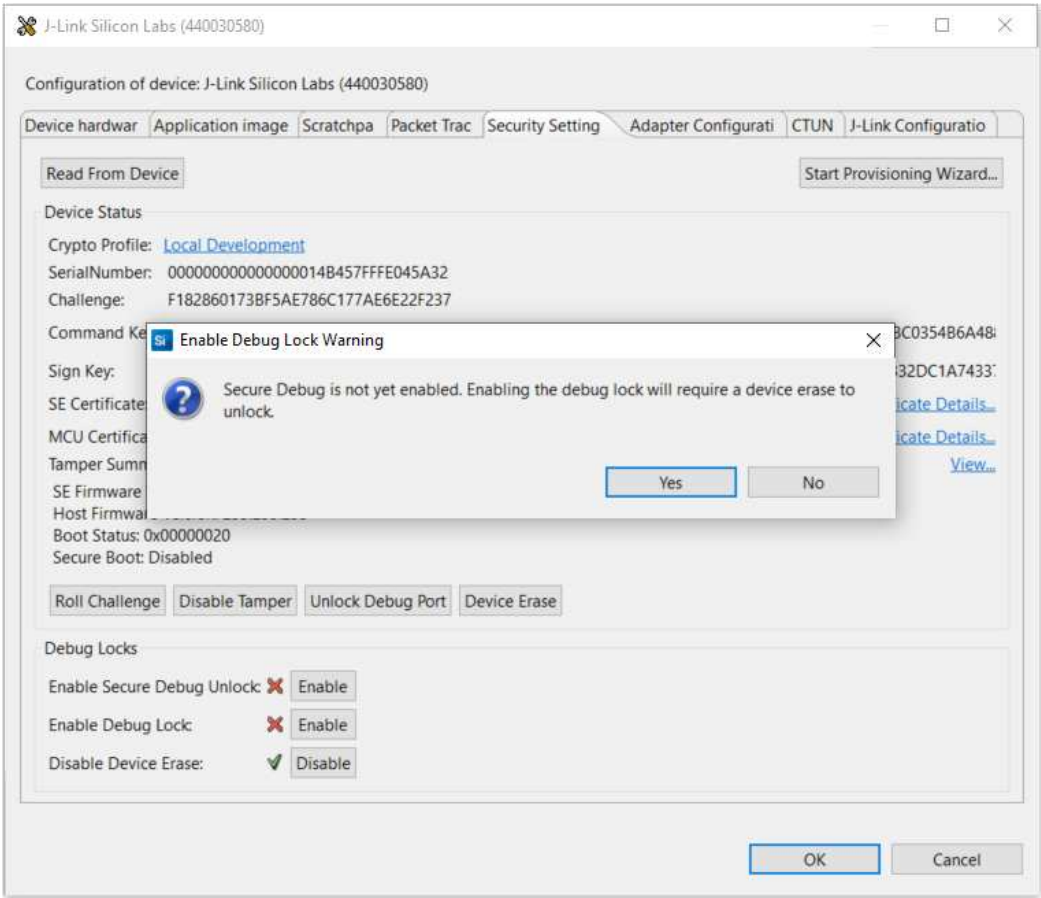
6. Copy Public Command Key (X-point `B1BC...` first, then Y-point `7C04...`) to **Command Key:** box.



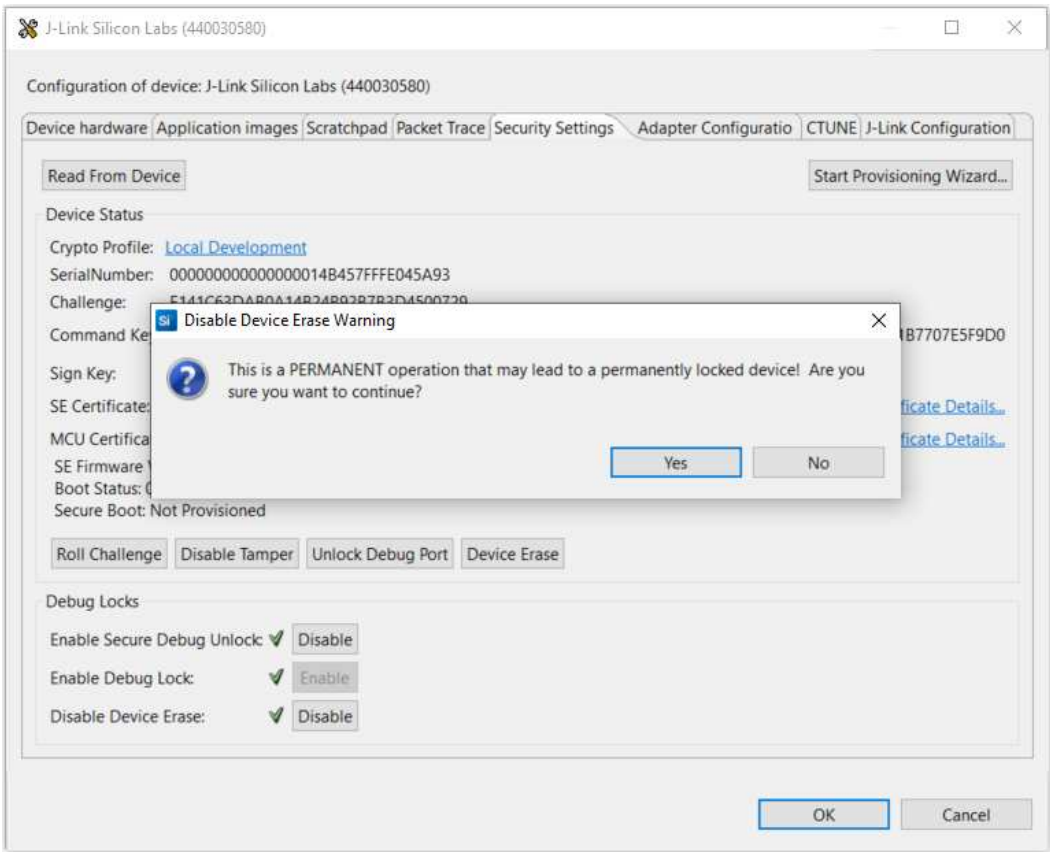
7. Click **[Write]** to provision the Public Command Key.  
 8. Click **[Enable]** next to **Enable Secure Debug Unlock:** to enable the secure debug functionality.



Click [Enable] next to **Enable Debug Lock**: to lock the device. This configures secure debug lock.



10. Click [Disable] next to **Disable Device Erase**: to disable the device erase. The following **Disable Device Erase Warning** is displayed. Click [Yes] to confirm.



Note: This is an IRREVERSIBLE action, and should be the last step in production.

## Secure Debug Unlock and Roll Challenge

### SE Manager - Secure Debug Platform Example

Click the [View Project Documentation](#) link to open the `readme` file.

Note: The secure debug platform example can only run on the HSE device.

#### Platform - SE Manager Secure Debug

This example project demonstrates the secure debug API of SE Manager.

[View Project Documentation](#)

CREATE

1. Use a [secure debug lock](#) device with matched Public Command Key.



```
. Get SE status... SL_STATUS_OK (cycles: 8496 time: 223 us)
+ The SE firmware version (MSB..LSB): 0001020E
+ Debug lock: Enabled
+ Device Erase: Disabled
+ Secure debug: Enabled
+ Secure boot: Disabled
+ Debug lock state: Locked
+ Non-secure, Invasive debug lock (DBGLOCK) configuration: Unlocked
+ Non-secure, Non-invasive debug lock (NIDLOCK) configuration: Unlocked
+ Secure, Invasive debug lock (SPIDLOCK) configuration: Locked
+ Secure, Non-invasive debug lock (SPNIDLOCK) configuration: Locked
+ Non-secure, Invasive debug lock (DBGLOCK) current state: Unlocked
+ Non-secure, Non-invasive debug lock (NIDLOCK) current state: Unlocked
+ Secure, Invasive debug lock (SPIDLOCK) current state: Locked
+ Secure, Non-invasive debug lock (SPNIDLOCK) current state: Locked. The device is in secure debug lock state.
+ Press ENTER to issue a secure debug unlock or press SPACE to exit.
```

2. Press `ENTER` to unlock the device with `debug mode request 0x3e`.

```
+ Creating a private certificate key in a buffer... SL_STATUS_OK (cycles: 202354 time: 5325 us)
+ Exporting a public certificate key from a private certificate key... SL_STATUS_OK (cycles: 199394 time: 5247 us)
+ Read the serial number of the SE and save it to access certificate... SL_STATUS_OK (cycles: 7084 time: 186 us)
+ Signing the access certificate with private command key... SL_STATUS_OK (cycles: 221849 time: 5838 us)
+ Request challenge from the SE and save it to challenge response... SL_STATUS_OK (cycles: 4418 time: 116 us)
+ Signing the challenge response with private certificate key... SL_STATUS_OK (cycles: 220833 time: 5811 us)
+ Creating an unlock token (DEBUG_MODE_REQUEST = 0x3e) to unlock the device... SL_STATUS_OK (cycles: 935778 time: 24625 us)
+ Get debug status to verify the device is unlocked... SL_STATUS_OK (cycles: 9017 time: 237 us)
+ Success to unlock the device!
. Get SE status... SL_STATUS_OK (cycles: 8683 time: 228 us)
+ The SE firmware version (MSB..LSB): 0001020D
+ Debug lock: Enabled
+ Device Erase: Enabled
+ Secure debug: Enabled
+ Secure boot: Disabled
+ Debug lock state: Unlocked
+ Non-secure, Invasive debug lock (DBGLOCK) configuration: Unlocked
+ Non-secure, Non-invasive debug lock (NIDLOCK) configuration: Unlocked
+ Secure, Invasive debug lock (SPIDLOCK) configuration: Locked
+ Secure, Non-invasive debug lock (SPNIDLOCK) configuration: Locked
+ Non-secure, Invasive debug lock (DBGLOCK) current state: Unlocked
+ Non-secure, Non-invasive debug lock (NIDLOCK) current state: Unlocked
+ Secure, Invasive debug lock (SPIDLOCK) current state: Unlocked
+ Secure, Non-invasive debug lock (SPNIDLOCK) current state: Unlocked. The device is in secure debug unlock state.
+ Issue a power-on or pin reset to re-enable the secure debug lock.
+ Press ENTER to roll the challenge to invalidate the current unlock token or press SPACE to exit.
```

3. Press `ENTER` to roll the challenge.

```
. Check and roll the challenge.
+ Request current challenge from the SE... SL_STATUS_OK (cycles: 4450 time: 117 us)
+ The current challenge (16 bytes):
  FA A7 AA 5E EF E6 18 23 E5 21 89 84 DB 7E 52 7D
+ Rolling the challenge... SL_STATUS_OK (cycles: 19757 time: 519 us)
+ Request rolled challenge from the SE... SL_STATUS_OK (cycles: 4628 time: 121 us)
+ The rolled challenge (16 bytes):
  5A 7A 81 CC 6E 46 C1 EF B4 A4 CA 7A DD A9 85 EB
+ Issue a power-on or pin reset to activate the rolled challenge.
. SE manager deinitialization... SL_STATUS_OK (cycles: 9 time: 0 us)
```

## Simplicity Commander

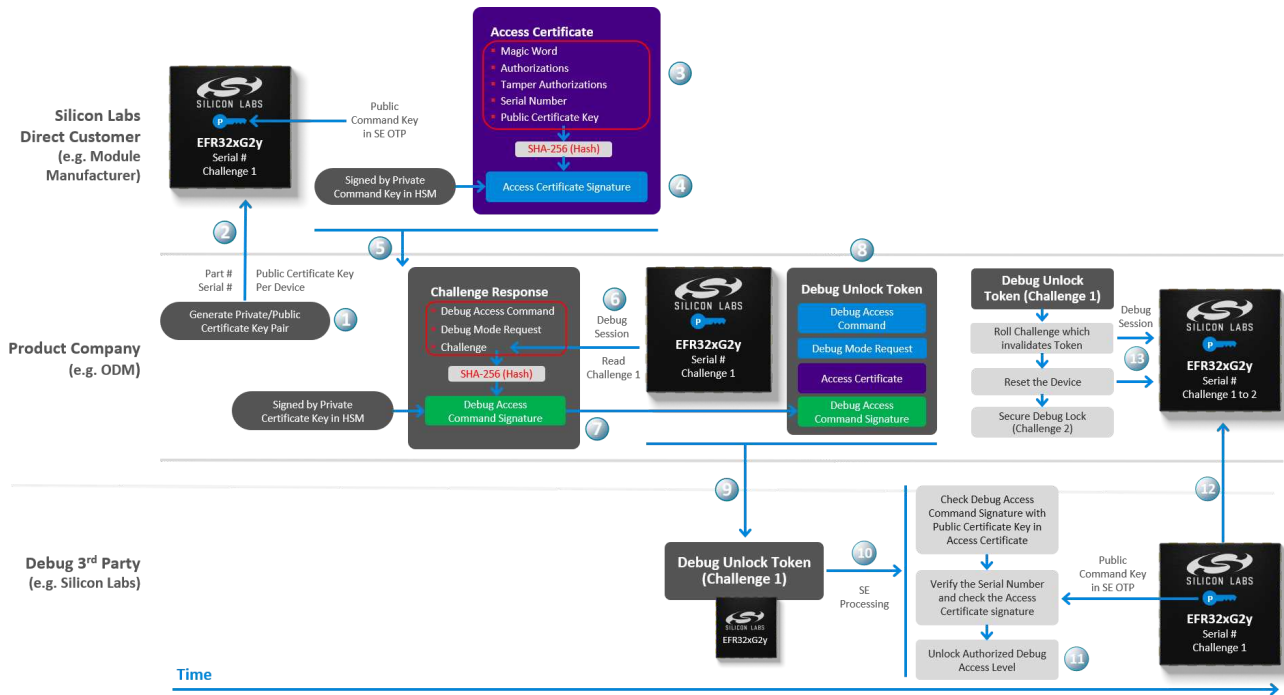
The secure debug was designed with three organizations in mind:



- Direct Customer to whom Silicon Labs sells the chip. This chip has the Public Command Key installed in the SE OTP.
- That Direct Customer may be creating a white-labeled product for another company or a sub-component that goes into another company's product. The Product Company is the customer of the direct customer.
- The Debug 3rd Party could be anyone, internal or external, that the Product Company decides is qualified to debug the device.

Because the Public Command Key is installed into the SE OTP of a large number of devices and cannot be changed, the corresponding Private Command Key must be guarded by a very stringent process. If this Private Command Key is ever leaked, all the devices programmed with the corresponding Public Command Key will be compromised.

A secure debug unlock user case is described in the following figure.



The secure debug unlock flow moving across the time axis from left to right is explained below:

- In this example, the Private/Public Certificate Key pair ( `cert_key.pem` and `cert_pubkey.pem` ) is generated by running the `util genkey` command.

```
Generating ECC P256 key pair...
Writing private key file in PEM format to cert_key.pem
Writing public key file in PEM format to cert_pubkey.pem
DONE
```

- If necessary, run the `security status` command to get the device serial number.

```
SE Firmware version : 1.2.14
Serial number      : 000000000000000014b457ffe045a32Debug lock      : Enabled
Device erase      : Disabled
Secure debug unlock : Enabled
Tamper status      : OK
Secure boot        : Disabled
Boot status        : 0x20 - OK
DONE
```

- Device part number
- Device serial number
- Public Certificate Key

```
Authorization file written to Security Store:
C:/Users/<username>/AppData/Local/SiliconLabs/commander/SecurityStore/device_000000000000000014b457ffe045a32/certificate_authorization.der

Cert key written to Security Store:
C:/Users/<username>/AppData/Local/SiliconLabs/commander/SecurityStore/device_000000000000000014b457ffe045a32/cert_pubkey.pem

Created an unsigned certificate in Security Store:
C:/Users/<username>/AppData/Local/SiliconLabs/commander/SecurityStore/device_000000000000000014b457ffe045a32/access_certificate.der

DONE
```

- The `--extsign` option to create an unsigned access certificate is only available in Simplicity Commander Version 1.11.2 or above.
- The unsigned access certificate is generated with the default certificate authorization file (`certificate_authorization.json`) which uses `0x00000003e` for Authorizations and `0x000000000` (HSE-SVM device) for Tamper Authorizations ([Table Elements of the Access Certificate on page 12](#)).

- In this example, the OpenSSL is used to sign the access certificate ( `access_certificate.extsign` ) in Security Store with the Private Command Key ( `command_key.pem` ). The [access certificate signature](#) is in the `cert_signature.binfile`.

Run the `util signcert` command with the following parameters to verify the signature and generate the signed access certificate ( `access_certificate.bin` ):

- Unsigned access certificate
- Access certificate signature
- Public Command Key

```
commander util signcert access_certificate.extsign --cert-type access --signature cert_signature.bin
--verify command_pubkey.pem --outfile access_certificate.bin
```

```
R = D97E43FEA278207080D6D0808B46810C1167F123AF1CA9FAF2DE0F4322B97ACE
S = FEDFEA11A3C83AFFCD5293283B13A50580862B9F651AAE08012C2BFB6BA8E697
Successfully verified signature
Successfully signed certificate
DONE
```

#### Notes:

- Put the required files in the same folder to run the command.
  - The `util signcert` command for access certificate is only available in Simplicity Commander Version 1.11.2 or above.
  - The access certificate signature can be in a Raw or Distinguished Encoding Rules (DER) format.
- The access certificate is passed to the Product Company. The purpose of the access certificate is to grant overall debug access capabilities to the Product Company and authorize them to allow third parties to debug the device. The Product Company can now use the access certificate to generate the [Debug Unlock Token](#). The same access certificate can be used to generate as many Debug Unlock Tokens as necessary without having to ever go back to the Direct Customer.
  - To create the Debug Unlock Token, a debug session must be started with the device and the challenge value (which is a random number `Challenge 1` in this example) should be read out to generate the [challenge response](#).  
Run the `security gencommand` command to generate the challenge response without [debug access command signature](#) and store it in a file ( `command_unsign.bin` ).

```
commander security gencommand --action debug-unlock--unlock-param 1111 -o command_unsign.bin --nostore
--device EFR32MG21A010F1024 --serialno 440048205
```

```
Unsigned command file written to:
command_unsign.bin
DONE
```

#### Notes:

- The data in the `--unlock-param` option are the bits 2 to 5 of [debug mode request](#) in the [challenge response](#).
  - The default value 1111(reset all debug options) is in place if the `security gencommand` command does not include the `--unlock-param` option.
- The challenge response is then cryptographically hashed (SHA-256) to create a digest. The digest is then signed by the Private Certificate Key to generate the debug access command signature.  
The signing of the challenge response can be done by passing an unsigned challenge response to a Hardware Security Module (HSM) containing the Private Certificate Key.  
In this example, the OpenSSL is used to sign the challenge response ( `command_unsign.bin` ) with the Private Certificate Key ( `cert_key.pem` ). The debug access command signature is in the `command_signature.bin` file.

```
openssl dgst -sha256 -binary -sign cert_key.pem -out command_signature.bincommand_unsign.bin
```

- Run the `security unlock` command with the access certificate ( `access_certificate.bin` ) from Direct Customer and debug access command signature ( `command_signature.bin` ) in step 7 to generate the Debug Unlock Token.

```
commander security unlock --cert access_certificate.bin --command-signature command_signature.bin--unlock-param 1111 --device
EFR32MG21A010F1024 --serialno 440048205
```

```
Certificate written to Security Store:
C:/Users/<username>/AppData/Local/SiliconLabs/commander/SecurityStore/device_00000000000000014b457ffe045a32/access_certificate

R = 67B51151F1E5F1BB9A49EC8D5885B221BD3D31D53741EEF54F81F0F3CB40455
S = 066C6AB5EDE3AE784DB1F75F44C5CA931736116D5A2104DBF44BC77ED8F49282
Command signature is valid
Secure debug successfully unlocked
Command unlock payload was stored in Security Store
DONE
```

**Notes:**

- Put the required files in the same folder to run the command.
- The debug access command signature can be in a Raw or Distinguished Encoding Rules (DER) format.
- It requires Simplicity Commander Version 1.11.2 or above to support signature in DER format.
- The data in the `--unlock-param` option are the bits 2 to 5 of [debug mode request](#) in the [Debug Unlock Token](#). This value **MUST** be equal to the value of `--unlock-param` option in step 6.
- The default value `1111` (reset all debug options) is in place if the `security unlock` command does not include the `--unlock-param` option.

9. **(Alternative)** The key protection is not required if the Private Certificate Key is ephemeral. Steps 6 to 8 can be implemented by running the `security unlock` command with the access certificate ( `access_certificate.bin` ) from the Direct Customer and Private Certificate Key ( `cert_key.pem` ) to generate the Debug Unlock Token.

```
commander security unlock --cert access_certificate.bin --cert-privkey cert_key.pem--unlock-param 1111
--device EFR32MG21A010F1024 --serialno 440048205
```

```
Certificate written to Security Store:
C:/Users/<username>/AppData/Local/SiliconLabs/commander/SecurityStore/device_00000000000000014b457ffe045a32/access_certificate

Cert key written to Security Store:
C:/Users/<username>/AppData/Local/SiliconLabs/commander/SecurityStore/device_00000000000000014b457ffe045a32/cert_pubkey.pem


Created unsigned unlock command
Signed unlock command using
C:/Users/<username>/AppData/Local/SiliconLabs/commander/SecurityStore/device_00000000000000014b457ffe045a32/cert_key.pem
Secure debug successfully unlocked
Command unlock payload was stored in Security Store
DONE
```

**Notes:**

- The data in the `--unlock-param` option are the bits 2 to 5 of [debug mode request](#) in the [Debug Unlock Token](#).
- The default value `1111` (reset all debug options) is in place if the `security unlock` command does not include the `--unlock-param` option.

10. The Debug Unlock Token (aka `Command unlock payload` ) file ( `unlock_payload_00000000000111110.bin` , where `00000000000111110` is the value of [debug mode request](#) ) is stored in the Security Store. The location in Windows is `C:\Users\<PC user name>\AppData\Local\SiliconLabs\commander\SecurityStore\device_\<Serial number>\challenge_\<Challenge value>` .

File Explorer Path: This PC > OSDisk (C:) > Users > amleung > AppData > Local > SiliconLabs > commander > SecurityStore > device\_00000000000000014b457ffe045a32 > challenge\_8b925526a33b1ad3a95075055246e044

Name	Date modified	Type	Size
 <code>unlock_payload_00000000000111110.bin</code>	6/10/2021 3:37 PM	BIN File	1 KB

Users can also use the `security getpath` command to get the path of the Security Store or a specified device.

```
commander security getpath
```

```
C:/Users/<username>/AppData/Local/SiliconLabs/commander/SecurityStore
DONE
```

```
commander security getpath --deviceserialno 00000000000000014b457ffe045a32
```

```
C:/Users/<username>/AppData/Local/SiliconLabs/commander/SecurityStore/device_00000000000000014b457ffe045a32
DONE
```

11. The Debug Unlock Token and the device are now delivered to the Debug 3rd Party.

Run the `security gencommand` command to create the Security Store to place the Debug Unlock Token file.

```
commander security gencommand --action debug-unlock --device EFR32MG21A010F1024 --serialno 440048205
```

```
Unsigned command file written to Security Store:
C:/Users/<username>/AppData/Local/SiliconLabs/commander/SecurityStore/device_00000000000000014b457ffe045a32/challenge_8b9255
DONE
```

Copy the Debug Unlock Token file ( `unlock_payload_000000000011110.bin` ) from Product Company to the Windows Security Store `challenge_<Challenge value>` folder located in `C:\Users\<PC user`

`name>\AppData\Local\SiliconLabs\commander\SecurityStore\device_<Serial number>\challenge_<Challenge value>` .

12. The device compares the Debug Unlock Token contents with its internal serial number, challenge value, and Public Command Key to determine the token's authenticity. If authentic, it will execute the [debug access command](#) to unlock the device; otherwise, it will ignore the command.

Run the `security unlock` command to unlock the device.

```
commander security unlock --unlock-param 1111 --device EFR32MG21A010F1024 --serialno 440048205
```

```
Unlocking with unlock payload:
C:/Users/<username>/AppData/Local/SiliconLabs/commander/SecurityStore/device_00000000000000014b457ffe045a32/challenge_8b9255
debug successfully unlocked
DONE
```

#### Notes:

- If the security store has multiple tokens for the selected device, use `--unlock-param` option to specify which unlock token is chosen to unlock the device.
- Simplicity Commander will only use the token with value 1111 (error if not available) from the security store to unlock the device if the security unlock command does not include the `--unlock-param` option.

13. For Simplicity Commander  $\geq$  v1.13.3, run the `security status --trustzone` command to check the full debug lock status of the device.

```
commander security status --trustzone --device EFR32MG21A010F1024 --serialno 440048205
```

```
SE Firmware version : 1.2.14
Serial number      : 0000000000000000014b457ffe045a32
Debug lock        : Enabled
Device erase      : Disabled
Secure debug unlock : Enabled
Debug lock state: Unlocked
Non-secure, invasive debug lock (DBGLOCK) : Unlocked
Non-secure, non-invasive debug lock (NIDLOCK) : Unlocked
Secure, invasive debug lock (SPIDLOCK) : Locked
Secure, non-invasive debug lock (SPNIDLOCK): Locked
Non-secure, invasive debug lock state (DBGLOCK) : Unlocked
Non-secure, non-invasive debug lock state (NIDLOCK) : Unlocked
Secure, invasive debug lock state (SPIDLOCK) : Unlocked
Secure, non-invasive debug lock state (SPNIDLOCK): Unlocked
Tamper status     : OK
Secure boot       : Disabled
Boot status       : 0x20 - OK
DONE
```

14. The Debug 3rd Party can now use this same Debug Unlock Token to unlock the device (step 12), over and over again after each power-on or pin reset, until they have finished debugging the device.
15. Once the Debug 3rd Party has finished debugging, they will send the device back to the Product Company.

will effectively invalidate any Debug Unlock Token that has been previously given to any third party.

Run the `security rollchallenge` command and reset the device to invalidate the current Debug Unlock Token. The challenge cannot be rolled before it has been used at least once — that is, by running the `security unlock` or `security disabletamper` command.

```
commander security rollchallenge --device EFR32MG21A010F1024 --serialno 440048205
```

```
Challenge was rolled successfully.
DONE
```

The unlock token is invalidated after rolling the challenge because any previously issued Debug Unlock Token now contains a different challenge value ( Challenge 1 ) than the challenge value currently in the device ( Challenge 2 ).

The validation process of any previously issued Debug Unlock Token will always fail until a new Debug Unlock Token is issued with a current matching challenge value ( Challenge 2 ).

**Note:** Direct Customer can directly use the Private Command Key on the connected chip to generate the Debug Unlock Token in Security Store. But it has a high risk (cannot use HSM) to leak the Private Command Key to a 3rd party when using this approach.

```
commander security unlock --command-key command_key.pem--unlock-param 1111 --device EFR32MG21A010F1024
--serialno 440048205
```

Authorization file written to Security Store:

C:/Users/<username>/AppData/Local/SiliconLabs/commander/SecurityStore/device\_00000000000000014b457ffe045a32/certificate\_authori

Generating ECC P256 key pair...

Cert public key stored at:

C:/Users/<username>/AppData/Local/SiliconLabs/commander/SecurityStore/device\_00000000000000014b457ffe045a32/cert\_pubkey.pem

Cert private key stored at:

C:/Users/<username>/AppData/Local/SiliconLabs/commander/SecurityStore/device\_00000000000000014b457ffe045a32/cert\_key.pem

Command public key stored at:

C:/Users/<username>/AppData/Local/SiliconLabs/commander/SecurityStore/device\_00000000000000014b457ffe045a32/command\_pubkey

Command private key stored at:

C:/Users/<username>/AppData/Local/SiliconLabs/commander/SecurityStore/device\_00000000000000014b457ffe045a32/command\_key.per

Certificate was signed with key:

C:/Users/<username>/AppData/Local/SiliconLabs/commander/SecurityStore/device\_00000000000000014b457ffe045a32/command\_key.per

Certificate written to Security Store:

C:/Users/<username>/AppData/Local/SiliconLabs/commander/SecurityStore/device\_00000000000000014b457ffe045a32/access\_certificate

Created unsigned unlock command

Signed unlock command using

C:/Users/<username>/AppData/Local/SiliconLabs/commander/SecurityStore/device\_00000000000000014b457ffe045a32/cert\_key.pem

Secure debug successfully unlocked

Command unlock payload was stored in Security Store

DONE

## Simplicity Studio

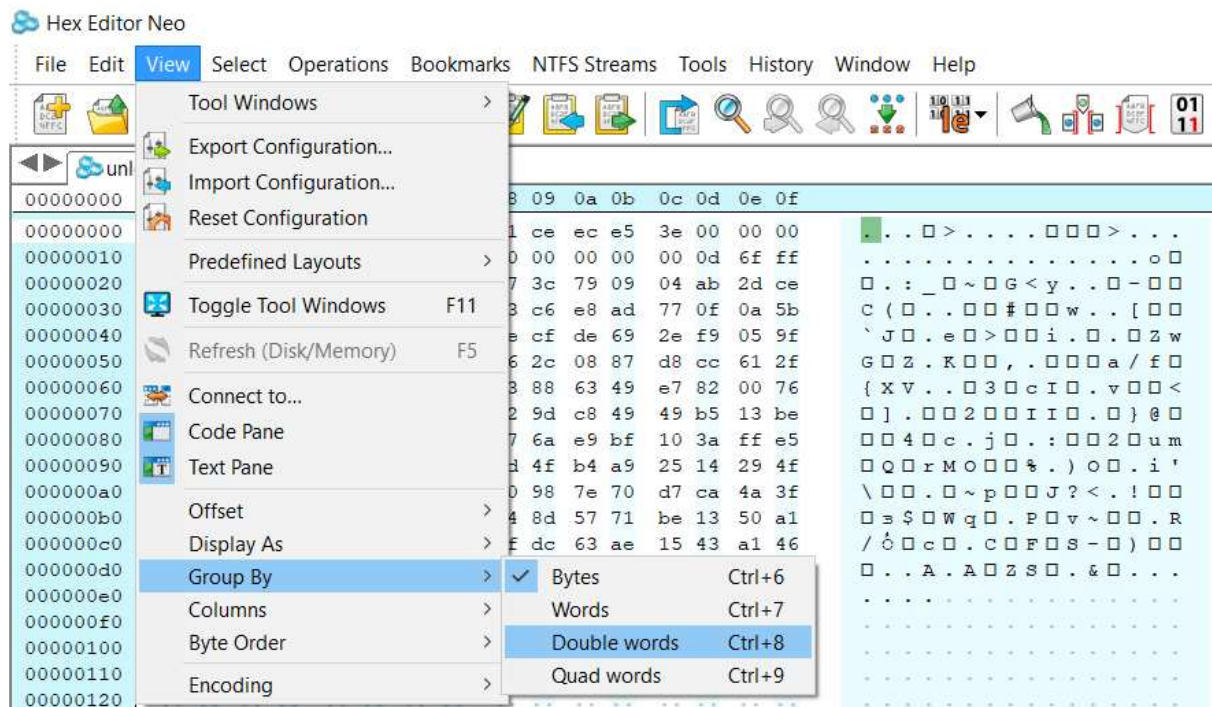
Use the Debug Unlock Token file ( `unlock_payload_000000000111110.bin` ) generated in [Using Simplicity Commander](#) steps 8 or 9 to unlock the device with Simplicity Studio.

1. Open the `unlock_payload_000000000111110.bin` file with the [Hex Editor Neo](#).



unlock_payload_000000000111110.bin																
00000000	00	01	02	03	04	05	06	07	08	09	0a	0b	0c	0d	0e	0f
00000000	01	00	01	fd	3e	00	00	00	01	ce	ec	e5	3e	00	00	00
00000010	00	00	00	00	00	00	00	00	00	00	00	00	00	0d	6f	ff
00000020	fe	0a	3a	5f	b2	7e	f0	b4	47	3c	79	09	04	ab	2d	ce
00000030	cb	43	28	e5	10	07	ba	85	23	c6	e8	ad	77	0f	0a	5b
00000040	a3	e8	60	4a	c5	12	65	ae	3e	cf	de	69	2e	f9	05	9f
00000050	5a	77	47	cd	5a	00	4b	90	e6	2c	08	87	d8	cc	61	2f
00000060	66	87	7b	58	56	05	13	ed	33	88	63	49	e7	82	00	76
00000070	e6	eb	3c	92	5d	11	e6	c2	32	9d	c8	49	49	b5	13	be
00000080	7d	40	f9	e5	df	34	ff	63	07	6a	e9	bf	10	3a	ff	e5
00000090	32	b4	75	6d	9c	51	d9	72	4d	4f	b4	a9	25	14	29	4f
000000a0	b2	00	69	27	5c	c1	93	13	e0	98	7e	70	d7	ca	4a	3f
000000b0	3c	13	21	87	f5	87	d0	b7	24	8d	57	71	be	13	50	a1
000000c0	76	7e	f7	a6	12	52	2f	d9	8f	dc	63	ae	15	43	a1	46
000000d0	fc	53	2d	db	29	90	c7	d0	03	1c	41	1f	41	ee	5a	53
000000e0	f4	10	26	f1	..	..	..	..	..	..	..	..	..	..	..	..

- Click **View** to open the context menu, and then select **Group By** → **Double words** to convert the token into a little-endian format.



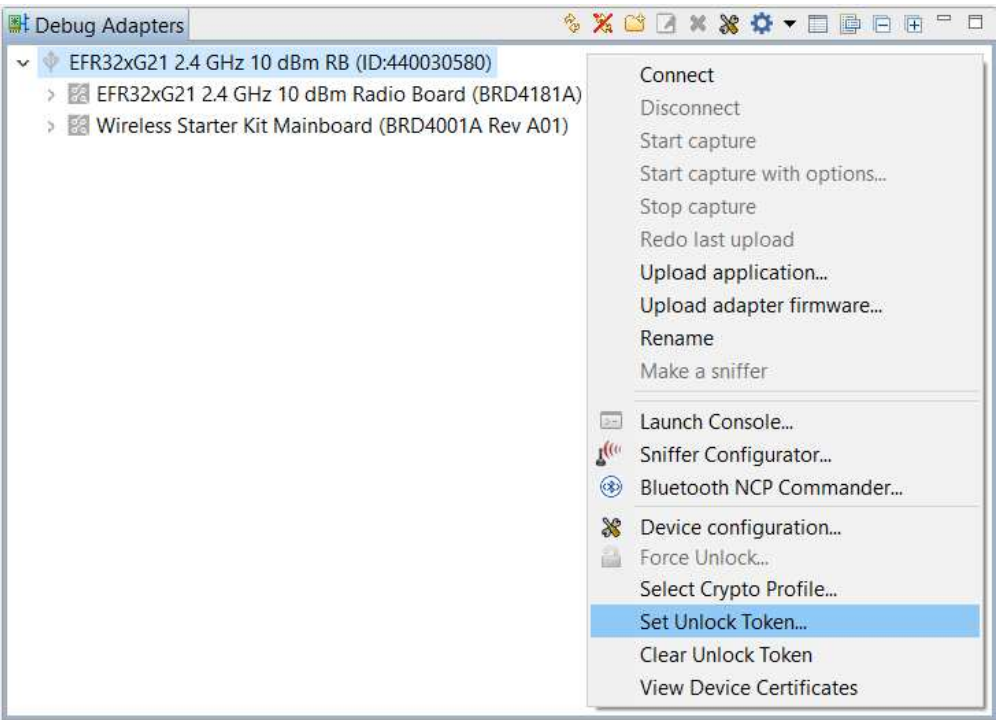
- Select all (Ctrl+A) and copy (Ctrl+C) the Debug Unlock Token to a text editor.

unlock_payload_000000000111110.bin																
00000000	00	01	02	03	04	05	06	07	08	09	0a	0b	0c	0d	0e	0f
00000000	fd010001	0000003e	e5ecce01	0000003e	00000000	00000000	ff6f0d00	5f3a0afe	b4f07eb2	09793c47	ce2dab04	e52843cb	85ba0710	ade8c623	5b0a0f77	
00000010	4a60e8a3	ae6512c5	69decf3e	9f05f92e	cd47775a	904b005a	87082ce6	2f61ccd8	587b8766	ed130556	49638833	760082e7	923cebe6	c2e6115d	49c89d32	be13b549
00000020	e5f9407d	63ff34df	bfe96a07	e5ff3a10	6d75b432	72d9519c	a9b44f4d	4f291425	276900b2	1393c15c	707e98e0	3f4acac7	8721133c	b7d087f5	71578d24	a15013be
00000030	a6f777e6	d92f5212	ae63dc8f	46a14315	db2d53fc	d0c79029	1f411c03	535aee41	f12610f4	.....	.....	.....	.....	.....	.....	.....

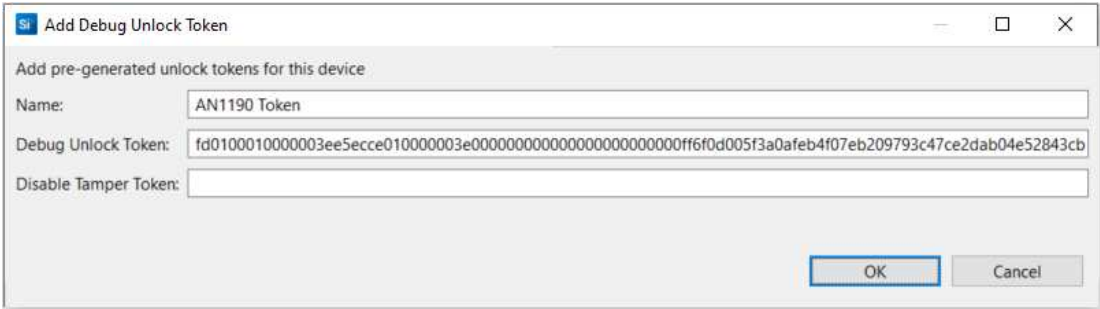
- Use the text editor to remove all the spaces from the token.

```
fd0100010000003ee5ecce010000003e00000000000000000000ff6f0d00...
```

- Right-click the selected debug adapter **RB (ID:J-Link serial number)** to display the context menu.



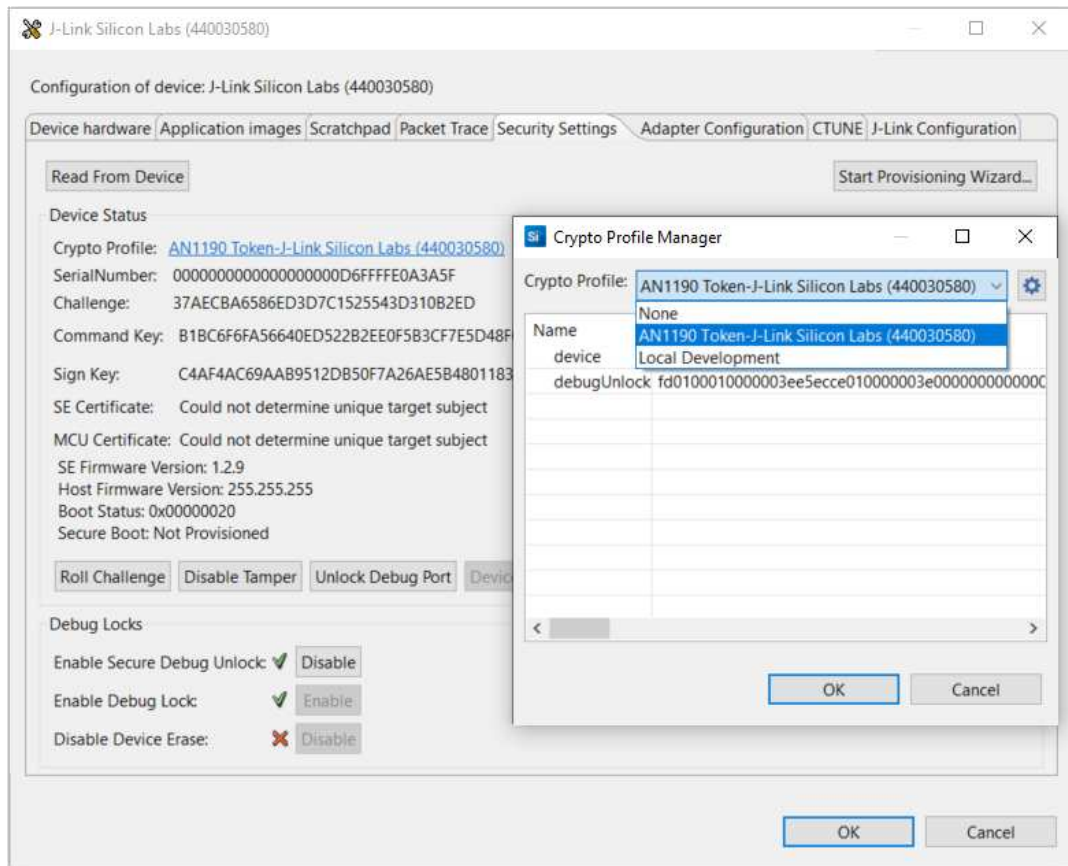
6. Click **Set Unlock Token** to open the **Add Debug Unlock Token** dialog box. Enter the name (e.g., AN1190 Token ) for this Debug Unlock Token, and copy the content in step 4 to the **Debug Unlock Token:** box. Click **[OK]** to confirm and exit.



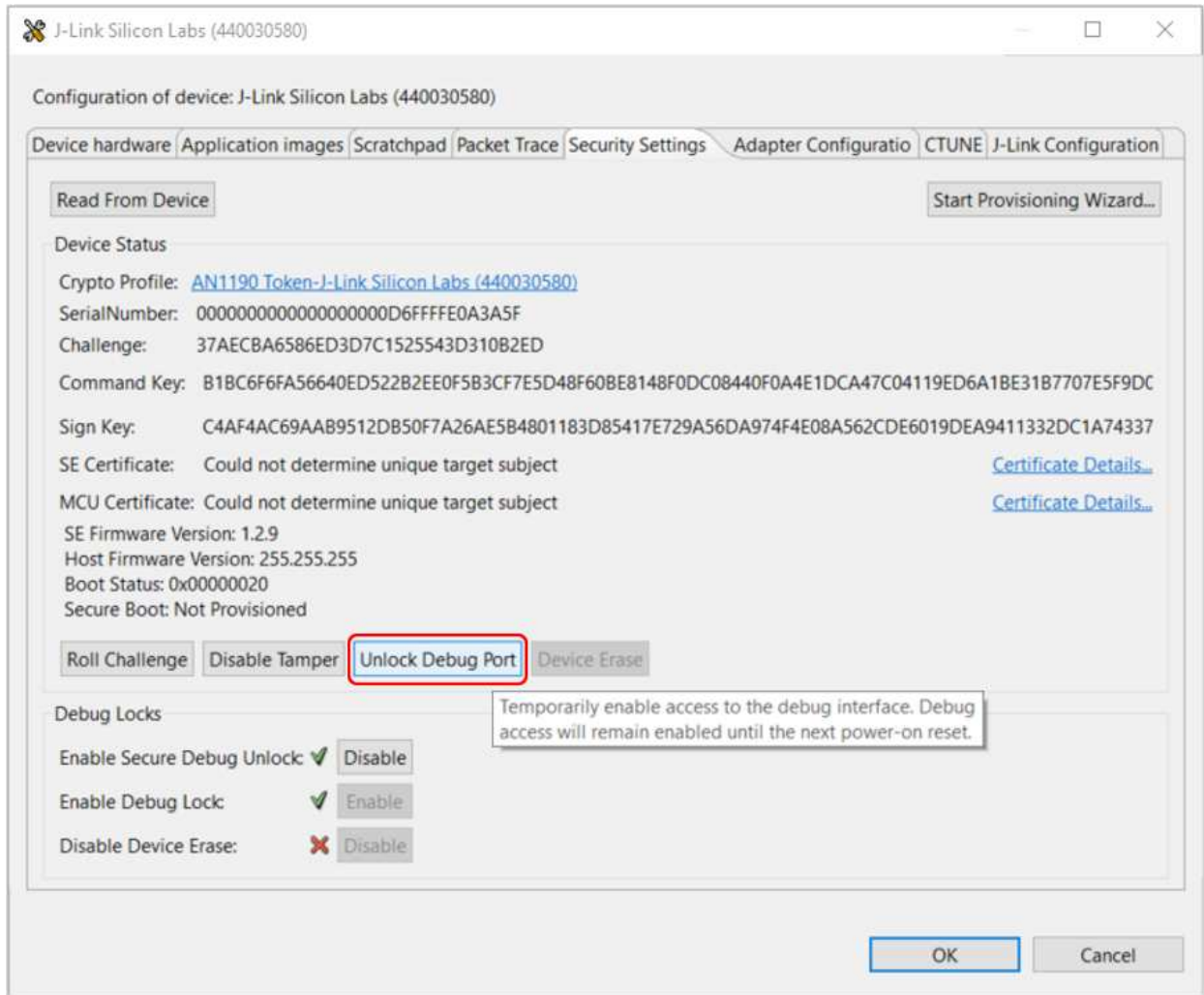
**Note:** The Simplicity Studio can only keep one Debug Unlock Token on each WSTK.

- 7. Open **Security Settings** of the selected device as described in [Using Simplicity Studio](#).
- 8. The token added in step 6 should display on the **Crypto Profile:** field. If not, click the link next to **Crypto Profile:** to select the token from the **Crypto Profile Manager** drop-down list. The Simplicity Studio will automatically add the WSTK J-Link serial number ( -J-Link Silicon Labs (serial number) ) to the token's name.

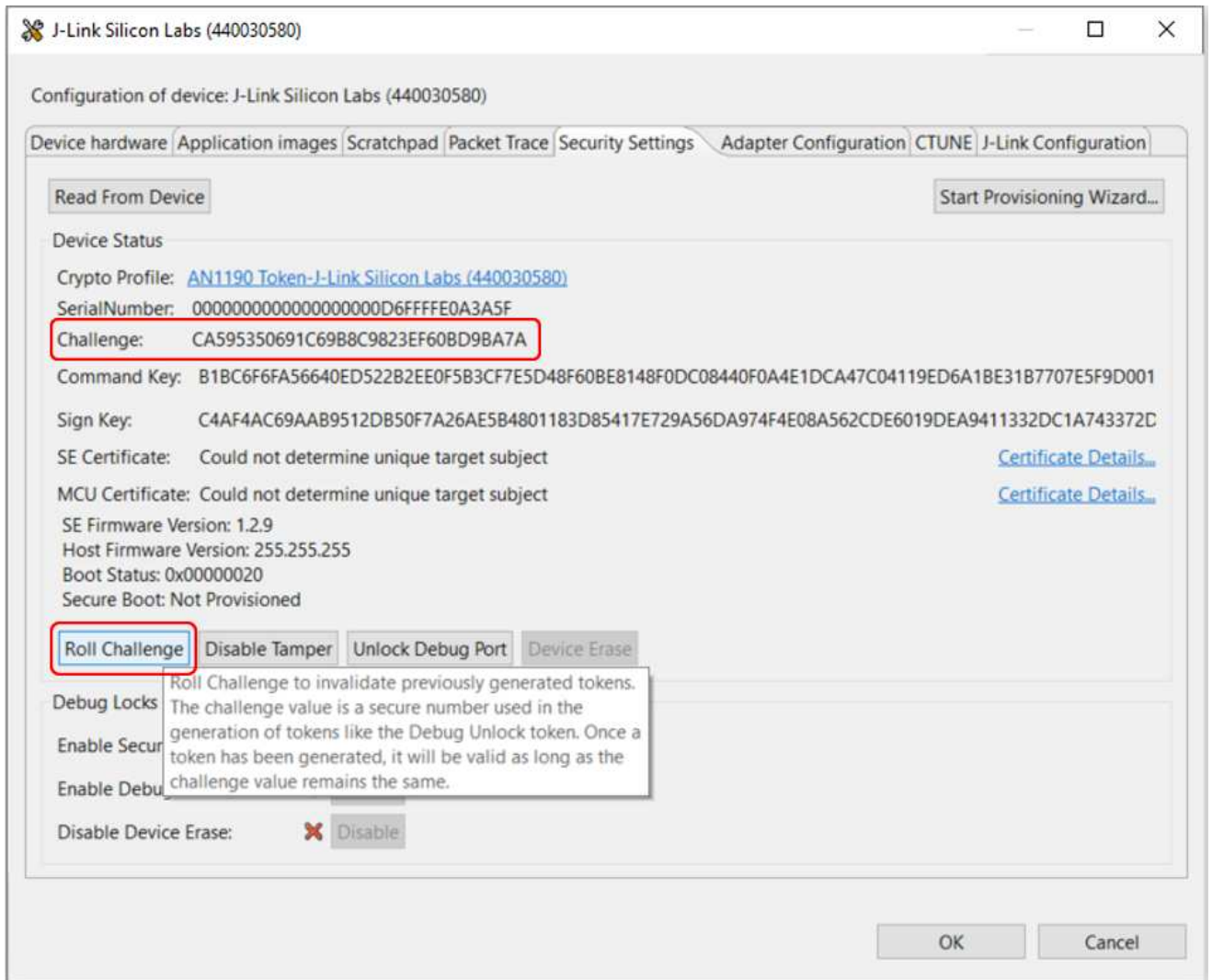




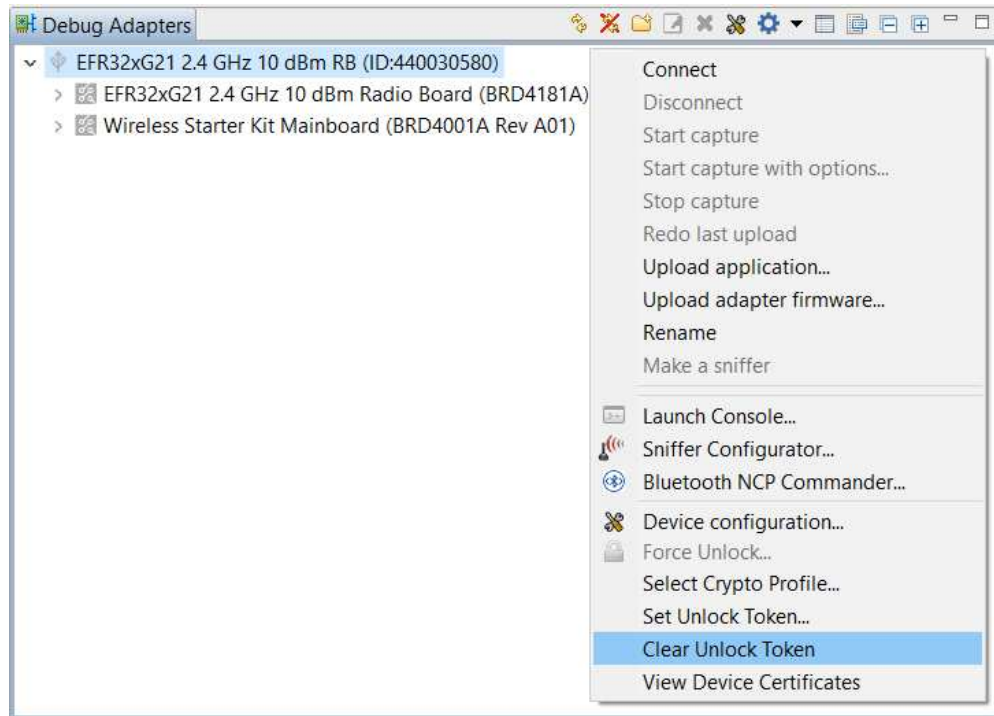
9. Click **[Unlock Debug Port]** to use the token in **Crypto Profile**: to unlock the device (invalid token will display an error message). The device stays in the unlock state until the next power-on or pin reset. Click **[OK]** to exit.



10. The Simplicity IDE will automatically use the selected Debug Unlock Token in **Crypto Profile** for debugging and flashing. For other IDE, the device should unlock again (step 9) after power-on or pin reset. After finished debugging, open the **Security Settings** of the selected device as described in [Using Simplicity Studio](#).
11. Click **[Roll Challenge]** to generate a new challenge value to invalidate the Debug Unlock Token added in step 6. Click **[OK]** to exit.



12. Right-click the selected debug adapter **RB Board** (ID:J-Link serial number) to display the context menu.

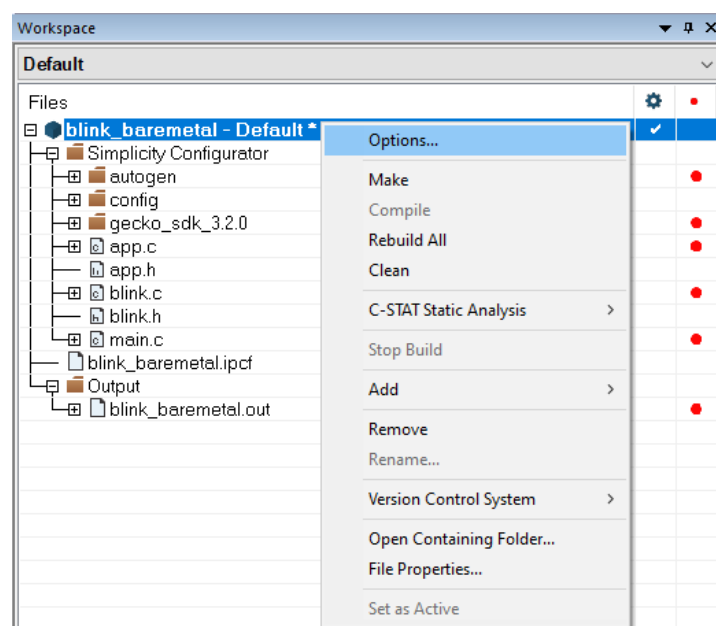


13. Click **[Clear Unlock Token]** to delete the WSTK Debug Unlock Token from Simplicity Studio.

## IAR

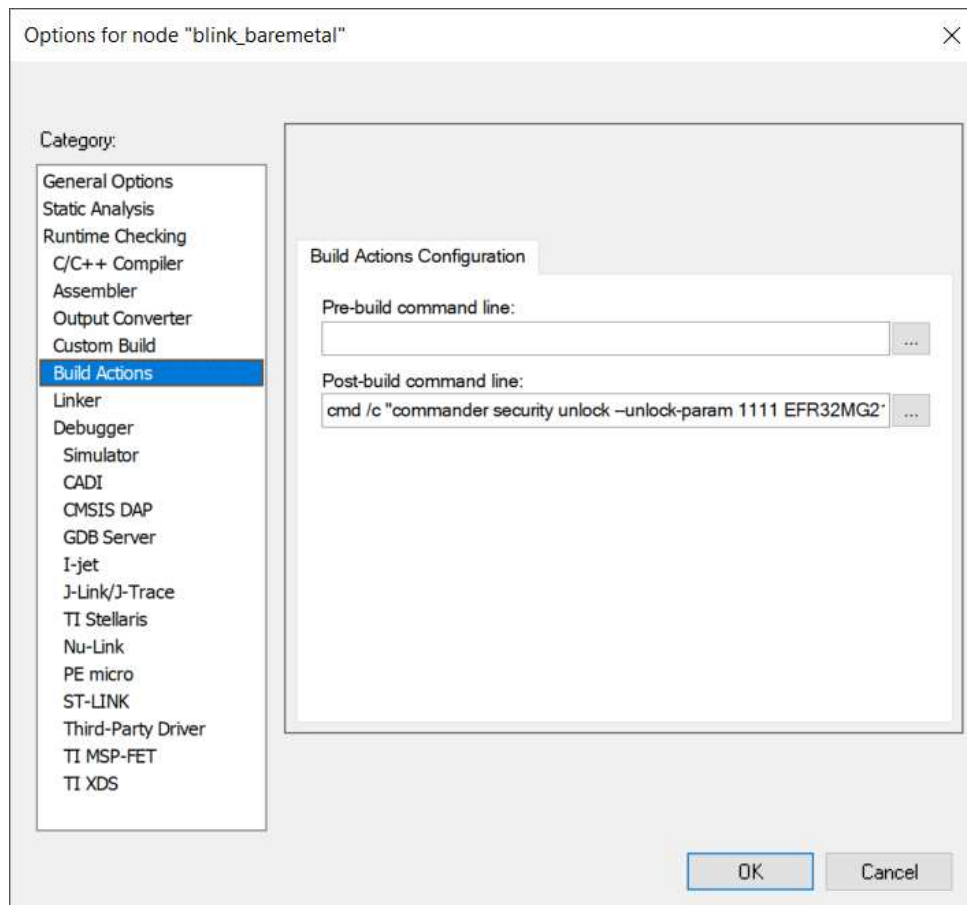
Use the Debug Unlock Token file ( `unlock_payload_0000000000111110.bin` ) in the Security Store ( [Simplicity Commander](#) step 11) to unlock the device with IAR (Windows).

1. The Windows environment variable `PATH` should include the folder ( `C:\SiliconLabs\SimplicityStudio\v5\developer\adapter_packs\commander` ) that locates the `commander.exe` of Simplicity Commander.
2. Right-click the project in the workspace, and then click **Options....**

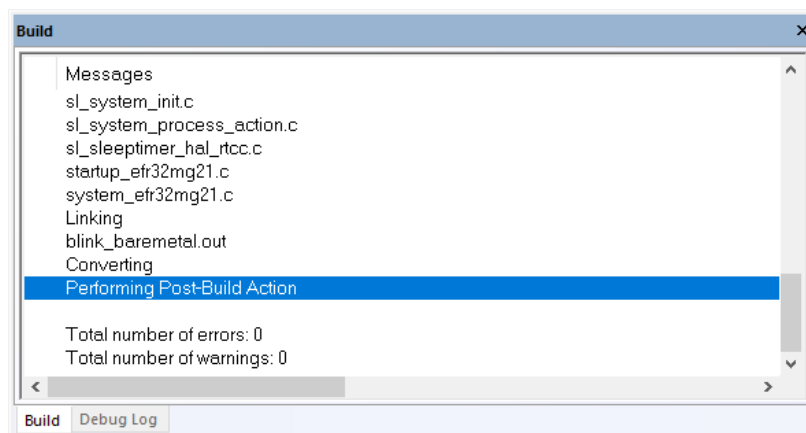


3. Click **Build Actions** to open the **Build Actions Configuration** dialog box. Enter the phrase below to the **Post-build command line**: box. Click [OK] to exit.

```
cmd /c "commander security unlock --unlock-param 1111 EFR32MG21A010F1024 --serialno 440048205 > $PROJ_DIR$\log.txt 2>&1"
```

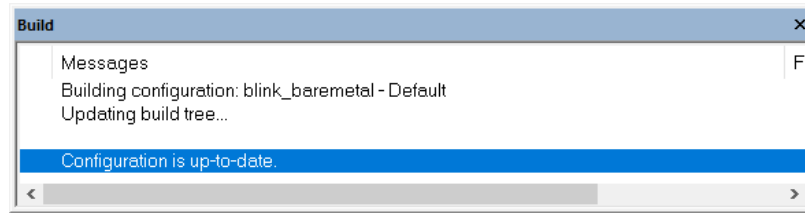


4. After building the project, the `security unlock` in the **Post-build command** unlocks the device using the Debug Unlock Token in Security Store. The device stays in the unlock state until the next power-on or pin reset.

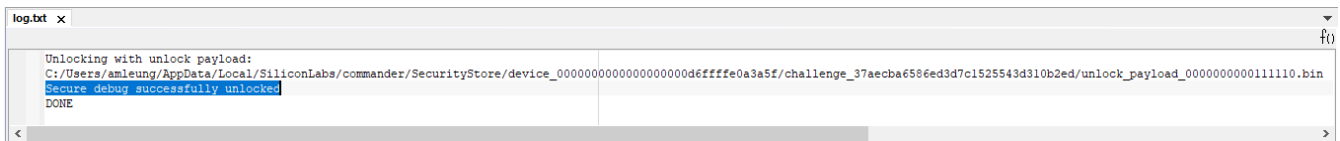


**Note:** If the project is already up-to-date, it will not invoke the **Post-build command** to unlock the device. Use a dummy edit (add space or newline) on one of the source files in the project to trigger the build

action.



5. The `> $PROJ_DIR$log.txt 2>&1` redirects the `security unlock` command output to the `log.txt` file in the IAR project folder.



## Permanent Debug Lock

### Simplicity Commander

1. Run the `security status` command to get the selected device configuration.

```
commander security status --device EFR32MG21A010F1024 --serialno 440048205
```

```
SE Firmware version : 1.2.14
Serial number      : 000000000000000014b457fffe045a32
Debug lock        : Disabled
Device erase      : Enabled
Secure debug unlock : Disabled
Tamper status     : OK
Secure boot       : Disabled
Boot status      : 0x20 - OK
DONE
```

2. Run the `security lock` command to lock the selected device.

```
commander security lock --device EFR32MG21A010F1024 --serialno 440048205
```

```
WARNING: Secure debug unlock is disabled. Only way to regain debug access is to run a device erase.
Device is now locked.
DONE
```

3. Run the `security disabledeviceerase` command to disable device erase.

```
commander security disabledeviceerase --device EFR32MG21A010F1024 --serialno 440048205
```

```
=====
THIS IS A ONE-TIME command which Permanently disables device erase.
If secure debug lock has not been set, there is no way to regain debug access to this device.
Type 'continue' and hit enter to proceed or Ctrl-C to abort:
=====
continue
Disabled device erase successfully
DONE
```

**Note:** This is an **IRREVERSIBLE** action, and should be the last step in production.

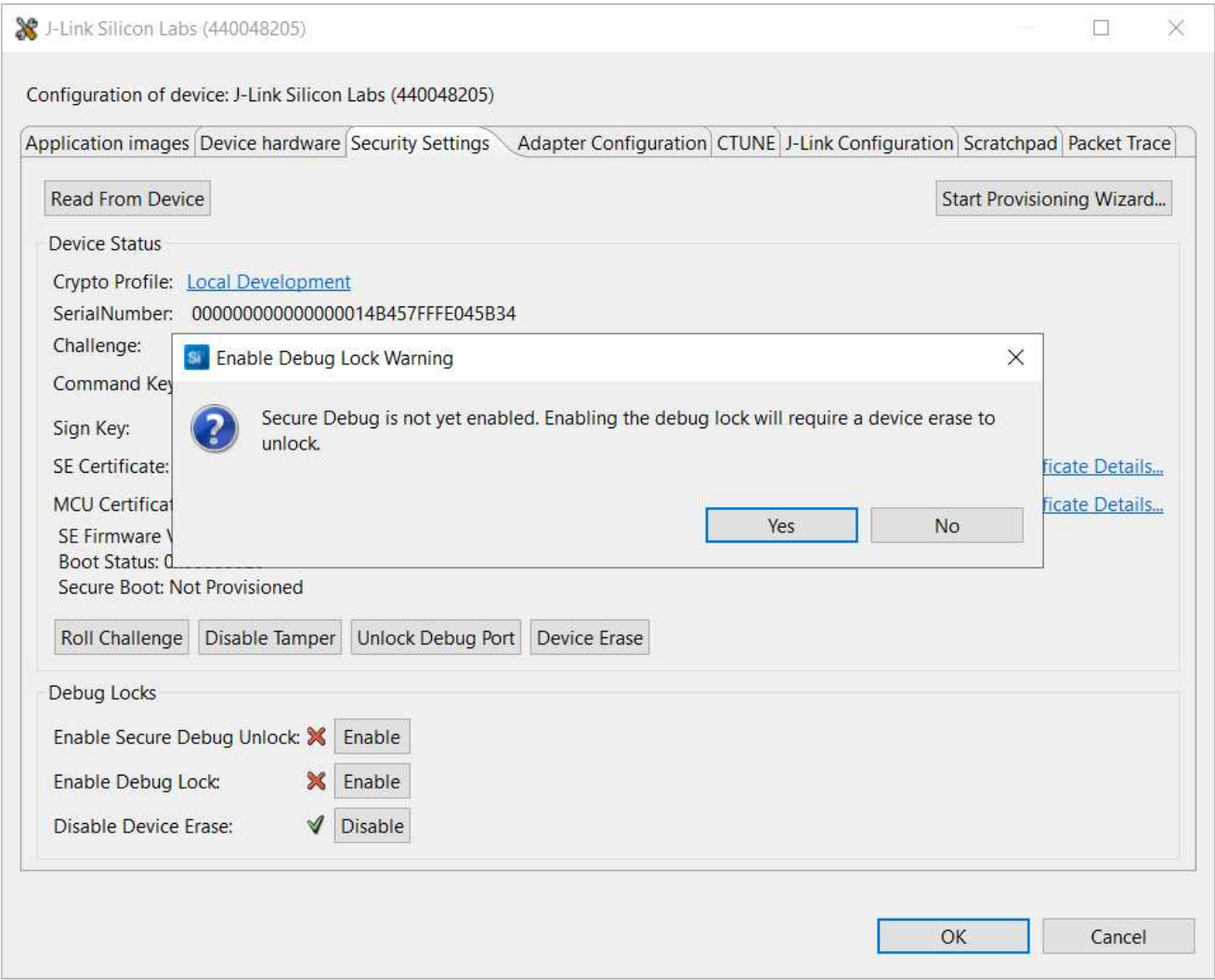
4. Run the `security status` command again to check the device configuration.

```
commander security status --device EFR32MG21A010F1024 --serialno 440048205
```

```
SE Firmware version : 1.2.14
Serial number      : 000000000000000014b457ffe045a32
Debug lock        : Enabled
Device erase       : Disabled
Secure debug unlock : Disabled
Tamper status      : OK
Secure boot        : Disabled
Boot status        : 0x20 - OK
DONE
```

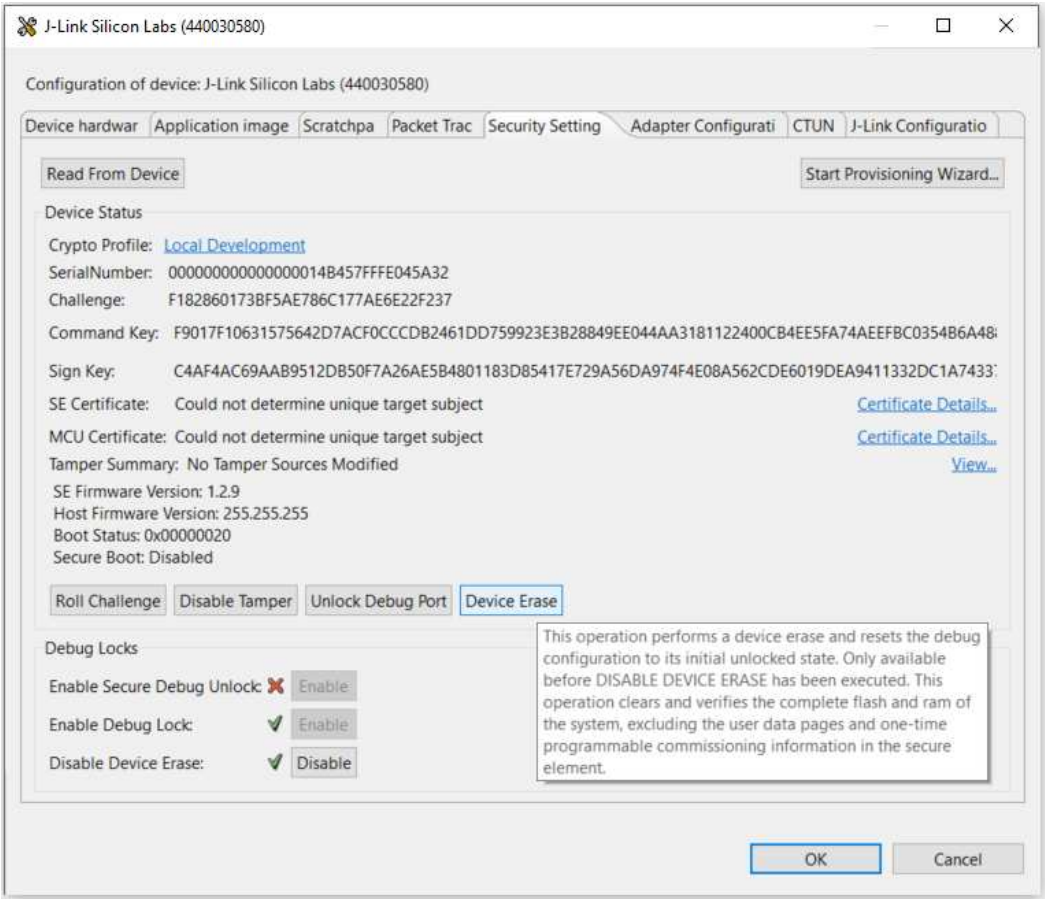
Simplicity Studio

- 1. Open **Security Settings** of the selected device as described in [Using Simplicity Studio](#).
- 2. Click **[Enable]** next to **Enable Debug Lock**: to lock the device. The following **Enable Debug Lock Warning** is displayed. Click **[Yes]** to confirm. This configures standard debug lock.





3. Click **[Disable]** next to **Disable Device Erase**: to disable the device erase. The following **Disable Device Erase Warning** is displayed. Click **[Yes]** to confirm. This configures a permanent debug lock.



**Note:** This is an **IRREVERSIBLE** action, and should be the last step in production.



Precautions

# Precautions

## Device Erase for Secure Debug

Disable the [Device Erase](#) is mandatory for secure debug as described in the following table.

Secure Debug	Device Erase	Debug Lock	State	Description
Enabled	Enabled	Enabled	Insecure debug lock (1)	The device will return to the default debug lock properties after applying the standard debug unlock. (2)
Enabled	Disabled (3)	Enabled	Secure debug lock	The device cannot be unlocked using the Erase Device command. The device will change to the permanent debug lock state if disabling the Secure Debug property. (4)

Notes:

- 1. This state is only for secure debug testing.
- 2. See [Standard Debug Unlock](#).
- 3. This is an **IRREVERSIBLE** action and should be disabled **AFTER** the secure debug is enabled.
- 4. See [Permanent Debug Lock](#).

```
commander security lockconfig --secure-debug-unlock disable --device EFR32MG21A010F1024
--serialno 440048205
```

```
=====
WARNING: Device erase is disabled and secure debug access is locked.
If disabling secured debug access, there is no way to regain debug access to this device if continuing with this command.
Type 'continue' and hit enter to proceed or Ctrl-C to abort:
=====
continue
Secure debug unlock was disabled
DONE
```

## Secure Boot and Debug Lock

The following table describes the different debug lock scenarios on the secure boot-enabled device.

Secure Debug	Device Erase	Debug Lock	State	Recover from Secure Boot Failure
Disabled	Enabled	Disabled	Standard debug unlock	Flash a correctly signed image.
Disabled	Enabled	Enabled	Standard debug lock	Flash a correctly signed image after standard debug unlocking the device.
Disabled	Disabled	Enabled	Permanent debug lock	There is no way to recover the device. Make sure the programmed image is correctly signed before locking the device.
Enabled	Disabled	Enabled	Secure debug lock	Flash a correctly signed image after secure debug unlocking the device.

**Note:** See section *Recover Devices when Secure Boot Fails* in [Series 2 Secure Boot with RTSL](#) to flash a correctly signed image on different debug lock scenarios.

Failure Analysis

# Failure Analysis

The following table describes the different scenarios when returning a Series 2 device to Silicon Labs for failure analysis.

State	Secure Boot Disabled	Secure Boot Enabled (2)
Standard debug unlock	Device erase is not necessary for failure analysis.	Device erase is not necessary, but a correctly signed image is required to perform failure analysis.
Standard debug lock	Device erase is required to perform failure analysis.	Require device erase and correctly signed image to perform failure analysis.
Permanent debug lock	Cannot perform failure analysis.	Cannot perform failure analysis.
Secure debug lock (1)	Require debug unlock token to perform failure analysis.	Require debug unlock token and correctly signed image to perform failure analysis.

Notes:

- 1. Follow the procedures in [Simplicity Commander](#) to generate a valid debug unlock token for each device returned to Silicon Labs for failure analysis.
- 2. Secure boot enabled devices, especially with secure boot failure, may limit Silicon Labs' ability to determine the root cause of failure.

## Series 2 TrustZone

# Series 2 TrustZone

NOTE: This section replaces *AN1374: TrustZone*. Further updates to this user guide will be provided here.

ARMv8-M TrustZone is a technology that provides a foundation for improved system security in embedded applications. It allows the ARMv8-M to be aware of the security states of the system. Series 2 devices use the Cortex-M33 core to implement the ARMv8-M TrustZone security extension, which provides the ability to restrict access to peripherals and memory regions based on the processor security attribute. TrustZone works with the MPU, which controls privileged/unprivileged execution of code to provide a complete security solution.

ARMv8-M TrustZone is an extensive topic. The references below are publicly available on the [ARM Developer Documentation](#) website.

- [ARMv8-M Architecture Reference Manual](#)
  - [ARMv8-M Architecture Technical Overview](#)
  - [ARM Cortex-M33 Processor Technical Reference Manual](#)
  - [System Design with ARMv8-M](#)
  - [TrustZone technology for ARMv8-M Architecture](#)
  - [ARM Cortex-M33 Devices Generic User Guide](#)
  - [Secure software guidelines for ARMv8-M](#)
  - [Software Development in ARMv8-M Architecture](#)
- Reading guides:
- Beginner
  - Minimal experience with TrustZone, starting with [TrustZone Basics](#)
  - Intermediate - Have a basic understanding of the TrustZone technology, starting with [Bus Level Security](#)
  - Advanced - Developed experience on TrustZone, starting with [TrustZone Implementation](#)-Demo - Starting with [TrustZone Platform Examples](#)

## Key Points

- TrustZone Basics
- Bus Level Security (BLS)
- Secure and Privileged Programming Model
- TrustZone Implementation
- Upgrade Existing Application to TrustZone
- TrustZone Platform Examples

Series 2 Device Security Features

# Series 2 Device Security Features

Protecting IoT devices against security threats is central to a quality product. Silicon Labs offers several security options to help developers build secure devices, secure application software, and secure communication paths to manage those devices. Silicon Labs’ security offerings were significantly enhanced by the introduction of the Series 2 products that included a Secure Engine. The Secure Engine is a tamper-resistant component used to securely store sensitive data and keys, and to execute cryptographic functions and secure services.

On Series 2 devices, the security features are implemented by the Secure Engine and CRYPTOACC (if available). The Secure Engine may be hardware-based or virtual (software-based). Throughout this document, the following abbreviations are used:

- HSE - Hardware Secure Engine
- VSE - Virtual Secure Engine
- SE - Secure Engine (either HSE or VSE)

Additional security features are provided by Secure Vault. Three levels of Secure Vault feature support are available, depending on the part and SE implementation, as reflected in the following table:

Level (1)	SE Support	Part
Secure Vault High (SVH)	HSE only (HSE-SVH)	Refer to <a href="#">IoT Endpoint Security Fundamentals</a> for details on supporting devices.
Secure Vault Mid (SVM)	HSE (HSE-SVM)	"
Secure Vault Mid (SVM)	VSE (VSE-SVM)	"
Secure Vault Base (SVB)	N/A	"

**Note:** 1. The features of different Secure Vault levels can be found in <https://www.silabs.com/security>.

Secure Vault Mid consists of two core security functions:

- Secure Boot: Process where the initial boot phase is executed from an immutable memory (such as ROM) and where code is authenticated before being authorized for execution.
- Secure Debug Access Control: The ability to lock access to the debug ports for operational security, and to securely unlock them when access is required by an authorized entity.

Secure Vault High offers additional security options:

- Secure Key Storage: Protects cryptographic keys by "wrapping" or encrypting the keys using a root key known only to the HSE-SVH.
- Anti-Tamper protection: A configurable module to protect the device against tamper attacks.
- Device authentication: Functionality that uses a secure device identity certificate along with digital signatures to verify the source or target of device communications.

Series 2 devices require a specific [SE firmware version](#) to support the TrustZone implementation. Refer to [AN1222: Production Programming of Series 2 Devices](#) to learn how to upgrade the SE firmware and [IoT Endpoint Security Fundamentals](#) for the latest SE Firmware shipped with Series 2 devices and modules.

Series 2 devices use Cortex-M33 core to implement the ARMv8-M Mainline TrustZone security extension and refer to TrustZone as [Bus Level Security](#). The following table lists the configuration of TrustZone related components in the Series 2 Cortex-M33 core.

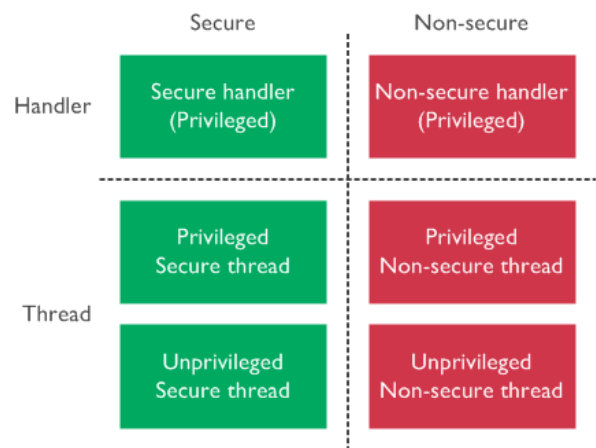
Component	Series 2 Configuration	Description
Security Extension (TrustZone)	Enabled	The security extension cannot be disabled, and the entire memory after RESET is Secure by default.
Memory Protection Unit (MPU)	16 regions (maximum)	The MPU regions for both Secure and Non-secure MPUs.
Security Attribution Unit (SAU)	8 regions (maximum)	The SAU regions for Non-secure and Non-secure Callable.

TrustZone Basics

# TrustZone Basics

## Introduction

TrustZone for ARMv8-M adds extra states to the Cortex-M processor operations to ensure there is a Secure and Non-secure state. These security states are orthogonal to the existing Thread and Handler modes, thereby having both a Thread and Handler mode in both Secure and Non-secure states. The Thread mode can also be either Privileged or Unprivileged.



Copyright © 1995-2022 Arm Limited (or its affiliates). All rights reserved.

TrustZone for ARMv8-M is an optional architecture extension. By default, the system starts up in a Secure state if the processor implements the TrustZone security extension. The division of Secure and Non-secure worlds is memory-map based (security state depends on the address of the fetched instruction), and the transitions happen automatically. It is also possible to leave the Non-secure state unused and execute the whole application in the Secure state.

## Memory Security Attributes

TrustZone classifies memory into four security attributes as described in the following table.

Security Attribute	Processor State	Description
Non-secure (NS)	Non-secure	Non-secure and Secure software can access these memory regions.
Secure (S)	Secure	Secure software can access these memory regions. Non-secure software cannot gain access to the Secure memory.
Non-secure Callable (NSC)	Secure	Secure memory with an NSC attribute provides entry points for Secure APIs that can be called from a Non-secure space. It is a region of memory that contains the Secure Gateway (SG) veneers that allow Non-secure code to call secure functions that exist in Secure code. Non-secure software cannot read/write to an NSC memory but can branch into it if the branch target is an SG instruction.

Security Attribute	Processor State	Description
Exempted	Secure/Non-secure	Non-secure and Secure software can access these memory regions (exempted from security checking). Exempted regions are typically used by debugging components that do not pose any security risk (e.g., system ROM table) when accessed by the Non-secure software.

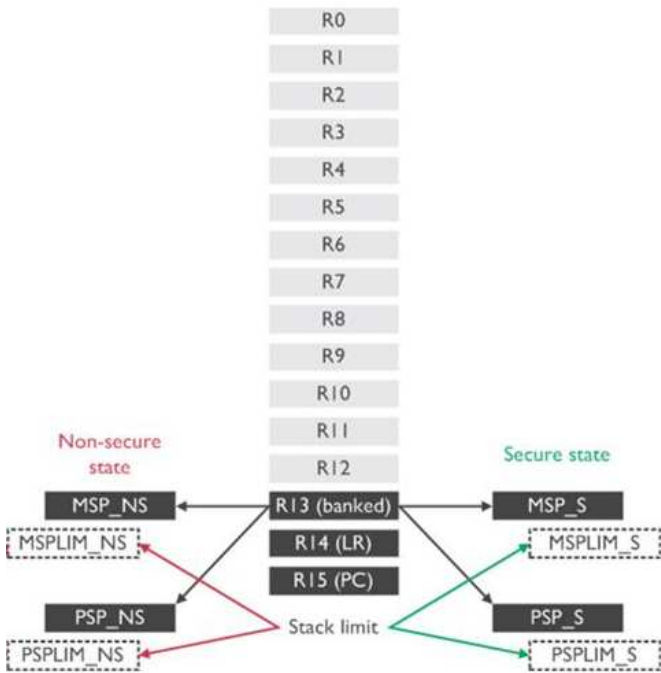
**Note:** The [Non-secure Callable](#) is also known as Secure Non-secure Callable (Secure NSC) to declare that this region resides in Secure memory.

## Banked Register

The concept of a banked register in ARMv8-M between Secure and Non-secure states means that there are two copies of the register, and the core automatically uses the copy that belongs to the current security state. When a register is banked, the `_S` and `_NS` suffixes are used in the ARMv8-M architecture to identify whether the resource is for the Secure state or Non-secure state.

## General-Purpose Registers

The Cortex-M processors have 16 general-purpose registers (R0 - R15) for data processing (R0 - R12) and control. The following figure shows the general-purpose register view of the ARMv8-M system with TrustZone. Refer to the [ARM Cortex-M33 Devices Generic User Guide](#) for details about these registers.

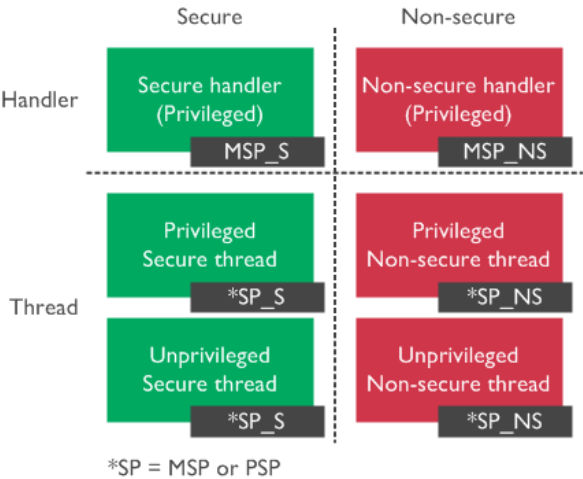


Copyright © 1995-2022 Arm Limited (or its affiliates). All rights reserved.

The Secure or Non-secure state can access the data processing registers R0 - R12 and special usage registers R13 - R15. The register R13 (banked SP) is the stack pointer alias, and the actual stack pointer ( `MSP_NS` , `PSP_NS` , `MSP_S` , `PSP_S` ) accessed depends on the state (Secure or Non-secure) and mode (Handler or Thread) as described in the following figure.

In addition, stack limit registers ( [special registers](#) ) enable hardware to detect stack overflow conditions. Two pairs of [stack limit registers](#) ( `MSPLIM_NS` and `PSPLIM_NS` , `MSPLIM_S` and `PSPLIM_S` ) are implemented, one per security state, to protect the Main Stack Pointer (MSP) and Process Stack Pointer (PSP).





Copyright © 1995-2022 Arm Limited (or its affiliates). All rights reserved.

In Thread mode, execution can be privileged or unprivileged. The stack pointer used can be the MSP or PSP, depending on the `SPSEL` bit in the `CONTROL` register. When in Handler mode, the processor is Privileged. The stack pointer is always MSP.

It is possible to directly [access](#) the stack pointers (MSP and PSP) and stack limit registers (MSPLIM and PSPLIM), providing that the processor is in a privileged state. If the processor is in a Secure privileged state, the software can also access the Non-secure stack pointers ( `MSP_NS` and `PSP_NS` ) through [Core Register Access Functions](#) in CMSIS-Core.

Special-Purpose Registers

Except for the general-purpose registers, there are several special-purpose registers for conditional flags, interrupt masking, control, and stack pointer limit. The following figure shows the special-purpose registers view of the ARMv8-M system with TrustZone. Refer to the [ARM Cortex-M33 Devices Generic User Guide](#) for details about these registers.

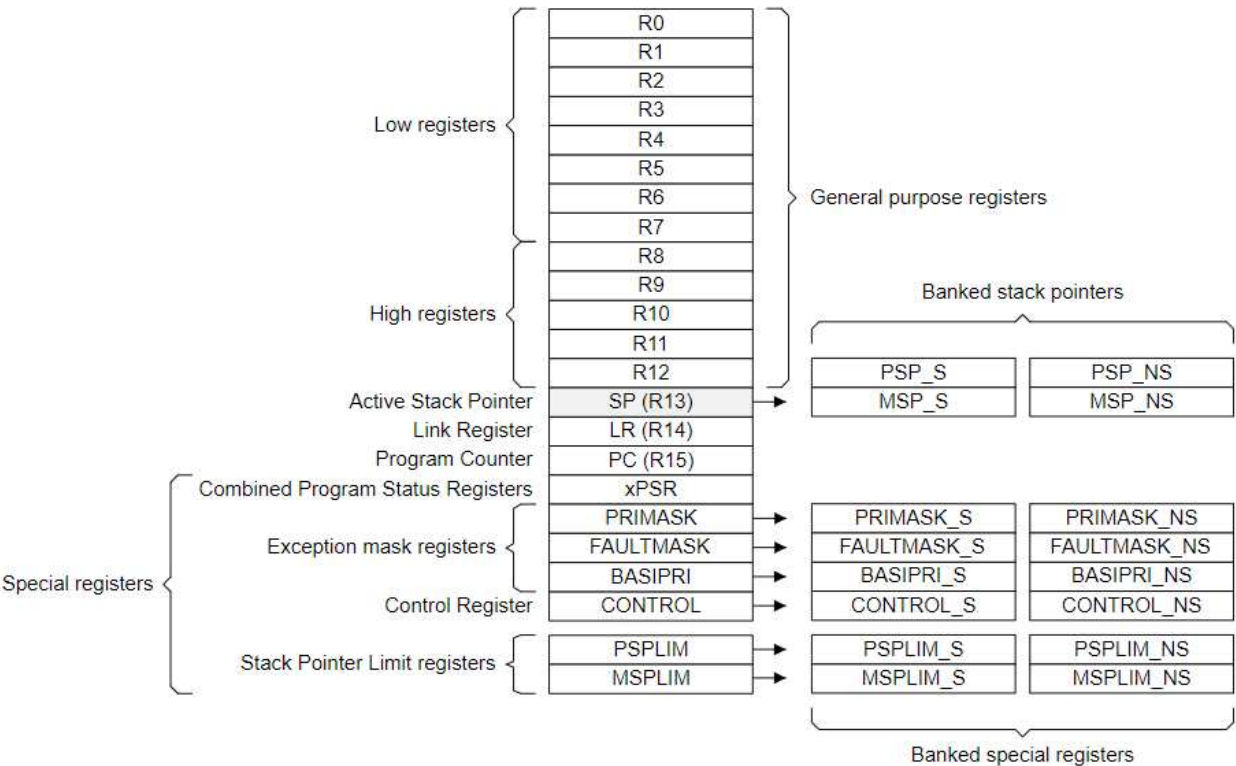


Image:<https://documentation-service.arm.com/>. Copyright © 1995-2022 Arm Limited (or its affiliates). All rights reserved.

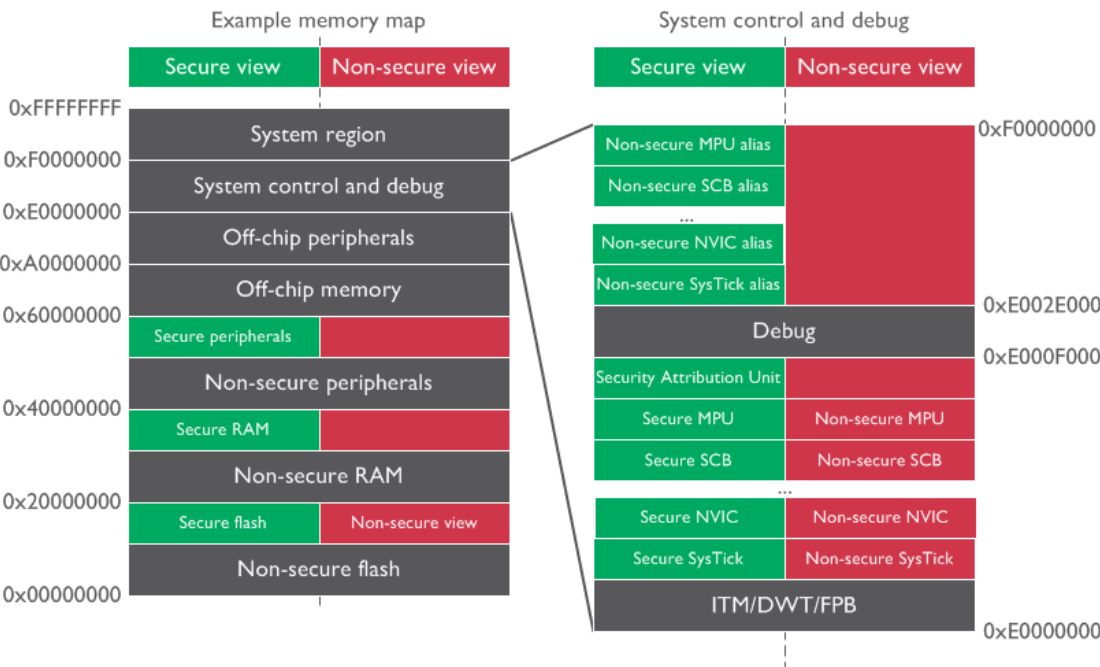
The Combined Program Status Register (xPSR) consists of the Application Program Status Register (APSR), Interrupt Program Status Register (IPSR), and Execution Program Status Register (EPSR).

Some of the special-purpose registers are banked between Secure and Non-secure states. Special-purpose registers are not memory-mapped and can be [accessed](#) using Core Register Access Functions in CMSIS-Core (except for EPSR in xPSR).

Secure privileged software can also access the Non-secure interrupt masking registers ( `PRIMASK_NS` , `FAULTMASK_NS` , and `BASEPRI_NS` ), `CONTROL` register ( `CONTROL_NS` ), and stack limit registers ( `MSPLIM_NS` and `PSPLIM_NS` ) through [Core Register Access Functions](#) in CMSIS-Core.

### System Private Peripheral Bus (PPB)

The banking of registers is usually used to separate the Secure and Non-secure information of the system components inside the processor. The following figure shows the System Private Peripheral Bus (PPB) registers view of the ARMv8-M system with TrustZone. Refer to the [ARM Cortex-M33 Devices Generic User Guide](#) for details about the System PPB registers.



System components for debugging and trace operations ( `0xE0000000` to `0xE0002FFF` ):

- Instrumentation Trace Macrocell (ITM)
- Data Watch point and Trace unit (DWT)
- Flash Patch and Breakpoint unit (FPB)

#### System Control Space (SCS):

- The registers in SCS address spaces are memory-mapped and can be accessed using pointers in software
- Secure SCS ( `0xE000E000` to `0xE000EFFF` ) - Secure software using this address space to access the banked Secure SCS registers (e.g., `SCB->CPUID` )
- Non-secure SCS ( `0xE000E000` to `0xE000EFFF` ) - Non-secure software using this address space to access the banked Non-secure SCS registers (e.g., `SCB->CPUID` )
- Non-secure alias SCS ( `0xE002E000` to `0xE002EFFF` ) - Secure software using this address space to access the Non-secure SCS registers (e.g., `SCB_NS->CPUID` )

The following table describes some core peripherals in the SCS and corresponding [data structures](#) defined in the CMSIS-Core header file to access the registers of core peripherals in two SCS address spaces.

Core Peripheral	Data Structure for Secure and NS SCS	Data Structure for NS Alias SCS
Implementation Control Block	SCnSCB (0xE000E004)	SCnSCB_NS (0xE002E004)
SysTick Timer	SysTick (0xE000E010)	SysTick_NS (0xE002E010)
Nested Vectored Interrupt Controller	NVIC (0xE000E100)	NVIC_NS (0xE002E100)
System Control Block	SCB (0xE000ECFC)	SCB_NS (0xE002ECFC)
Memory Protection Unit	MPU (0xE000ED90)	MPU_NS (0xE002ED90)
Security Attribution Unit	SAU (0xE000EDD0)	-
Debug Control Block	CoreDebug (0xE000EDF0)	CoreDebug_NS (0xE002EDF0)
Software Interrupt Generation	STIR (0xE000EF00)	STIR_NS (0xE002EF00)
Floating-Point Extension	FPU (0xE000EF34)	FPU_NS (0xE002EF34)

#### Notes:

- The SCB is a group of system control registers for the various usages below.
  - System Control Register (SCR) to configure processor low power mode
  - Fault Status Register (xFSR) to provide fault status information
  - [Vector Table Offset Register \(VTOR\)](#) for vector table relocation
- The [SAU](#) register is accessible from the Secure state only.
- The STIR register is not physically banked.
- Core peripherals such as SysTick, SCB, and MPU are duplicated. One instance is Secure and the other one is Non-secure.
- Secure software can use the corresponding functions for ARMv8-M in CMSIS-Core to configure the Non-secure [NVIC](#) and [SysTick](#) through the Non-secure alias SCS.

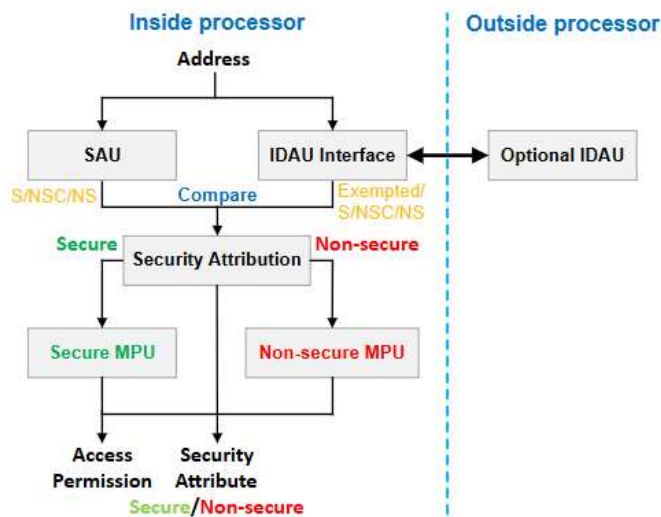
#### Debug or vendor specific components ( 0xE0040000 to 0xE00FFFFF ):

- Optional debug components (e.g., ETM)
- [External Private Peripheral Bus \(EPPB\)](#) allows designers to add their own debug or vendor-specific components
- [System ROM Table](#) is a simple lookup table that enables debug tools to extract the addresses of debug and trace components

## Secure Attribution Unit (SAU), Implementation Defined Attribution Unit (IDAU), and Memory Protection Unit (MPU)

Two units determine the security attribute of an address:

- The internal programmable [Secure Attribution Unit \(SAU\)](#).
- The external Implementation Defined Attribution Unit (IDAU), through the IDAU interface, returns the security attribute and region number of an address.



Three possible configurations to define the security attribute of an address:

- 1. Internal SAU only
- 2. External IDAU only
- 3. A combination of the internal SAU and external IDAU

Notes:

- Series 2 devices use configuration 3.
- IDAU in Series 2 devices is the [External Secure Attribution Unit \(ESAU\)](#).

The [Memory Protection Unit \(MPU\)](#) is a programmable unit that allows privileged software to define memory access permission. If the TrustZone is enabled, there can be up to two MPUs, one for Secure and one for Non-secure.

- The number of [MPU regions](#) for the Secure and the Non-secure MPU can be different.
- The MPU registers are memory-mapped and are placed in the [System Control Space \(SCS\)](#).
- Secure software can use the [MPU Functions for ARMv8-M](#) in CMSIS-Core to configure the Non-secure MPU through the [Non-secure alias SCS](#) ( 0xE002ED90 - 0xE002EDC4 ).

Software	Non-secure MPU Registers	Secure MPU Registers	MemManage Fault
Non-secure privileged	0xE000ED90 - 0xE000EDC4	-	Non-secure MPU violation
Secure privileged	0xE002ED90 - 0xE002EDC4	0xE000ED90 - 0xE000EDC4	Secure MPU violation

## Exceptions and Interrupts

### Type of Exceptions

The following table describes the types of exceptions in the TrustZone implemented system.

Section	Guidance	Type	Default State
1 (-)	Reset	Secure only	Secure
2 (-14)	NMI	Configurable	Secure
3 (-13)	HardFault	Configurable	Secure
4 (-12)	MemManage Fault	Banked	Banked
5 (-11)	BusFault	Configurable	Secure

Section	Guidance	Type	Default State
6 (-10)	UsageFault	Banked	Banked
7 (-9)	SecureFault	Secure only	Secure
11 (-5)	SVCall	Banked	Banked
12 (-4)	DebugMonitor	Configurable	Secure
14 (-2)	PendSV	Banked	Banked
15 (-1)	SysTick	Banked	Banked
16 - 495 (0 - 479)	IRQ0 - IRQ479	Configurable	Secure

Notes:

- "Secure only" means the system exceptions can only trigger in the Secure state.
- "Configurable" means the system exceptions and interrupts can be configured to target either the Secure state or the Non-secure state.
- Banked means the system exceptions can have Secure and Non-secure versions. Both can be triggered and executed independently and have different priority level settings.

Exception Priorities

It may cause a security issue if the Non-secure software uses high priority levels to mask the Secure interrupts. To avoid this issue, TrustZone introduces a programmable bit in the AIRCR register called PRIS (Prioritize Secure exception) for Secure software to prioritize, if needed, Secure exceptions and interrupts.

The AIRCR.PRIS is set to 0 out of reset, which means Secure and Non-secure exceptions/interrupts share the same configurable programmable priority level space (columns 2 and 3 in the following table). When the AIRCR.PRIS is set to 1, all Non-secure configurable exceptions/interrupts are placed in the lower half of the priority level space so that Secure exceptions/interrupts can potentially have higher priorities (columns 2 and 4 in the following table).

Priority Value	Secure Priority	Non-secure Priority (PRIS = 0)	Non-secure Priority (PRIS = 1)
0	0	0 (0x00)	128 (0x80)
1	32	32 (0x20)	144 (0x90)
2	64	64 (0x40)	160 (0xA0)
3	96	96 (0x60)	176 (0xB0)
4	128	128 (0x80)	192 (0xC0)
5	160	160 (0xA0)	208 (0xD0)
6	192	192 (0xC0)	224 (0xE0)
7	224	224 (0xE0)	240 (0xF0)

**Note:** This table uses three bits (Bit [7:5]) of the group priority level ( AIRCR.PRIGROUP ) to limit the maximum number of preemption levels to 8. A lower priority value indicates a higher priority.

Vector Tables

The following figure shows two vector tables for Secure and Non-secure exceptions and interrupts. The vector table offset is defined by a Vector Table Offset Register ( VTOR at 0xE000ED08 ), which can only be programmed in the privileged state.

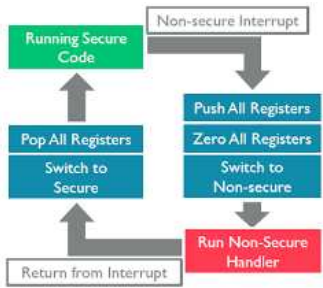
Exception number	IRQ number	Secure Vector	Non-secure Vector	Offset
495	479	IRQ479	IRQ479	0x7BC
.	.	.	.	.
.	.	.	.	.
18	2	IRQ2	IRQ2	0x48
17	1	IRQ1	IRQ1	0x44
16	0	IRQ0	IRQ0	0x40
15	-1	SysTick_S	SysTick_NS	0x3C
14	-2	PendSV_S	PendSV_NS	0x38
13	.	Reserved	Reserved	0x30
12	-3	DebugMonitor	DebugMonitor	.
11	-5	SVCall_S	SVCall_NS	0x2C
10	.	Reserved	Reserved	.
9	.			.
8	.	SecureFault		0x1C
7	-9			.
6	-10	UsageFault_S	UsageFault_NS	0x18
5	-11	BusFault_S	BusFault_NS	0x14
4	-12	MemManage_S	MemManage_NS	0x10
3	-13	HardFault_S	HardFault_NS	0x0C
2	-14	NMI_S	NMI_NS	0x08
1	.	Reset		0x04
.	.	Initial SP value		0x00

Image: [Vector Table](#). Copyright © 1995-2022 Arm Limited (or its affiliates). All rights reserved.

Notes:

- The `VTOR_S` defines the address of the Secure vector table in Secure memory, and the [Secure Main Stack Pointer](#) ( `MSP_S` ) is the default stack for the Secure exception handler.
- The `VTOR_NS` defines the address of the Non-secure vector table in Non-secure memory, and the [Non-secure Main Stack Pointer](#) ( `MSP_NS` ) is the default stack for the Non-secure exception handler.
- Secure privileged software can access the `VTOR_NS` using the [Non-secure SCB alias](#) ( `0xE002ED08` ).
- The [System Control Space](#) contains registers for the SysTick timer, NVIC, and SCB.
- The interrupt masking registers ( `PRIMASK`, `FAULTMASK`, and `BASEPRI` ) are [banked](#) between security states. The priority level space is shared between the Secure and the Non-secure world, setting an interrupt mask register on one side can block some, or all, of the exceptions on the other side.
- Interrupts ( `IRQ0` - `IRQ479` ) are defined as Secure by default. Each interrupt can be configured as Secure or Non-secure and is determined by the Interrupt Target Non-secure ( `NVIC_ITNS` ) register, which is only programmable in the Secure software.

State Transitions in Exceptions and Interrupts



The following figure shows transitions between the [processor states](#) in ARMv8-M TrustZone.

Image (left): [Switching-between-Secure-and-Non-secure-states](#). Copyright © 1995-2022 Arm Limited (or its affiliates). All rights reserved.

1. Secure Thread → Secure Handler or Non-secure Thread to Non-secure Handler
  - No security state transition.
  - The exception sequence is almost identical to the exception stacking mechanism of current Cortex-M processors.

- The Interrupt Service Routine (ISR) is executed in the current security state (either Secure or Non-secure).
- 2. Non-secure Thread → Secure Handler or Non-secure Handler → Secure Handler
  - The transition from Non-secure to Secure state.
  - The exception sequence is almost identical to the exception stacking mechanism of current Cortex-M processors.
  - The ISR is executed in a Secure state.
- 3. Secure Thread → Non-secure Handler or Secure Handler → Non-secure Handler
  - The transition from Secure to Non-secure state.
  - To avoid an information leak when transitioning from the Secure to Non-secure state. The processor automatically pushes all general-purpose registers into the Secure stack and erases the contents of all general-purpose registers before executing the Non-secure ISR. The processor pops the contents of all general-purpose registers from the Secure stack when returning from the Non-secure ISR (right side in [Figure 2.6 State Transitions on page 12](#)). It incurs a slightly longer interrupt latency.
  - The ISR is executed in a Non-secure state.
- 4. Secure Privileged Thread ↔ Non-secure Privileged Thread or Secure Unprivileged Thread ↔ Non-secure Unprivileged Thread
  - The transition from Secure to Non-secure state or Non-secure to Secure state.
  - The [Function calls and returns](#) can be used when the privileged level remains the same.

**Note:** Subject to interrupt priority, there are no restrictions regarding whether a Non-secure or Secure interrupt can occur when the processor runs Non-secure or Secure code.

## Switching Between Secure and Non-secure States

The TrustZone allows direct calling between Secure and Non-secure software. The following figure shows how to use an API function call to trigger security state transitions. The state transitions can also happen because of [exceptions and interrupts](#).

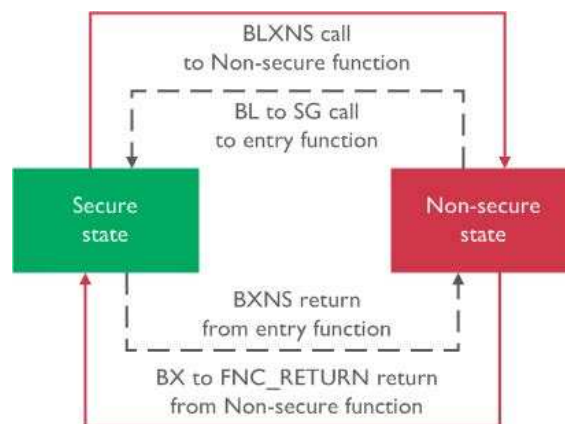


Image: [Switching-between-Secure-and-Non-secure-states](#). Copyright © 1995-2022 Arm Limited (or its affiliates). All rights reserved.

### Switching from Non-secure to Secure State

When the Non-secure program calls a Secure software, the first instruction must be a Secure Gateway ( `SG` ) instruction residing in Non-secure Callable memory. The Secure Gateway entry points (veneers) decouple the address of the `SG` instructions in the Non-secure Callable memory region from the rest of the Secure code. It can eliminate the risk of having inadvertent entry points when the Secure software contains a pattern that matches the opcode of the `SG` instruction.



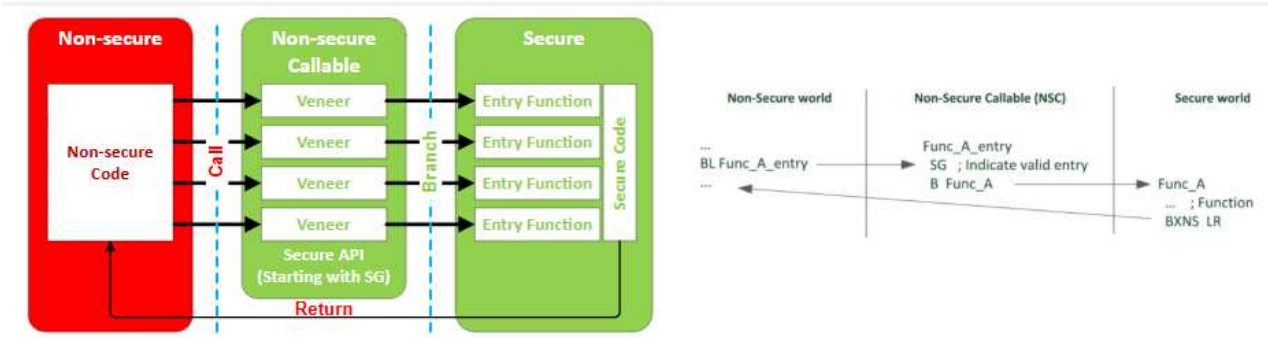


Image (right): *Whitepaper - ARMv8-M Architecture Technical Overview. Copyright © 1995-2022 Arm Limited (or its affiliates). All rights reserved.*

The bit 0 of the [Link Register \(LR\)](#) is cleared to zero by `SG` instruction to indicate that returning from this function transits from Secure to Non-secure. The processor is still in the Non-secure state when the `SG` instruction is executed. The `BXNS LR` instruction is used when returning since a normal `BX LR` instruction interprets it as an unsupported execution mode change. A [SecureFault](#) exception is triggered if the processor returns to a Secure address. It prevents a hacker from calling a Secure API with a fake return address pointing to a Secure program location. If bit 0 of LR is 1, the `BXNS LR` instruction behaves like a normal `BX LR`. Therefore, Secure code can call a Secure API in the NSC region even it is not a usual practice.

Program	Call Instruction	SG Instruction	Return Instruction
Non-secure call Non-secure	BL or BLX	-	BX LR (Return to Non-secure state)
Non-secure call Secure	BL or BLX	Clear bit 0 of LR	BXNS LR (Return to Non-secure state)
Secure call Secure	BL or BLX	Set bit 0 of LR	BXNS LR (Return to Secure state)

To help software developers create Secure APIs in C/C++, the [Cortex-M Security Extension \(CMSE\)](#) defines a C function attribute called `cmse_nonsecure_entry`.

- GCC — `__attribute__((cmse_nonsecure_entry))`
- IAR — `__cmse_nonsecure_entry`

### Test Target (TT) Instruction

The software can use an ARMv8-M instruction called Test Target (TT) and the region number generated by the SAU or the IDAU to determine if a contiguous range of memory shares common security attributes and privilege levels.

The TT instruction returns the [SAU/IDAU](#) region number, security attributes (S/NS), and MPU region number after passing the start and end addresses of the memory range to the TT instruction. The software can determine whether the memory range has required security attributes and resides in the same region number.



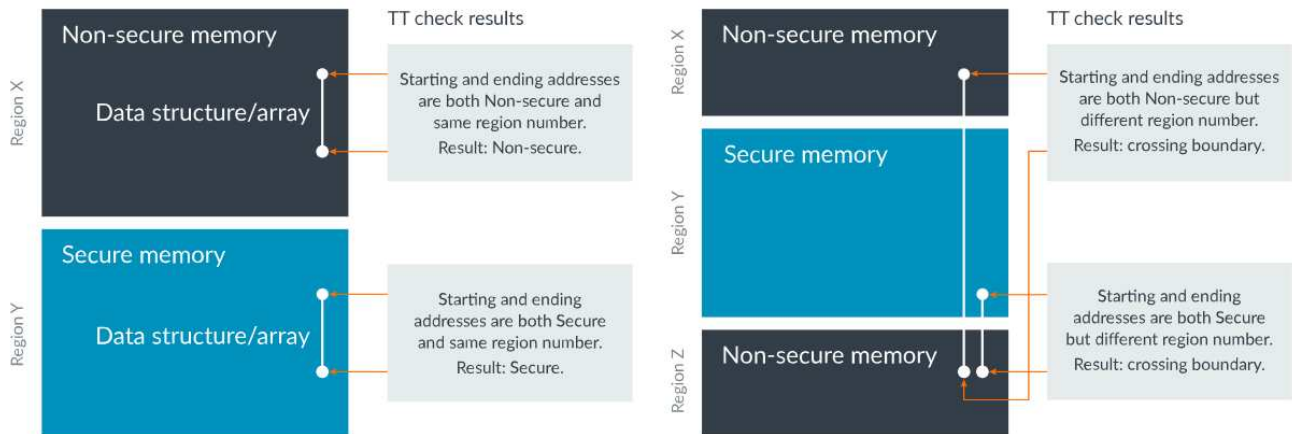


Image: [Test-target-instruction](#). Copyright © 1995-2022 Arm Limited (or its affiliates). All rights reserved.

This mechanism allows security checking at the beginning of the API service (instead of during the operation) to determine if the memory referenced by a pointer from Non-secure software points to the Non-secure address. It prevents Non-secure software from using those APIs in Secure software to access or modify Secure data.

To make these operations easier in a C/C++ programming environment, the [Cortex-M Security Extension \(CMSE\)](#) has defined a range of [intrinsic functions](#) for dealing with pointer checks with the TT instructions.

## Switching from Secure to Non-secure State

When the Secure program calls a Non-secure software, the Secure program must use a `BLXNS <reg>` instruction to invoke the process. If bit 0 of the `<reg>` is 0, the processor must switch to the Non-secure state when branching to the target address. During the state transition, the return address and some processor state information are pushed onto the Secure stack, while the return address on the [Link Register \(LR\)](#) is set to a special value called `FNC_RETURN` (0xFEFFFFFF).

The Non-secure function completes by performing a branch (`BX LR`) to the `FNC_RETURN` address (bit 0 is 1 to indicate the function was called from the Secure state). It automatically triggers the unstacking of the actual return address from the Secure stack and returns to the calling function. The `FNC_RETURN` hides the return address of the Secure program from the Non-secure software to avoid the leakage of any secret information. It also prevents Non-secure software from modifying the Secure return address stored in the Secure stack.

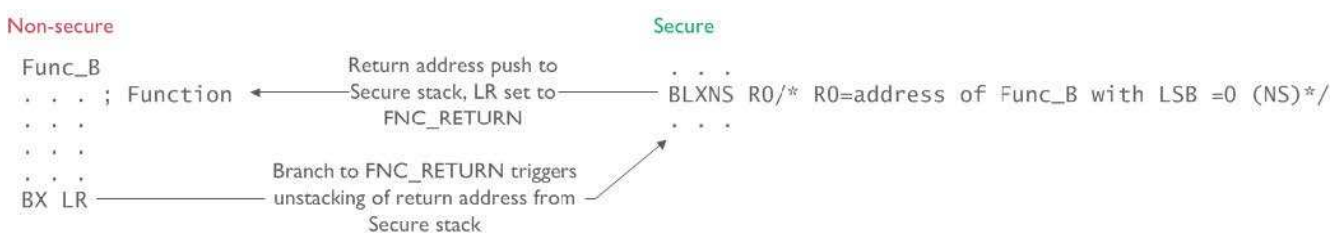


Image: [Switching-between-Secure-and-Non-secure-states](#). Copyright © 1995-2022 Arm Limited (or its affiliates). All rights reserved.

To help software developers declare Non-secure function pointers in C/C++, the [Cortex-M Security Extension \(CMSE\)](#) defines a C function attribute called `cmse_nonsecure_call`.

- GCC: `__attribute__((cmse_nonsecure_call))`
- IAR: `__cmse_nonsecure_call`

## Software Flow

The following figure describes a software flow example in a TrustZone implemented system.

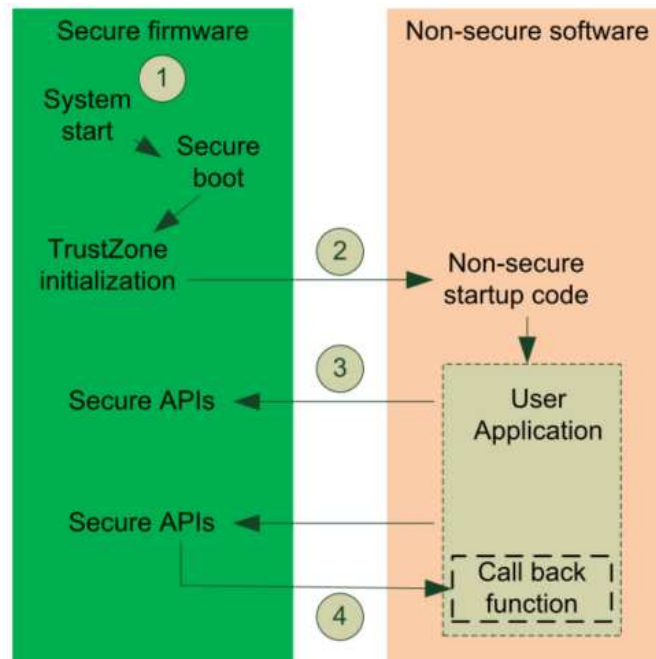


Image: [Software Development in ARMv8-M Architecture](#). Copyright © 1995-2022 Arm Limited (or its affiliates). All rights reserved.

- The system starts executing code in the Secure state after a power-on or reset (Secure boot).
  - The Secure [stack pointer](#) ( `MSP_S` ) is set from the address of the Secure vector table ( `VTOR_S` ).
  - The Secure Reset Handler pointed by the `VTOR_S` is called.
  - Perform various initialization tasks such as C startup code.
  - Place peripherals and associated interrupts in either Secure or Non-secure applications.
  - Program [SAU/IDAU](#) to partition the entire memory into Secure, Non-secure Callable, and Non-secure regions.
  - Program the address of the Non-secure vector table ( `VTOR_NS` ).
  - Initialize the two first entries of the table for the Non-secure stack pointer ( `MSP_NS` ) and Reset Handler to emulate a Non-secure reset.
- The Secure firmware branches to the entry point (Reset Handler pointed by the `VTOR_NS` ) of the Non-secure application.
  - The Non-secure software has its Reset Handler.
  - Perform various initialization tasks such as C startup code and hardware initialization (e.g., Non-secure peripherals).
  - It does not conflict with initialization from the Secure code as the stack and heap spaces of Secure and Non-secure code are separated.
- During the execution of Non-secure applications, the application could call Secure APIs through the [Secure Gateway \(SG\) veneer](#) in the Non-secure Callable region.
- In some cases, Secure APIs might need to call [Non-secure call-back functions](#) (e.g., a hardware driver).

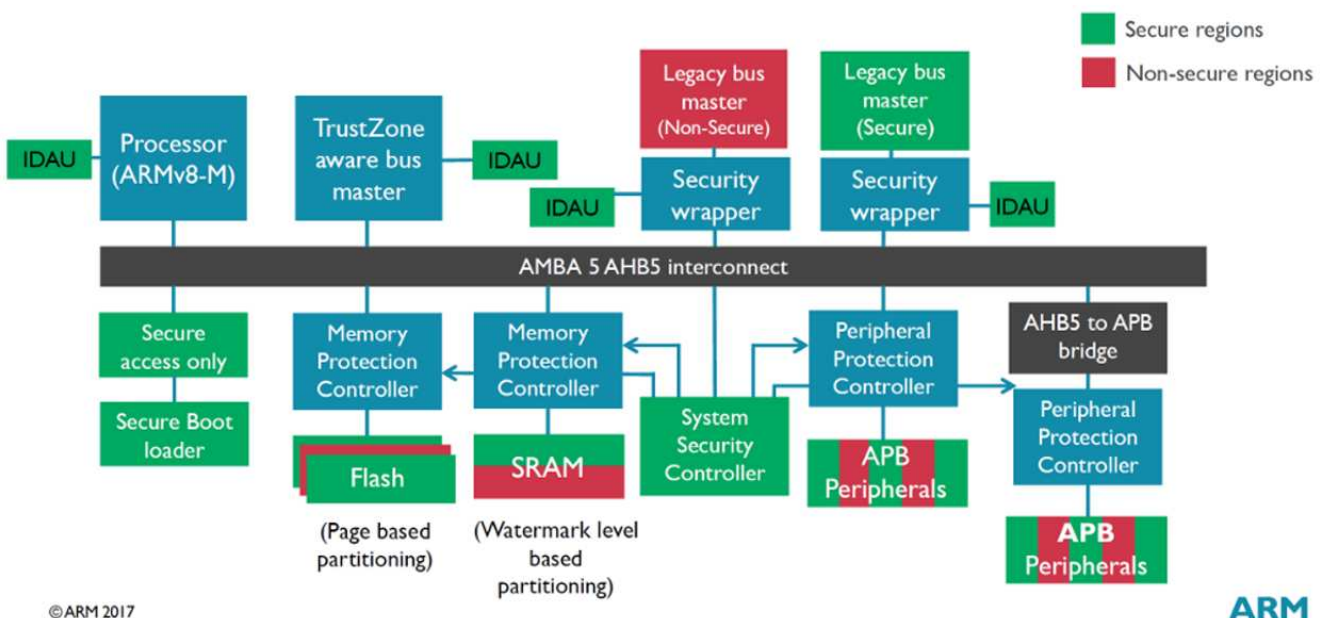
## Bus Level Security (BLS)

# Bus Level Security (BLS)

## System Design

The following figure shows two system designs:

- The sample system contains an ARMv8-M processor and the required components to support TrustZone.
- Bus Level Security (BLS) on Series 2 devices implements the concepts introduced in the ARM TrustZone sample system. BLS enforces Secure and privileged programming models and uses security components (colored blocks) to configure the security attribute and privileged level of peripherals and Bus Masters.



## ARMv8-M Processor

The ARMv8M processor is TrustZone capable of Secure and Non-secure states. It has a dedicated internal [SAU](#) that is fully programmable up to 8 different memory regions. Out of reset, the processor is in a Secure state and every transaction is a Secure transaction.

ARMv8-M Processor in Series 2 devices is the Cortex-M33.

## System Security Controller

The system security controller is the central location for all security settings in the system. Each type of controller, IDAU, and wrapper receives its security configuration and bus response configuration from this block.

System Security Controller in Series 2 devices is the [Security Management Unit \(SMU\)](#).

## Implementation Defined Attribution Unit (IDAU)

The IDAU generates the security attribute for a given address. All IDAUs in the system have the same memory partitioning. The IDAU is intended only for ARMv8-M cores and utilizes the entire IDAU interface for the core. The lite IDAU uses only the Secure and Non-secure interface from the IDAU and is intended for Non-ARMv8-M Bus Masters.

IDAU in Series 2 devices is the [External Secure Attribution Unit \(ESAU\)](#).

## Security Wrapper

The Security Wrapper gives a legacy Bus Master the ability to drive security attribution. The security wrapper outputs the transaction address to the lite IDAU which returns the security attribute of the address. If the wrapper is configured as Non-secure, any transactions to a Secure address are blocked.

Security Wrapper in Series 2 devices is the [Bus Master Protect Unit \(BMPU\)](#).

## Memory Protection Controller (MPC)

MPC has a security configuration for a per block of memory or memory above and below the watermark. If the security attribute of the block or memory region does not match the security attribute of the address, the transaction is blocked. This controller is used in a system that alias RAM or flash memory locations. This controller is not needed when the memory region size is programmable in an IDAU.

Series 2 devices have a programmable flash and RAM region in the [ESAU](#) (equivalent to IDAU) and are not implementing this block.

## Peripheral Protection Controller (PPC)

PPC has a security configuration for every peripheral. If the security attribute of the selected peripheral does not match the security attribute of the address, the transaction is blocked. This controller is used in systems that alias the peripheral memory locations.

PPC in Series 2 devices is the [Peripheral Protection Unit \(PPU\)](#).

Hardware security is now extended to the peripheral bus system of the processor. Each component on the bus can verify and propagate the security level for each bus operation. The following sections describe the individual security component for BLS on Series 2 devices.

## Security Management Unit (SMU)

The SMU is the only user-facing block in the BLS architecture and houses all the configuration and status for the [ESAUs](#), [BMPUs](#), and [PPUs](#).

- Thirteen memory regions ([ESAU](#))
- Per Bus Master privileged and security attribute ([BMPU](#))
- Interrupt flag for Bus Master security fault (fault table in BMPU section)
- Per peripheral privileged and security attribute ([PPU](#))
- Interrupt flags for privileged, security, and instruction peripheral access faults (fault tables in PPU section)
- Separate Secure and Privileged IRQ

The SMU configurations can be [locked](#) down and protected from runaway code. The `SMU_LOCK` register resets to UNLOCK. Any write other than the unlock code ( 0xACCE55 ) locks all SMU registers from further updates. The `SMU_STATUS` register contains a `SMU_LOCK` bitfield with the current lock state of the SMU.

The `SMU_M33CTRL` register can [lock](#) down internal security and privileged configurations below.

- Cortex-M33 SAU
- Non-secure MPU
- Secure MPU
- Non-secure Vector Table Offset Register (VTOR)
- Secure AIRCR register

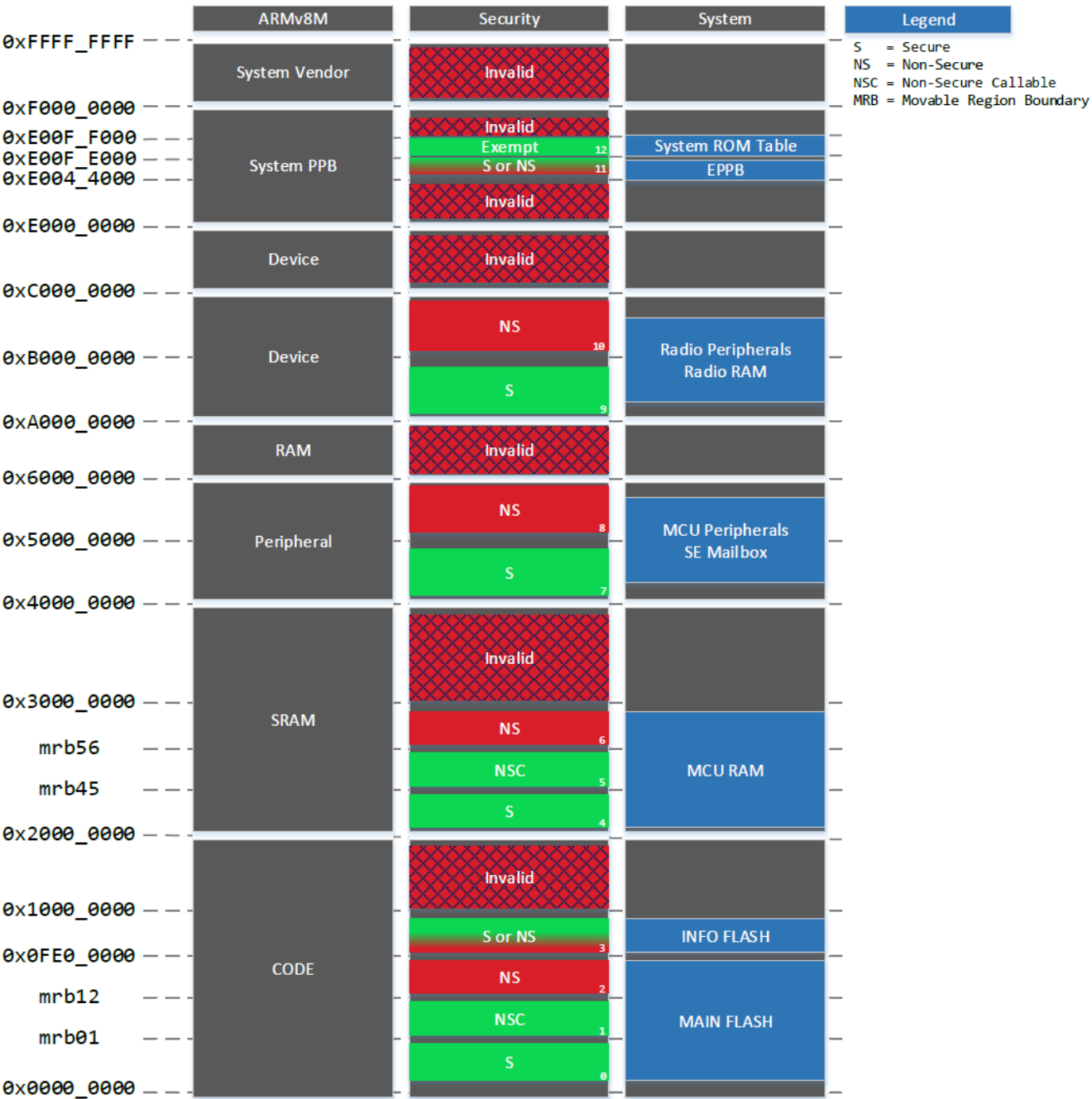
Interrupt flags in the `SMU_IF` register can generate a [Secure or Privileged interrupt](#) in the table below when its corresponding interrupt enable bit in the `SMU_IEN` register is set and `IRQn` is enabled.

Enable Bit in SMU_IEN Register	IRQn	Interrupt Handler
BMPUSEC, PPUSEC	SMU_SECURE_IRQn	SMU_SECURE_IRQHandler()
PPUPRIV, PPUINST	SMU_PRIVILEGED_IRQn	SMU_PRIVILEGED_IRQHandler()

Each interrupt flag in the `SMU_IF` register can be cleared by writing 1 to the corresponding bit of the `SMU_IF_CLR` register.

## External Secure Attribution Unit (ESAU)

The ESAU is responsible for determining the memory region and [security attribute](#) of a given address. Referring to [ARMv8-M TrustZone Implementation on page 16](#), the Cortex-M33 interfaces with an ESAU and the BMPUs of other Bus Masters interface with lite ESAUs to determine the security attribute of all transactions. The following figure describes the security attributes of different memory regions defined by the ESAU on Series 2 devices.



**Notes:**

- For Series 2 devices with base address `0x08000000` in region 0, the memory address from `0x0` to `0x07FFFFFF` is an invalid region.
- The invalid regions are deemed as Secure.
- The NSC and Exempted attributes are only available to the ESAU, and all lite ESAUs in the system view these attributes as [Secure](#).

The ESAU divides the memory map into 13 memory regions and has a maximum of 6 Non-secure regions.

- Four Movable Region Boundaries (MRBs) determine the size of 6 regions.
- Two regions have configurable security attributes.
- Each memory region consists of a base address that specifies the start of the region and a limit address that specifies the end of the region plus one (+ 1).
- The address is valid if it falls between the base ( $\geq$  base) and limit ( $<$  limit) of a region.
- If the memory region is not defined, it is deemed invalid and Secure.

The MRBs distinguish the Secure, Non-secure Callable, and Non-secure regions in flash and RAM. The two configurable regions determine if the Info flash and Cortex-M33 [EPPB](#) regions are Secure or Non-secure. The MRBs have a specific programming sequence. Any [misprogramming](#) results in a `SMUPRGERR` in the `SMU_STATUS` register.

**ARMv8-M CODE Regions**

- [Regions 0, 1, and 2](#) are in the Main space of flash. Region 3 is the info space of flash.
- The `mrb01` (`ESAUMRB01` in `SMU_ESAUMBR01` register) determines the end of region 0 and the start of region 1.
- The `mrb12` (`ESAUMRB12` in `SMU_ESAUMBR12` register) determines the end of region 1 and the start of region 2.
- The size of region 3 is device-dependent.
- Three regions' security attributes are static, and one region is configurable. Region 0 is always Secure, region 1 is always Non-secure Callable, and region 2 is always Non-secure. [Region 3](#) is configurable as either Secure or Non-secure (`ESAUR3NS` in `SMU_ESAURTYPE0` register, default is secure after reset).
- Sizes of regions 0, 1, and 2 are adjusted in **4 kB increments** with the lower 12 bits of `ESAUMRB##` in `SMU_ESAUMBR##` ignored.
  - The Secure region can be set to size 0 when `mbr01` = base address of region 0.
  - The Non-secure Callable regions can be set to size 0 when `mbr01` = `mbr12`.
- The default value of `mbr01` is equal to base address + `0x02000000`, so the size of region 0 is 32 MB. Out of reset, all flash is Secure since all Series 2 devices have less than 32 MB of flash.

Region	Memory	Base Address	Limit Address	Security Attribute
0	Main flash	<code>0x00000000</code> or <code>0x08000000</code>	<code>(0x00000000 or 0x08000000)   mbr01</code>	Secure
1	Main flash	<code>(0x00000000 or 0x08000000)   mbr01</code>	<code>(0x00000000 or 0x08000000)   mbr12</code>	Non-secure Callable
2	Main flash	<code>(0x00000000 or 0x08000000)   mbr12</code>	<code>0x0FE00000</code>	Non-secure
3	Info flash	<code>0x0FE00000</code>	<code>0x10000000</code>	Secure or Non-secure

**ARMv8-M RAM Regions**

- [Regions 4, 5, and 6](#) cover the entire available RAM in the device.
- The `mrb45` (`ESAUMRB45` in `SMU_ESAUMBR45` register) determines the end of region 4 and the start of region 5.
- The `mrb56` (`ESAUMRB56` in `SMU_ESAUMBR56` register) determines the end of region 5 and the start of region 6.
- All three regions' security attributes are static. Region 4 is always Secure, region 5 is always Non-secure Callable, and region 6 is always Non-secure.
- Sizes of all three regions are adjusted in **4 kB increments** with the lower 12 bits of `ESAUMRB##` in `SMU_ESAUMBR##` ignored.
  - The Secure region can be set to size 0 when `mbr45` = base address of region 4.



The Non-secure Callable region can be set to size 0 when mbr45 = mbr56.

- The default value of mbr45 is equal to 0x02000000, so the size of region 4 is 32 MB. Out of reset, all RAM is Secure since all Series 2 devices have less than 32 MB of RAM.

Region	Memory	Base Address	Limit Address	Security Attribute
4	SRAM	0x20000000	0x20000000   mbr45	Secure
5	SRAM	0x20000000   mbr45	0x20000000   mbr56	Non-secure Callable
6	SRAM	0x20000000   mbr56	0x30000000	Non-secure

## ARMv8-M Peripheral Regions

- These regions are aliases to the [chip peripherals and SE mailbox](#) (a device with HSE).
- Both regions have a fixed size.
- Both regions' security attributes are static. Region 7 is always Secure, and region 8 is always Non-secure.

Region	Memory	Base Address	Limit Address	Security Attribute
7	Chip Peripherals	0x40000000	0x50000000	Secure
8	Chip Peripherals	0x50000000	0x60000000	Non-secure

## ARMv8-M Device Regions

- These regions are aliases to all radio peripherals and radio RAM.
- Both regions have a fixed size.
- Both regions' security attributes are static. Region 9 is always Secure, and region 10 is always Non-secure.
- From the perspective of the device bus system, the [radio is one peripheral that is either Secure or Non-secure](#). So any Bus Master accessing the radio needs to know the security attribute of the radio. From the perspective of the radio, all of its radio bus peripherals are accessible regardless of the security attribute. However, the radio needs to know the security attribute of chip bus peripherals to access them through the correct alias.

Region	Memory	Base Address	Limit Address	Security Attribute
9	Radio Peripherals	0xA0000000	0xB0000000	Secure
10	Radio Peripherals	0xB0000000	0xC0000000	Non-secure

## ARMv8-M System Private Peripheral Bus (PPB) Regions

- Both regions have a fixed size.
- [Region 11](#) is the Cortex-M33 EPPB memory region and is configurable as either Secure or Non-secure (ESAU11NS in SMU\_ESAURTPES1 register, default is secure after reset). It is important to note that the Cortex-M33 core is the only Bus Master that sees these memory regions. All other Bus Masters in the system do not have access to the System PPB, and it is an invalid region.
- Region 12 has a static security attribute of Exempted. It means that the Cortex-M33 core allows the transaction in all cases. It permits debuggers to read the system ROM Table regardless of the state of the Cortex-M33 core.

Region	Memory	Base Address	Limit Address	Security Attribute
11	EPPB	0xE0044000	0xE00FE000	Secure or Non-secure
12	System ROM Table	0xE00FE000	0xE00FF000	Exempted

### Notes:

- The regions in flash (0/1/2) and RAM (4/5/6) can only create in the order of Secure, Non-secure Callable, and Non-secure.
-

The [ESAU and lite ESAUs](#) handle the transactions of Bus Masters and must have consistent security attribute mapping. Therefore, configurations in the SMU registers apply to ESAU and lite ESAUs.

- Unlike other Bus Masters using B MPU and lite ESAU, merging the address lookup results from the [internal SAU and ESAU](#) determines the [security attribute](#) of the Cortex-M33 transaction.

Bus Master	Security Attribution
Cortex-M33	SAU and ESAU
Other	Lite ESAU

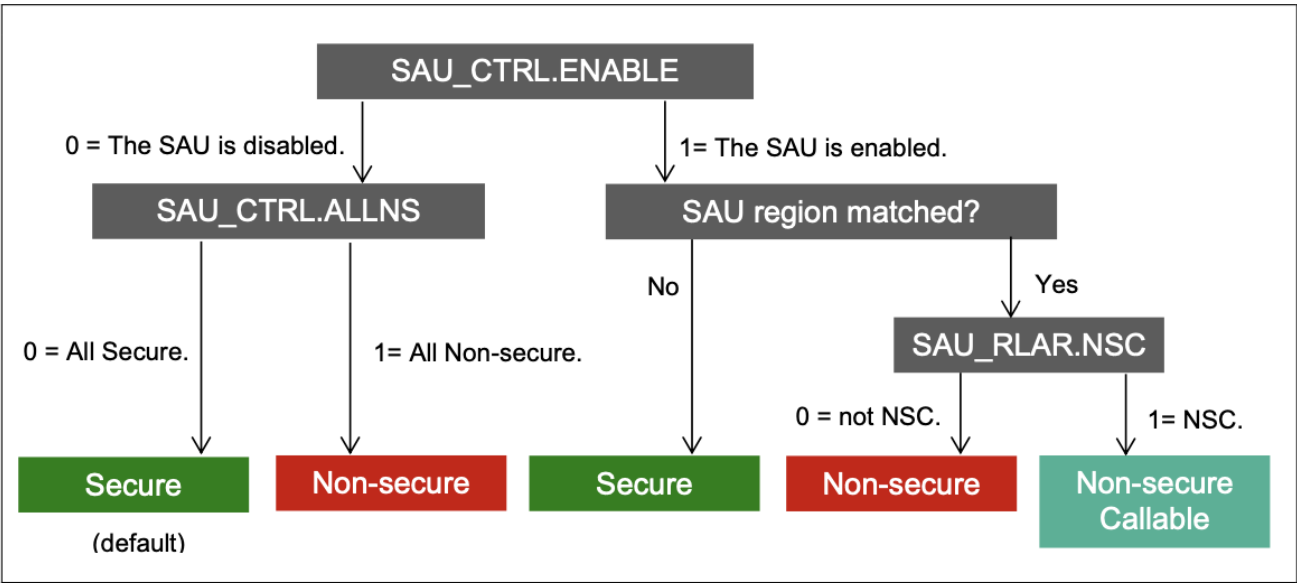
## Security Attribution Unit

In Series 2 devices, the combination of the integrated SAU in the Cortex-M33 processor and an ESAU determine the security attribute of a Cortex-M33 transaction.

The SAU consists of several [programmable registers](#). These registers are placed in the [System Control Space \(SCS\)](#) and are only accessible from the Secure privileged state.

- SAU Control Register ( `SAU_CTRL` ) — The SAU is disabled after RESET
- SAU Type Register ( `SAU_TYPE` ) — Indicates the number of [available regions](#) (read-only)
- SAU Region Number Register ( `SMU_RNR` ) — Assigns a region number
- SAU Region Base Address Register ( `SAU->RBAR` ) — Configures selected region base address
- SAU Region Limit Address Register ( `SAU->RLAR` ) — Configures selected region limit address and security attribute (NSC or NS), enable or disable the region

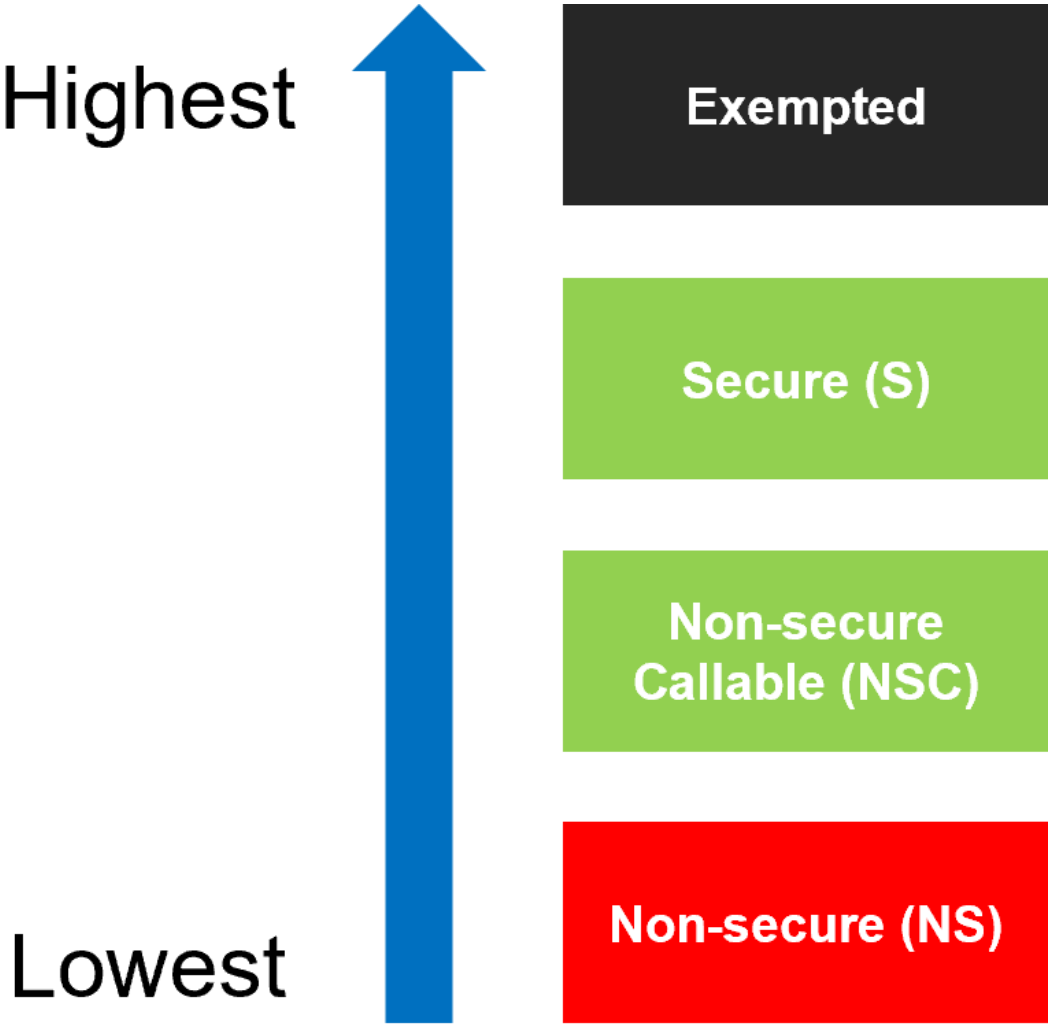
The following figure shows three different SAU configurations for determining the security attribute of a Cortex-M33 transaction.



Notes:

- All address ranges after RESET in SAU are Secure by default.
- The SAU can configure a **32 bytes aligned** region as Non-secure or Non-secure Callable. Any address not defined in the SAU defaults to Secure.
- An [ESAU](#) can configure or hard-code a region as Secure, Non-secure Callable, Non-secure, or Exempted. An [Exempted](#) region enables Non-secure debuggers to access debugging components and establish a debug connection to the processor before the SAU is configured.
- The processor determines the final attribute of the address based on the higher security attribute (Exempted > S > NSC > NS) from either the SAU or the ESAU.





All Secure Configuration

Highlights:

- SAU is disabled.
- ALLNS bit in the SAU Control register is clear.
- The whole memory is in a Secure state (highest security attribute apart from Exempted).
- All Cortex-M33 transactions in this configuration are Secure or Exempted and give the Cortex-M33 access to all memory locations through either the Secure or Non-secure alias after RESET.

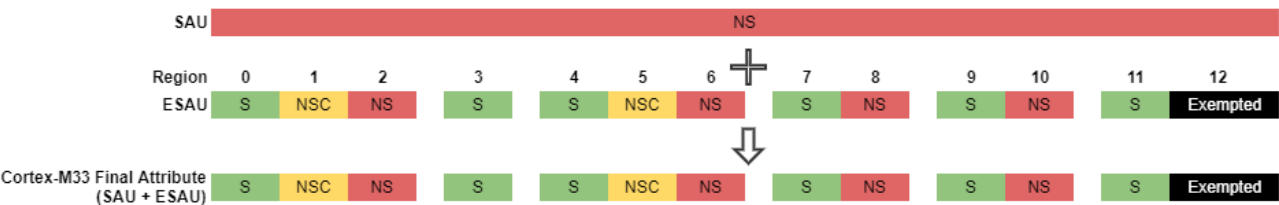
SAU	S												
Region	0	1	2	3	4	5	6	7	8	9	10	11	12
ESAU	S	NSC	NS	S	S	NSC	NS	S	NS	S	NS	S	Exempted
+													
↓													
Cortex-M33 Final Attribute (SAU + ESAU)	S		S		S		S		S		S		Exempted

- It is up to the boot procedure in a Secure state to keep the current configuration or use other configurations once the boot process is complete.

All Non-secure Configuration

Highlights:

- SAU is disabled.
- ALLNS bit in the SAU Control register is set.
- The whole memory is in a Non-secure state (lowest security attribute).
- Therefore the ESAU configuration determines the security attribute of all Cortex-M33 transactions.

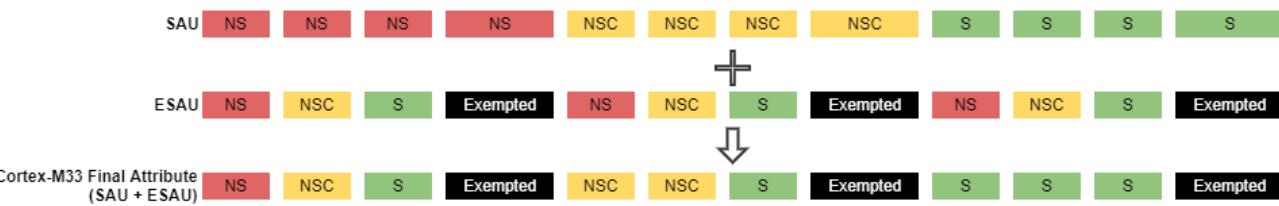


- Except for the SAU\_CTRL register, this configuration does not require programming on other SAU registers.

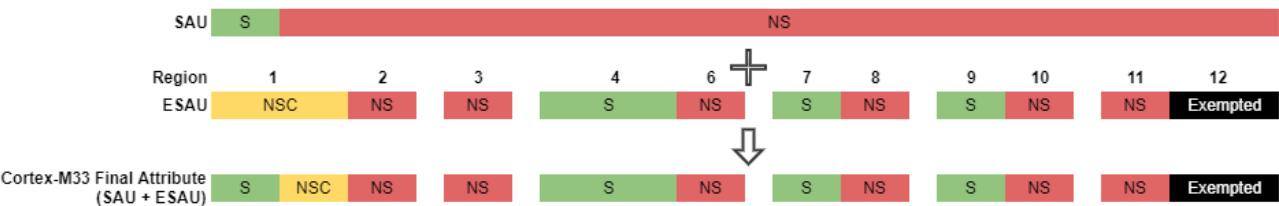
Configurable Configuration

Highlights:

- SAU is enabled.
- ALLNS bit in the SAU Control register can be 0 or 1 (do not care).
- The NSC bit on the SAU\_RLAR register determines the security attribute of an address as Non-secure or Non-secure Callable if an address matches an SAU region.
- The security attribute of an address is Secure by default if the address does not match any SAU region.
- This configuration programs SAU\_RNR, SAU\_RBAR, and SAU\_RLAR registers to correlate the Non-secure regions in ESAU.
- The SAU or ESAU overrides the attribute to a higher security level if any security attribute mismatch occurs in a memory region.



- The following figure is an example of a configurable configuration with the size of ESAU regions 0 and 5 are set to zero.



**Note:** The Cortex-M33 has an internal SAU that defaults all undefined addresses to Secure if enabled. If the Secure regions do not align between the [Cortex-M33 \(SAU + ESAU\)](#) and [other Bus Masters \(lite ESAU\)](#), the Cortex-M33 treats a memory region as Secure while other Bus Masters treat it as Non-secure. It can lead to the leaking of secure data if the Cortex-M33 stores secure data in what other Bus Masters think is a Non-secure area ([Main Flash Layout on page 34](#)).

## Bus Master Protection Unit (BMPU)

The BMPU is a security wrapper used for assigning a Bus Master specific security and privileged states. Referring to [Figure 3.1 ARMv8-M TrustZone Implementation on page 16](#), the BMPU generally lies between the Bus Master and the Advanced High-performance Bus (AHB) Matrix. BMPU interfaces with a [lite ESAU](#) to determine the security attribute of all Bus Master transactions.

The registers below in SMU configure the [security](#) and [privileged](#) state of a Bus Master. The Bus Masters in group 0 are device-dependent. Out of reset, each Bus Master is Secure and privileged.

Register	Description
SMU_BMPUPATD0	Bitfields (privileged if set) for privileged attribute configuration on Bus Master group 0
SMU_BMPUSATD0	Bitfields (Secure if set) for security attribute configuration on Bus Master group 0

**Note:** The Bus Master privileged attribute only applies to peripheral accesses. Flash and RAM accesses ignore the privileged attribute of the Bus Master.

The BMPU generates a security fault when the security attribute of the bus transaction is Secure, and the security attribute ( `SMU_BMPUSATD0` ) for the BMPU is configured as Non-secure.

Below is the security fault table that shows how the security attribute on the bus is driven based on the lite ESAU attribute and the BMPU security configuration. The interrupt is triggered if `BMPUSEC` in `SMU_IEN` is set and the `SMU_SECURE_IRQn` is enabled.

Lite ESAU Attribute	Secure Bus Master	Non-secure Bus Master
Non-secure	Non-secure	Non-secure
Secure	Secure	FAULT

Upon a BMPU fault, the registers in SMU below notify that a BMPU security fault occurred and on which Bus Master. The registers also identify the offending fault address. If a fault is detected, the response is Read As Zero (RAZ) or Write Ignored (WI) and the corresponding interrupt flag is set in the `SMU_IF` register. The values in `SMU_BMPUFS` and `SMU_BMPUFSADDR` do not change until the BMPU fault ( `BMPUSEC` ) in the `SMU_IF` register is cleared by software.

Register	Bitfield	Fault
SMU_IF	BMPUSEC	Security Fault if set
SMU_BMPUFS	BMPUFSMASTERID	ID of the Bus Master that triggered the fault
SMU_BMPUFSADDR	BMPUFSADDR	Access address that triggered the fault

**Note:** No privileged fault is generated because all the other Bus Masters in the system do not drive the privileged attribute.

## Peripheral Protection Unit (PPU)

The PPU is a security wrapper used for assigning a Bus Slave peripheral specific security and privileged states. Referring to [Figure 3.1 ARMv8-M TrustZone Implementation on page 16](#), the PPU comes in the form of a PPU in Advanced High-performance Bus (AHB) and a PPU in Advanced Peripheral Bus (APB).

- The PPU AHB generally lies between the Bus Matrix and an AHB Bus Slave peripheral.
- The PPU APB lies between the output of an AHB to APB bridge and all of the APB Slaves on that APB bus.

The registers below in SMU configure the [security](#) and [privileged](#) state of a peripheral. The peripherals in groups 0 and 1 are device-dependent. Out of reset, each peripheral is Secure and privileged. While each peripheral in address `0x40000000` (region 7) or `0x50000000` (region 8) can be configured independently, the radio subsystem in `0xA0000000` (region 9) or `0xB0000000` (region 10) is configured as a [unit](#).

Register	Description
SMU_PPUPATD0	Bitfields (privileged if set) for privileged access configuration on peripheral group 0
SMU_PPUPATD1	Bitfields (privileged if set) for privileged access configuration on peripheral group 0
SMU_PPUSATD0	Bitfields (Secure if set) for security access configuration on peripheral group 0
SMU_PPUSATD1	Bitfields (Secure if set) for security access configuration on peripheral group 1

The PPU can generate three types of faults:

1. Privileged faults occur on unprivileged transactions to privileged peripherals. Below is the privileged fault table that shows when a privileged fault occurs based on the PPU peripheral privileged configuration and the bus transaction privileged attribute. The interrupt is triggered if `PPUPRIV` in `SMU_IEN` is set and the `SMU_PRIVILEGED_IRQn` is enabled.

Bus Attribute	Privileged Peripheral	Unprivileged Peripheral
Privileged	SUCCESS	SUCCESS
Unprivileged	FAULT	SUCCESS

2. Security faults occur on Secure transactions to Non-secure peripherals and Non-secure transactions to Secure peripherals. Below is the security fault table that shows when a security fault occurs based on the PPU Peripheral security configuration and the bus transaction security attribute. The interrupt is triggered if `PPUSEC` in `SMU_IEN` is set and the `SMU_SECURE_IRQn` is enabled.

Bus Attribute	Secure Peripheral	Non-secure Peripheral
Secure	SUCCESS	FAULT
Non-secure	FAULT	SUCCESS

3. Instruction faults occur on any transaction marked as an instruction fetch. Below is the instruction fault table that shows when a PPU instruction fault occurs based on the bus transaction instruction attribute. The interrupt is triggered if `PPUINST` in `SMU_IEN` is set and the `SMU_PRIVILEGED_IRQn` is enabled.

Bus Attribute	Secure Peripheral	Non-secure Peripheral
Secure	SUCCESS	FAULT
Non-secure	FAULT	SUCCESS

Upon a [PPU fault](#), the registers below in SMU notifies which PPU fault occurred and on which peripheral. If a fault is detected, the response is Read As Zero (RAZ) or Write Ignored (WI) and set the corresponding interrupt flag in the `SMU_IF` register. The values in `SMU_IF` and `SMU_PPUFS` do not change until all PPU faults in the `SMU_IF` register are cleared by software.

Register	Bitfield	Fault
SMU_IF	PPUPRIV	Privilege Fault if set
SMU_IF	PPUSEC	Security Fault if set
SMU_IF	PPUINST	Instruction Fault if set

Register	Bitfield	Fault
SMU_PPUFS	PPUFSPERIPHID	ID of the peripheral that caused the fault

## Compatibility

Secure software usually controls the SYSCFG and SMU peripherals to prevent Non-secure software from changing critical configurations in the Secure domain. It requires switching between Secure and Non-secure states when Non-secure software wants to update the registers in these peripherals. Therefore dedicated registers for Non-secure access are added to SYSCFG and SMU peripherals on newer Series 2 devices.

## System Configuration (SYSCFG)

Except for EFR32xG21 devices, the following tables apply to all Series 2 devices.

Table: Dedicated Bitfield to Configure Access for Non-secure SYSCFG Registers

Bitfield (Register)	Description
SYSCFGCFGNS (SMU_PPUPATD0)	Bitfields (privileged if set) for privileged access configuration on NS SYSCFG registers
SYSCFGCFGNS (SMU_PPUSATD0)	Bitfields (Secure if set) for security access configuration on NS SYSCFG registers

**Note:** Reset SYSCFGCFGNS bit in SMU\_PPUSATD0 to allow Non-secure software to access NS SYSCFG registers.

Table: Dedicated SYSCFG Registers for Non-secure State

SYSCFG Non-secure Registers	Description
SYSCFG_CFGNS_CFGNSTCALIB	NS SysTick calibration value register
SYSCFG_CFGNS_ROOTNSDATA0	NS root data register 0
SYSCFG_CFGNS_ROOTNSDATA0	NS root data register 1

## Security Management Unit (SMU)

Except for EFR32xG21 devices, the following tables apply to all Series 2 devices.

Table: Dedicated Bitfield to Configure Access for Non-secure SMU Registers

Bitfield (Register)	Description
SMUCFGNS (SMU_PPUPATD1)	Bitfields (privileged if set) for privileged access configuration on NS SMU registers
SMUCFGNS (SMU_PPUSATD1)	Bitfields (Secure if set) for security access configuration on NS registers

**Note:** Reset SMUCFGNS bit in SMU\_PPUSATD1 to allow Non-secure software to access NS SMU registers.

The SMU\_CFGNS register file is for the TrustZone Non-secure state and has its register lock ( NSLOCK ). It allows hardware to maintain the privileged assignments for the NS state. The privileged configuration within the NS state is the same as the Secure state, except it has an "NS" to differentiate the registers.

Table: Dedicated SMU Registers for Non-secure State

SMU Non-secure Registers	Description
SMU_CFGNS_NSSTATUS	Lock status of SMU_CFGNS registers
SMU_CFGNS_NSLCOK	Lock and unlock the SMU_CFGNS registers
SMU_CFGNS_NSIF	Interrupt flags for NS privilege (PPUNSPRIVIF) and instruction (PPUNSINSTIF) faults
SMU_CFGNS_NSIEN	Interrupt enable flags for NS privilege (PPUNSPRIVIEN) and instruction (PPUNSINSTIEN) faults
SMU_CFGNS_PPUNSPATD0	Bitfields (privileged if set) for NS privileged access configuration on peripheral group 0
SMU_CFGNS_PPUNSPATD1	Bitfields (privileged if set) for NS privileged access configuration on peripheral group 1
SMU_CFGNS_PPUNSFS	ID (PPUFSPERIPHID) of the NS peripheral that caused the fault
SMU_CFGNS_BMPUNSPATD0	Bitfields (privileged if set) for privileged attribute configuration on NS Bus Master group 0

Table: Fault Statuses Only for Secure State

Bitfield (Register)	Description
PPUPRIV (SMU_IF)	Fault status now limited only to Secure state
PPUINST (SMU_IF)	Fault status now limited only to Secure state
PPUPRIV (SMU_IEN)	Fault status now limited only to Secure state
PPUINST (SMU_IEN)	Fault status now limited only to Secure state
PPUFSPERIPHID (SMU_PPUFS)	Fault status now limited only to Secure state

Table Dedicated SMU Interrupt for Non-secure State

Interrupt	Description
SMU_NS_PRIVILEGED_IRQHandler()	An interrupt flag in the SMU_CFGNS_NSIF register can generate an NS privileged interrupt when its corresponding interrupt enable bit in the SMU_CFGNS_NSIEN register is set and SMU_NS_PRIVILEGED_IRQn is enabled, and in which the peripheral (ID) that triggers the fault is in the SMU_CFGNS_PPUNSFS register.

## Secure And Privileged Programming Model

# Secure and Privileged Programming Model

The implementation of BLS on Series 2 devices, both flash and RAM, use a programmable watermark to delineate Secure, Non-secure Callable, and Non-secure regions. On the other hand, peripherals exist in both a Secure and Non-secure alias of memory.

## BLS SMU Programming

### Enabling SMU Clock

Except for the EFR32xG21 devices, all Series 2 devices enable the SMU clock in CMU before programming the SMU registers.

```
#if (_SILICON_LABS_32B_SERIES_2_CONFIG > 1)
    CMU->CLKEN1_SET = CMU_CLKEN1_SMU;
#endif
```

### Cortex-M33 Lock Control

The Cortex-M33 security and privileged configurations can be locked by programming the `SMU_M33CTRL` register.

```
// Lock Secure MPU configuration
SMU->M33CTRL |= SMU_M33CTRL_LOCKSMPU;
```

### Locking SMU Configuration

The entire SMU configuration can be locked down to avoid runaway code. Below is an example of how to lock and unlock the SMU.

```
uint32_t lock_status;
// Lock Down SMU
SMU->LOCK = ~SMU_LOCK_SMULOCKKEY_UNLOCK;
// Grab Lock Status
lock_status = (SMU->STATUS & _SMU_STATUS_SMULOCK_MASK) >> _SMU_STATUS_SMULOCK_SHIFT;
// Unlock SMU
SMU->LOCK = SMU_LOCK_SMULOCKKEY_UNLOCK;
```

### Interrupt Control

Each interrupt flag in `SMU_IF` can generate an interrupt when its corresponding interrupt enable flag in the `SMU_IEN` register is set. Each interrupt flag can be cleared by writing the clear alias of the `SMU_IF` register.

```
// Clear and enable the SMU PPUSEC and BMPUSEC interrupt
NVIC_ClearPendingIRQ(SMU_SECURE_IRQn);
SMU->IF_CLR = SMU_IF_PPUSEC | SMU_IF_BMPUSEC;
NVIC_EnableIRQ(SMU_SECURE_IRQn);
SMU->IEN = SMU_IEN_PPUSEC | SMU_IEN_BMPUSEC;
```

## BLS ESAU Programming

### Region Types

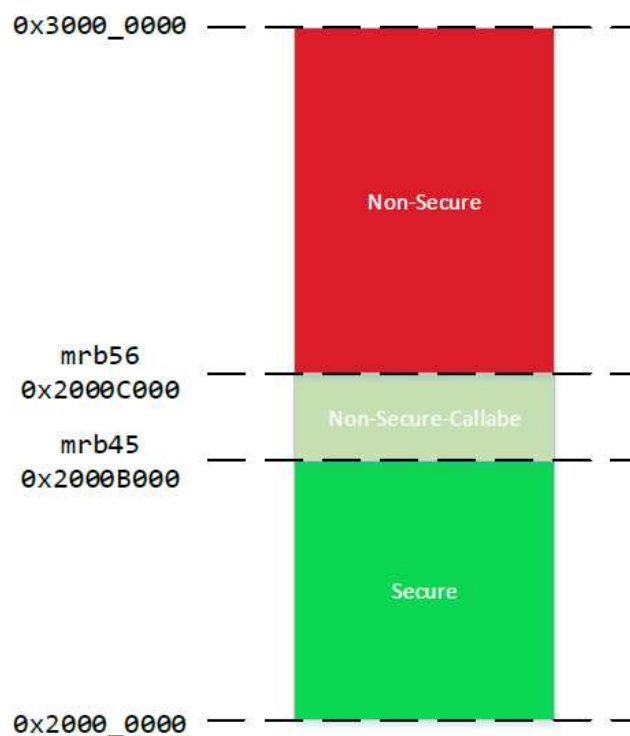
The `SMU_ESAURTYESn` registers are used to configure memory regions with a specific security attribute. All configurable memory regions reset to Secure. Below is an example of programming regions 3 and 11 to Non-secure.

```
// Region 3 (Info flash) is Non-secure
SMU->ESAURTYES0 = SMU_ESAURTYES0_ESAUR3NS;
// Region 11 (EPPB) is Non-secure
SMU->ESAURTYES1 = SMU_ESAURTYES1_ESAUR11NS;
```

### Region Sizes

The code and figure below highlight how to program the Movable Region Boundaries (MRBs) of ESAU.

```
// ESAU region 0/1/2 programming
// Boundary01 at 252kB and Boundary12 at 256kB
SMU->ESAUMRB01 = 0x0003F000U & _SMU_ESAUMRB01_MASK;
SMU->ESAUMRB12 = 0x00040000U & _SMU_ESAUMRB12_MASK;
// ESAU region 4/5/6 programming
// Boundary45 at 44kB and Boundary56 at 44kB (region 5 size = 0)
SMU->ESAUMRB45 = 0x0000B000U & _SMU_ESAUMRB45_MASK;
SMU->ESAUMRB56 = 0x0000B000U & _SMU_ESAUMRB56_MASK;
```





**Notes:**

- The `mr12` ( `ESAUMRB12` in `SMU_ESAURMBR12` ) has to be greater than or equal to `mr01` ( `ESAUMRB12` in `SMU_ESAURMBR12` ).
- The `mr56` ( `ESAUMRB56` in `SMU_ESAURMBR562` ) has to be greater than or equal to `mr45` ( `ESAUMRB45` in `SMU_ESAURMBR45` ).
- If one of the rules above is violated, the `SMU_STATUS.SMUPRGERR` is asserted.
- When `mr01` and `mr12` are equal, region 1 (NSC) is a size of 0 and is not seen by the system.
- When `mr45` and `mr56` are equal, region 5 (NSC) is a size of 0 and is not seen by the system.

## BLS SAU Programming

### All Secure Configuration

All secure configuration is the default state after reset. It clears the `SAU_CTRL.ENABLE` and the `SAU_CTRL.ALLNS` bits in SAU, and the entire memory is in a Secure attribute.

### All Non-secure Configuration

All Non-secure Configuration occurs when the `SAU_CTRL.ENABLE` bit is cleared, and the `SAU_CTRL.ALLNS` bit is set. The ESAU controls the security attribute of a Cortex-M33 transaction.

```
// Disable SAU (ALLNS = 1) and clear data and instruction pipe
SAU->CTRL = SAU_CTRL_ALLNS_Msk;
__DSB();
__ISB();
```

### Configurable Configuration

Configurable configuration occurs when the `SAU_CTRL.ENABLE` bit is set ( `SAU_CTRL.ALLNS` is irrelevant). Both SAU and ESAU determine the security attribute of a Cortex-M33 transaction. The code and figure below highlight how to program the SAU regions.

```
// Define all Non-secure (NS) and Non-secure Callable (NSC) Regions
#define REGION0_BASE 0x0001E000UL
#define REGION1_BASE 0x00020000UL
#define REGION2_BASE 0x20004000UL
#define REGION0_LIMIT 0x0001FFFFUL
#define REGION1_LIMIT 0x000FFFFFUL
#define REGION2_LIMIT 0x20017FFFUL
// CMSIS calls to enable SAU Regions
// SAU region 0 - Flash NSC at 120 kB to 128 kB (0x0001E000 - 0x0001FFFF)
SAU->RNR = (0UL & SAU_RNR_REGION_Msk);
SAU->RBAR = (REGION0_BASE & SAU_RBAR_BADDR_Msk);
SAU->RLAR = (REGION0_LIMIT & SAU_RLAR_LADDR_Msk) | SAU_RLAR_NSC_Msk | SAU_RLAR_ENABLE_Msk;
// SAU region 1 - Flash NS at 128 kB to 1024 kB (0x00020000 - 0x000FFFFF)
SAU->RNR = (1UL & SAU_RNR_REGION_Msk);
SAU->RBAR = (REGION1_BASE & SAU_RBAR_BADDR_Msk);
SAU->RLAR = (REGION1_LIMIT & SAU_RLAR_LADDR_Msk) | SAU_RLAR_ENABLE_Msk;
// SAU region 2 - RAM NS at 16 kB to 96 kB (0x20004000 - 0x20017FFF)
SAU->RNR = (2UL & SAU_RNR_REGION_Msk);
SAU->RBAR = (REGION2_BASE & SAU_RBAR_BADDR_Msk);
SAU->RLAR = (REGION2_LIMIT & SAU_RLAR_LADDR_Msk) | SAU_RLAR_ENABLE_Msk;
// CMSIS functions to enable SAU and clear data and instruction pipe
TZ_SAU_Enable();
__DSB();
__ISB();
```



## BLS B MPU Programming

### Bus Master Privileged Attribute

A Bus Master can be configured as either privileged (default) or unprivileged by programming the corresponding index in the `SMU_BMPUPATDn` register.

```
// Configure all odd Bus Masters unprivileged
for (i = 0; i < SMU_NUM_BMPUS; i++) {
    if (i & 0x01) {
        SMU->BMPUPATD0 &= ~(1 << i);
    }
}
```

### Bus Master Security Attribute

A Bus Master can be configured as either Secure (default) or Non-secure by programming the corresponding index in the `SMU_BMPUSATDn` register. Configure a Bus Master as Non-secure results in the Bus Master only being able to access Non-secure addresses.

```
// Configure all odd Bus Masters Non-secure
for (i = 0; i < SMU_NUM_BMPUS; i++) {
    if (i & 0x01) {
        SMU->BMPUSATD0 &= ~(1 << i);
    }
}
```

### Bus Master Fault Status

The Bus Master ID and the address that triggered the fault can be read from the `SMU_BMPUFS` and `SMU_BMPUFSADDR` registers.

```
uint32_t fs_bmpu_id;
uint32_t fs_bmpu_addr;
uint32_t fs_bmpu_secfault;
// Read Bus Master fault status
fs_bmpu_id = SMU->BMPUFS;
fs_bmpu_addr = SMU->BMPUFSADDR;
fs_bmpu_secfault = (SMU->IF & _SMU_IF_BMPUSEC_MASK) >> _SMU_IF_BMPUSEC_SHIFT;
// Clear the IF to capture a new fault
SMU->IF_CLR = SMU_IF_BMPUSEC;
```

## BLS PPU Programming

### Peripheral Privileged Attributes

A peripheral can be configured as either privileged (default) or unprivileged by programming the corresponding index in the `SMU_PPUPATDn` register.

```
// Configure all odd peripherals unprivileged
for (i = 0; i < SMU_NUM_PPU_PERIPHS; i++) {
    if (i & 0x01) {
        if (i >= 32) {
            SMU->PPUPATD1 &= ~(1 << (i-32));
        } else {
            SMU->PPUPATD0 &= ~(1 << i);
        }
    }
}
```

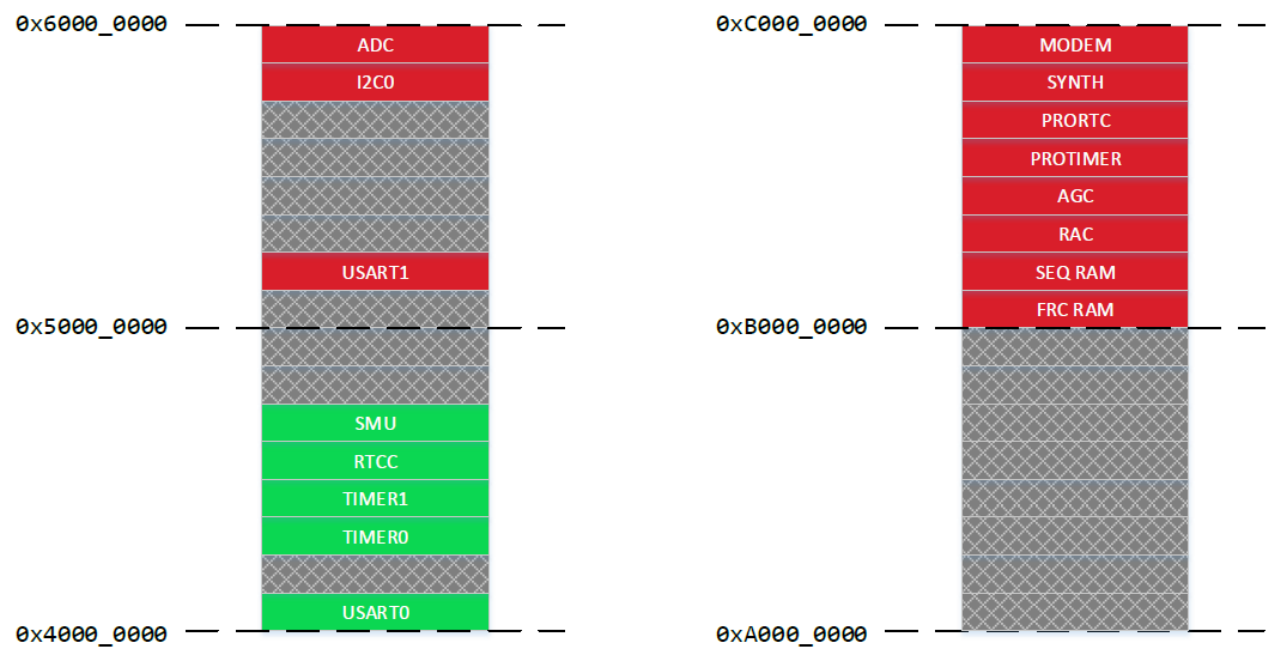
#### Notes:

- The peripherals in `SMU_PPUPATD0` and `SMU_PPUPATD1` are device-dependent.
- The privileged attribute of the radio subsystem (`AHBRADIO` index) is configured as a unit.

### Peripheral Security Attributes

A peripheral can be configured as either Secure (default) or Non-secure by programming the corresponding index in the `SMU_PPUSATDn` register. The figure below shows the memory map when the ADC, I2C0, USART1, and RADIO are configured as Non-secure and other peripherals (e.g., SMU, RTCC, TIMER1, TIMER0, USART0...) as Secure.

```
// Configure all the Non-secure peripherals
SMU->PPUSATD0 &= ~SMU_PPUSATD0_USART1;
SMU->PPUSATD1 &= ~(SMU_PPUSATD1_I2C0 | SMU_PPUSATD1_IADC0 | SMU_PPUSATD1_AHBRADIO);
```



- Notes:
- The peripherals in `SMU_PPUSATD0` and `SMU->PPUSATD1` are device-dependent.
  - The security attribute of the radio subsystem ( `AHBRADIO` index) is configured as a unit.

Peripheral Fault Status

The peripheral ID that triggered the fault can be read from the `SMU_PPUFS` register.

```
uint32_t fs_ppu_periph_id;
uint32_t fs_sec_fault;
uint32_t fs_priv_fault;
uint32_t fs_inst_fault;
// Read peripheral fault status
fs_ppu_periph_id = SMU->PPUFS;
fs_sec_fault = (SMU->IF & _SMU_IF_PPUSEC_MASK) >> _SMU_IF_PPUSEC_SHIFT;
fs_priv_fault = (SMU->IF & _SMU_IF_PPUPRIV_MASK) >> _SMU_IF_PPUPRIV_SHIFT;
fs_inst_fault = (SMU->IF & _SMU_IF_PPUINST_MASK) >> _SMU_IF_PPUINST_SHIFT;
// Clear the IF to capture a new fault
SMU->IF_CLR = SMU_IF_PPUSEC | SMU_IF_PPUPRIV | SMU_IF_PPUINST;
```

Floating Point Unit (FPU) Programming

If the Non-secure application enables the FPU at initialization, the Secure software needs to set up the `NSACR` register in `SCB` to grant the FPU access for Non-secure software.

```
// Enable Non-secure access to the FPU
SCB->NSACR |= SCB_NSACR_CP10_Msk + SCB_NSACR_CP11_Msk;
```

## TrustZone Implementation

# TrustZone Implementation

The goal of TrustZone implementation is to provide Secure Key Storage that can keep access to keys limited to Secure applications while at the same time allowing Non-secure applications to exercise the keys. It is an added feature for the SVM devices that do not have dedicated hardware for [Secure Key Storage](#) as in SVH devices.

The [PSA Crypto](#) is placed in a Secure region to keep key material hidden from the Non-secure application. The exposed PSA Crypto APIs stay the same while the backend provides persistent key encryption and decryption similar to the key wrapping and unwrapping functionality of the SVH device.

The following items need to be considered when upgrading the existing system for Secure Key Storage with TrustZone.

- [System Configuration](#)
- [Gecko Bootloader](#)
- [Secure Library](#)
- [TrustZone Secure Key Storage](#)
- [PSA Attestation](#)
- [SE Manager](#)
- [Common Vulnerabilities and Exposures \(CVE\)](#)

## System Configuration

The system configuration includes the following items:

- Enable system exceptions in the Secure state.
- Set the security attributes of different regions in the SAU and ESAU.
- Place peripherals and associated interrupts in either Secure or Non-secure applications.
- Assign the Bus Masters' security attributes.
- The system has two Secure/Non-secure pairs for the bootloader and application. The Secure part of each pair is responsible for properly configuring the split in its Secure application before branching to the Non-secure application.

**Note:** The secure application will issue a software reset at startup (fatal error) if the device's SE firmware version is lower than the [first version](#) that supports TrustZone.

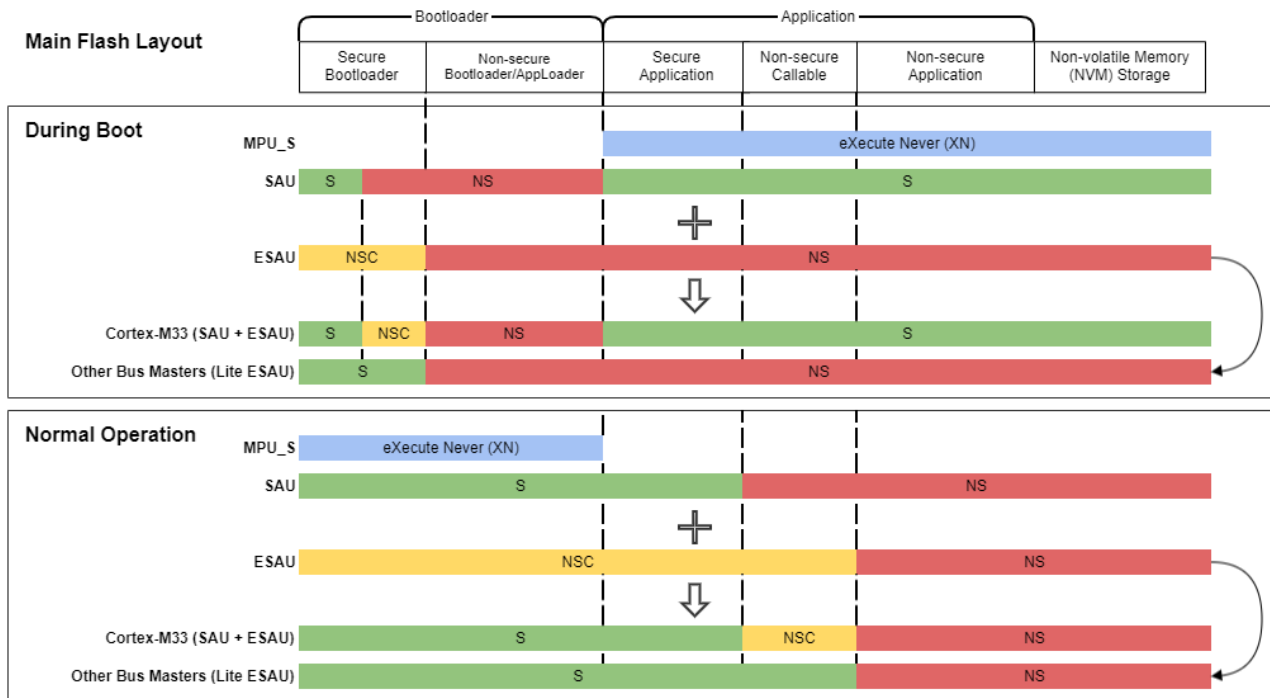
## System Exceptions

The following [system exceptions](#) are enabled in the Secure state for the bootloader and application.

- MemManage Fault
- BusFault
- UsageFault
- SecureFault

## Main Flash Layout

The following figure is an overview of the main flash layout that covers the isolation requirements for the Secure Key Storage solution. The SAU and ESAU configurations provide the required security to the Cortex-M33 and other Bus Masters during boot and normal operation.



#### 1. Settings:

- The application is set to non-executable (XN) by [Secure MPU](#) to avoid any code execution in this area during boot.
- The bootloader is set to non-executable (XN) by Secure MPU to avoid any code execution in this area during normal operation.

#### 2. The ESAU configuration only uses the NSC section by setting mr01 to the [base address of region 0](#). The reason is that lite ESAU in other Bus Masters treats both S and NSC as a Secure attribute. For the Cortex-M33, the SAU upgrades the NSC in the ESAU to Secure. The 32 bytes region alignment of SAU also relaxes the 4 kB alignment restriction on the start address of the NSC in ESAU.

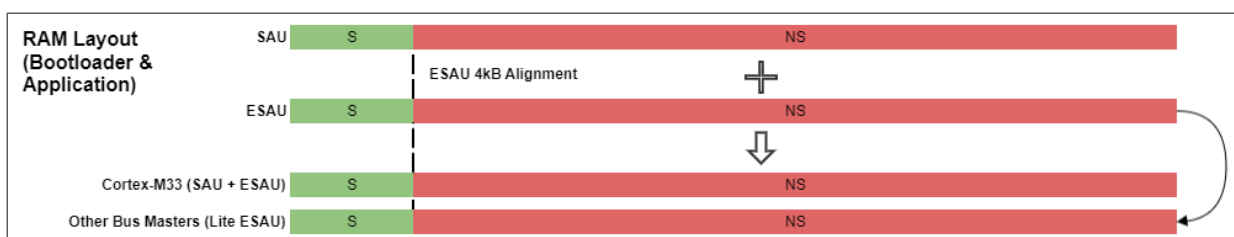
#### 3. The whole application area is set to Secure in SAU for Cortex-M33 during boot to hide details from the bootloader NS part.

#### 4. The ESAU cannot mark any region that comes after a Non-secure section as Secure (must be in the order of S/NSC/NS). Therefore the Secure application area does not align between the Cortex-M33 (SAU + ESAU) and other Bus Masters (lite ESAU) during boot. The secrets stored in that Secure region expose as Non-secure for other Bus Masters during boot (no such issue in normal operations). So the application must not save any plaintext secrets in that Secure region to overcome this limitation during boot.

#### 5. The NVM storage is in the Non-secure region, so the application must [encrypt](#) the persistent keys before storing them in this area.

## RAM Layout

The following figure is an overview of the RAM layout used for the bootloader and application. The SAU and ESAU are used to split the RAM into a Secure and Non-secure region (Non-secure Callable is not required).

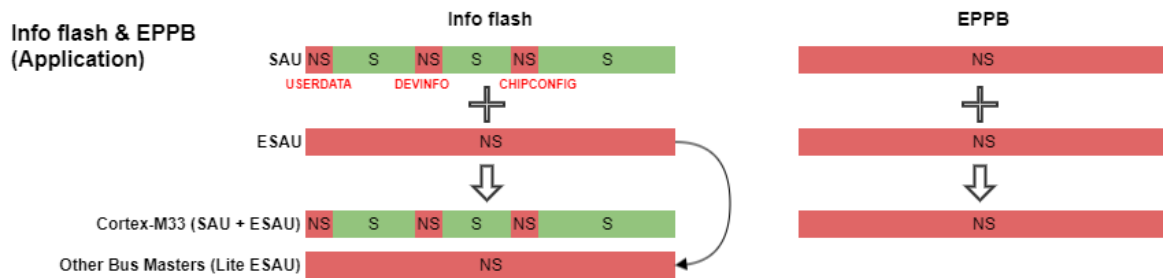


In practice, the Secure part (bootloader or application) takes ownership of the amount of RAM it needs from the beginning of RAM and leaves the rest (up to the ESAU 4 kB alignment requirement) configured as Non-secure.

The bootloader does not know how the application partitions the RAM between Secure and Non-secure. So bootloader removes any secrets from RAM before handing control to the application.

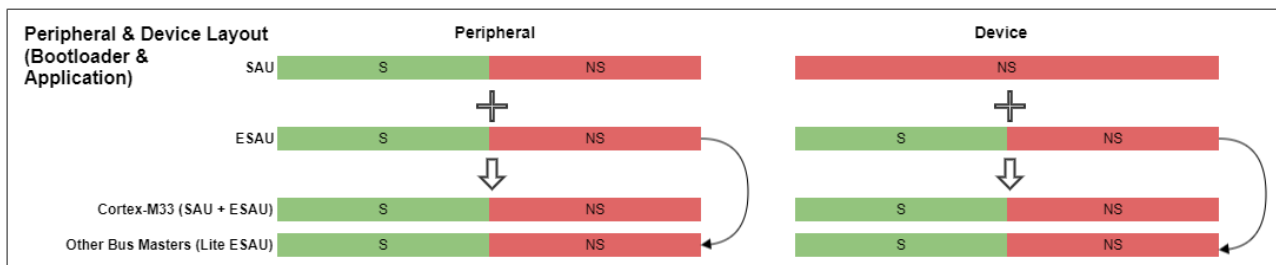
## Info Flash and EPPB

The following figure is an overview of the Info flash and EPPB layout for the application. The Cortex-M33 core is the only Bus Master that can access the EPPB region.



## Peripheral and Device

The following figure is an overview of the peripheral and device layout for the bootloader and application. The SAU and ESAU are used to split the peripheral and device into a Secure and Non-secure region.



The Secure software is responsible for moving all peripherals and associated interrupts to the Non-secure state at startup, except for the peripherals and interrupts that need to be Secure. The Non-secure software must not include code that attempts to directly access any peripheral that is used by the Secure software.

### Peripherals owned by the Secure software (application)

1. Security Management Unit (SMU)
  - It prevents Non-secure software from changing the configuration for the ESAUs, BMPUs, and PPUs.
  - Except for EFR32xG21 devices, some features are also available in the dedicated [Non-secure version of SMU registers](#) ( SMU\_CFGNS ).
2. CRYPTOACC (VSE devices) or SEMAILBOX (HSE devices)
  - The crypto engine is placed in the Secure domain for [Secure library](#).
3. System Configuration (SYSCFG)
  - It prevents Non-secure software from changing system configurations for Secure software.
  - Except for EFR32xG21 devices, some features are also available in the dedicated [Non-secure version of SYSCFG registers](#) ( SYSCFG\_CFGNS ).
4. Memory System Controller (MSC)
  -

It prevents Non-secure software from writing to Secure flash.

#### Peripheral interrupts owned by the Secure software:

**Table:** Secure Peripheral Interrupts (Application)

VSE Device	HSE Device
SMU_SECURE_IRQn	SMU_SECURE_IRQn
SYSCFG_IRQn	SYSCFG_IRQn
MSC_IRQn	MSC_IRQn
CRYPTOACC_IRQn	SEMBRX_IRQn
TRNG_IRQn	SEMBTX_IRQn
PKE_IRQn	

The `PRIS` bit in the `AIRCR` register is set to 1 to place all Non-secure exceptions/interrupts in [lower priority level space](#). Therefore any Secure exceptions/interrupts can be programmed with higher priority than Non-secure ones.

The `BMPUSEC` and `PPUSEC` interrupt enable flags in the `SMUIEN` register are set to enable the SMU security fault interrupts ( `SMU_SECURE_IRQn` ) on Bus Masters and peripherals.

#### Floating Point Unit (FPU):

The Secure application does not use the FPU. But the Secure startup code also enables the [FPU](#) for use by the Non-secure application.

## Bus Masters

To keep all secrets from the Non-secure world, only the Bus Masters in the table below can access data in the Secure world. For the Bus Masters living in the Secure world, the secure application must configure their corresponding control interfaces in the peripheral space to Secure. The Cortex-M33 core as a Bus Master is split to run in Secure and Non-secure contexts.

**Table:** Secure Bus Masters (Application)

Device	Secure Bus Master	Control Interface of Bus Master
VSE	CRYPTOACC	CRYPTOACC
HSE	SEDMA or SEEXTDMA	SEMAILBOX

#### Notes:

- Use `SMU_BMPUSATD0` register to [configure](#) the security attribute of a Bus Master.
- Use `SMU_PPUSATDn` register to [configure](#) the control interface of Bus Master as a Secure peripheral.
- LDMA is set as a Non-secure Bus Master to make sure it cannot be used to copy out data from the Secure memory.

## Application Transitions

The system contains two Secure/Non-secure pairs.

- The [bootloader pair](#bootloader pair) has a Secure bootloader and a Non-secure bootloader containing the communication interfaces.
- The [application pair](#application pair) has a Secure application and a Non-secure application consisting of the wireless stacks (if applicable) and application layers.

As described in the preceding sections, the Secure part of these pairs is responsible for setting the security configurations of the system during startup. For the handover between Secure/Non-secure pairs, the software must restore the system so the Secure part of the other pair can execute and reconfigure the system.

The software must reconfigure the following items before transitioning to the next Secure/Non-secure pair:

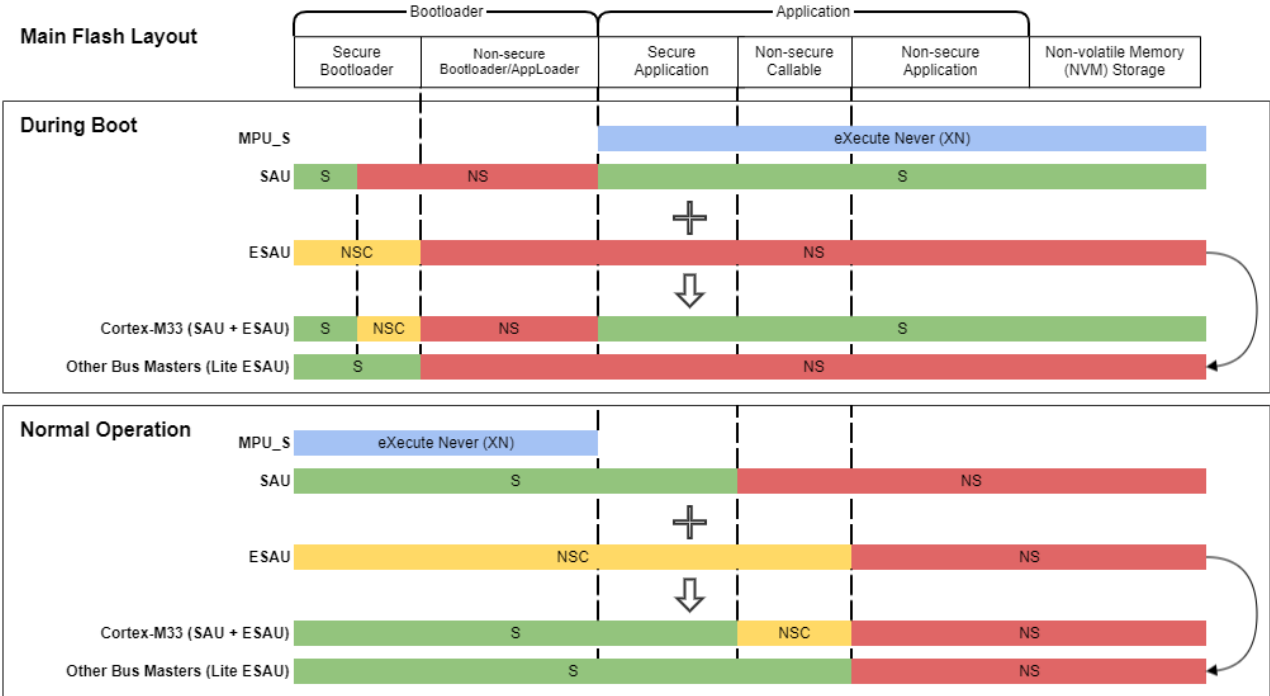
- Restored all peripherals and interrupts to Secure
- Reset ESAU to default configuration (all configurable regions to Secure)
- Reset SAU to default configuration (Secure for everything)



Reset MPU to default configuration (removes any XN)

## Gecko Bootloader

The [Gecko bootloader](#) ensures the Secure assets are protected during the boot flow and normal operation.



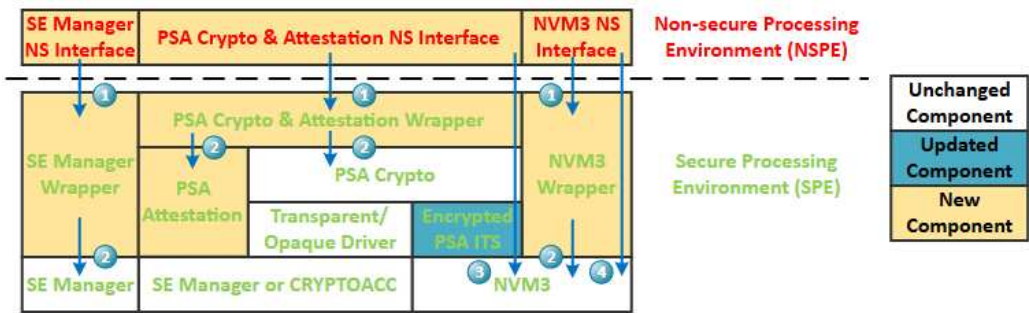
1. The SAU and [Secure MPU](#) mark all the flash for application and NVM as Secure and non-executable (XN) during boot. It guards against bootloader NS code execution branching into the application code.
2. The bootloader needs to split into Secure and Non-secure software to protect secrets in the system. Secure code can access the entire flash to validate or upgrade the system.
3. For VSE devices, the [GBL Decryption Key \(AES-128 key\)](#) is moved from the NS memory (last page of the main flash) to the Secure part of the bootloader. The Simplicity Commander v1.13 or higher provides a feature to inject the AES-128 key to the bootloader binary file.

```
commander convert <BL image file> --aeskey <decryption key file> --outfile <BL image with decryption key>
```

4. The bootloader communication interfaces are placed in the NS area to support various [communication components](#) below for firmware upgrades.
  - BGAPI UART
  - EZSP-SPI
  - UART XMODEM
5. The NS communication functions call into the [bootloader APIs](#) placed in the bootloader NSC region. The Secure application [validates](#) all input arguments before processing the request.
6. Before transiting from bootloader to normal operation, it resets the SAU to default configuration to make all the flash for bootloader as Secure.
7. The Non-secure application software can call [bootloader APIs](#) through application NSC, and the corresponding Secure function releases the non-executable (XN) restriction on the bootloader during normal operation.

## Secure Library

The goal of the Secure library is to keep the [PSA Crypto key](#) and [attestation token](#) protected from malicious code on the NSPE. The following figure overviews multiple components to support the Secure library.



1. The NS interfaces in NSPE are responsible for packing and passing all input arguments over the [NSC](#) functions on wrappers in SPE.
2. The wrappers in SPE validate all input arguments before calling into the corresponding APIs in different drivers.
3. Because of the system memory layout limitation, the [flash](#) for NVM3 storage is located in the NSPE. Therefore the updated PSA Internal Trusted Storage (ITS) driver needs to encrypt all crypto keys before storing them in Non-secure NVM.
4. Data stored directly using the NVM3 APIs are not encrypted.

The following table describes the new and updated components of the Secure library.

Component	Description
SE Manager NS interface	This component contains SE Manager API callable from the NSPE. It packages the list of input arguments in the appropriate format before calling into the SE Manager wrapper's NSC functions.
SE manager wrapper	This component contains the interface into SE Manager exposed to the NSPE. These NSC functions grant access to the SE Manager utility API and validate all input arguments before calling into SE Manager.
PSA Crypto & Attestation NS interface	This component contains PSA Crypto and attestation API callable from the NSPE. It packages the list of input arguments in the appropriate format before calling into the PSA Crypto and attestation wrapper's NSC functions.
PSA Crypto & Attestation wrapper	This component contains the interface into PSA Crypto and attestation exposed to the NSPE. These NSC functions grant access to the entire PSA Crypto and attestation API and validate all input arguments before calling into PSA Crypto and attestation.
PSA attestation	This component in SPE provides the functionality required by the PSA attestation specification.
Encrypted PSA ITS	The PSA ITS layer builds on top of NVM3. This component is updated to support encrypted storage to secure stored keys. The encryption is based on the device's TrustZone Root Key.
NVM3 NS interface	This component contains NVM3 API callable from the NSPE. It packages the list of input arguments in the appropriate format before calling into the NVM3 wrapper's NSC functions.
NVM3 wrapper	This component contains the interface into NVM3 exposed to the NSPE. These NSC functions grant access to the NVM3 API and validate all input arguments before calling into NVM3.

Notes:

- The SE Manager NS interface, PSA Crypto NS interface, and NVM3 NS interface in the NSPE provide drop-in replacement on [SE Manager utility](#), [PSA Crypto](#), and [NVM3](#) APIs for existing wireless stacks and user applications.
- The NSC calls can only take a limited number of arguments, so all NSC functions take a pointer to a list of parameters to support a long list of arguments. All arguments must be validated using the [intrinsic functions](#) from CMSIS.

TrustZone Secure Key Storage

The TrustZone Secure Key Storage provides a solution to store a user key in Secure RAM or an encrypted form in Non-secure flash.

The TrustZone Root Key stored in the SE NVM for Secure Key Storage encryption is generated or renewed by following operations.

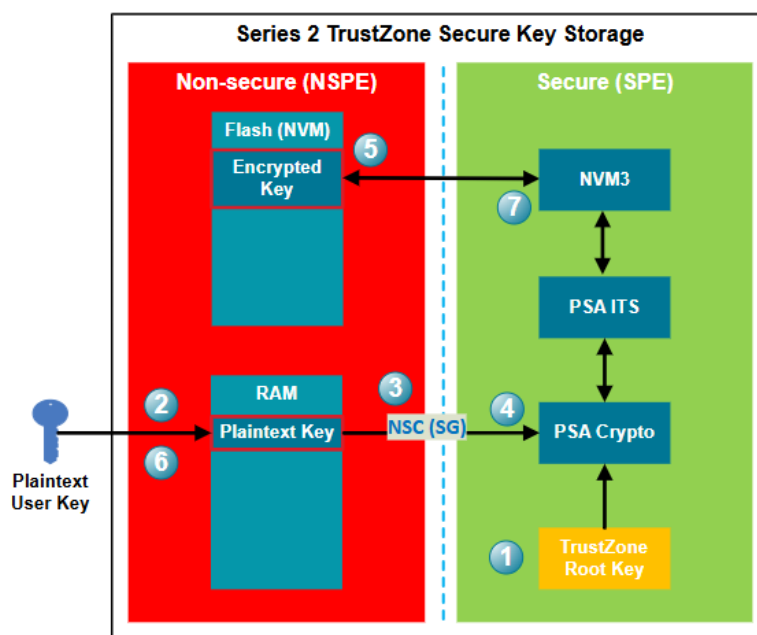
- The TrustZone Root Key had already existed if the shipped Series 2 device with [SE firmware version](#) supports this key.
- Generate a TrustZone Root Key when upgrading from a SE firmware version that did not support this key to the one that does.
- Renew a TrustZone Root Key after performing a [Device Erase](#).

**Note:** The TrustZone Root Key cannot be renewed if Device Erase is disabled.

The TrustZone Root Key is not exposable to the NSPE, and access to this key in SPE is different in HSE and VSE devices.

- HSE - The SPE can access the TrustZone Root Key as a built-in non-exportable key in HSE NVM.
- VSE - The SPE can access the TrustZone Root Key in Secure RAM, which is copied from VSE NVM during boot.

The TrustZone Root Key value after reset is identical to the value before reset. TrustZone Root Keys are unique on each device. The key allows a user to securely store a key in the Non-secure flash, limiting the number of keys that can be saved only by the amount of Non-secure storage. The following figure describes using the TrustZone Root Key to encrypt a plaintext key and store it in Non-secure NVM.



1. After power-on, the device's TrustZone Root Key is available for the SPE.
2. A user key is generated and imported into the device's Non-secure memory. In this example, the key is imported into Non-secure RAM for easy deletion, and the key is lost if device power is removed.
3. Call PSA Crypto API ( `psa_import_key()` or `psa_generate_key()` ) through SG in NSC to generate a key for crypto operations.
4. The plaintext key is passed to the PSA Crypto in SPE, where it is encrypted (AES-GCM) with the TrustZone Root Key.
5. The encrypted key is stored to the NVM in NSPE through the PSA ITS and NVM3 drivers.
6. The plaintext key can now be deleted from the Non-secure RAM.
7. Only the PSA Crypto in SPE can retrieve the encrypted key from NVM in NSPE and decrypt it for crypto operations in SPE.

**Note:** Ignore steps 2 and 6 if the plaintext key is randomly generated by the PSA Crypto.

The following tables describe the storage differences between SVM and SVH devices with and without TrustZone Secure Key Storage (SKS).

Key Type	Storage on SVM Device	Storage on SVH Device
Volatile Plaintext (without TrustZone SKS)	RAM	RAM
Persistent Plaintext (without TrustZone SKS)	NVM	NVM
Volatile Wrapped (without TrustZone SKS)	Not supported	RAM (1)
Persistent Wrapped (without TrustZone SKS)	Not supported	NVM (1)

Key Type	Storage on SVM Device	Storage on SVH Device
Volatile Plaintext (with TrustZone SKS)	Secure RAM (2)	Secure RAM
Persistent Plaintext (with TrustZone SKS)	Encrypted plaintext key in NS NVM (2)	Encrypted plaintext key in NS NVM
Volatile Wrapped (with TrustZone SKS)	Not supported	Secure RAM
Persistent Wrapped (with TrustZone SKS)	Not supported	Encrypted wrapped key in NS NVM

Notes:

- The NVM or [NS NVM](#) is at the last part of the main flash.
- It is possible to replace the [wrapped key](#) solution on the SVH device (1) with TrustZone Secure Key Storage on the SVM device (2), but this is a less secure approach.

## PSA Attestation

The device attestation service creates a token that contains a fixed set of device-specific data when requested by the caller. Each device must have a unique Initial Attestation Key (IAK) pair. The device uses the private IAK to sign the token, and the caller uses the public IAK to verify the token's authenticity.

The generation of the private IAK is different in SVM and SVH devices.

- SVM - If the private IAK does not exist in NVM3, it is randomly generated when requested from the [PSA Attestation](#) driver and saved to NVM3 through the [TrustZone Secure Key Storage](#).
- SVH - The private IAK is generated and securely stored in the HSE during chip production.

An Entity Attestation Token (EAT) is a mini-report that is cryptographically signed. An EAT is encoded in either one of two standardized data formats: a Concise Binary Object Representation ([CBOR](#)) or in the text-based format JSON. A digital signature is then used to protect its content. The technical specification defining the content of the EAT, which are claims about the hardware and the software running on a device, is specified by the Internet Engineering Task Force ([IETF](#)).

The EAT is a crypto-signed report card with claims. A claim is a data item that is represented as a Key-Value pair. Claims can relate to the device's pedigree or anything one wants the device to attest. Collected data can originate from the Root of Trust (RoT), any protected area, or non-protected areas.

The EAT must be signed following the structure of the CBOR Object Signing and Encryption ([COSE](#)) specification. For asymmetric key algorithms, the signature structure must be COSE-Sign1. A COSE-Sign1 is a CBOR encoded, self-secured data blob that contains headers, a payload, and a signature.

The primary need for EAT verification is to check correct formation and signing as for any token. In addition, though, the verifier can operate a policy where values of some of the claims in this profile can be compared to reference values that are registered with the verifier for a given deployment, to confirm that the device is endorsed by the manufacturer supply chain.

The [PSA attestation token](#) (aka Initial Attestation Token - IAT) is a profiled EAT. The Series 2 device will generate this token by (Nonce claim below) unless the SE OTP is uninitialized or the `SECURE_BOOT_ENABLE` option in SE OTP is disabled.

The following tables describe claims used in the PSA attestation token of the Series 2 device.

**Table:** Claims of PSA Attestation Token

Key	Claim Name (Present)	Claim Description	Claim Value
265 (-75000)	Profile Definition (Must)	The Profile Definition claim encodes the unique identifier corresponds to the EAT profile.	<a href="http://arm.com/psa/2.0.0">http://arm.com/psa/2.0.0</a>
2394 (-75001)	Client ID (Must)	The Client ID claim represents the security domain of the caller.	See note below (2 bytes)
2395 (-75002)	Security Lifecycle (Must)	The Security Lifecycle claim represents the current lifecycle state of the PSA RoT.	Device dependent (2 bytes)
2396 (-75003)	Implementation ID (Must)	The Implementation ID claim uniquely identifies the implementation of the immutable PSA RoT.	Device dependent (32 bytes)
2397 (-75004)	Boot Seed (Optional)	The Boot Seed claim represents a value created at system boot time that will allow differentiation of reports from different boot sessions.	Device dependent (32 bytes)
2399 (-75006)	Software Components (Must)	The Software Components claim is a list of software components that includes all the software loaded by the PSA RoT.	See note below
10 (-75008)	Nonce (Must)	The Nonce claim is used to carry the challenge provided by the caller to demonstrate freshness of the generated token. The length must be either 32, 48, or 64 bytes.	Random nonce (32/48/64 bytes)
256 (-75009)	Instance ID (Must)	The Instance ID claim represents the unique identifier of the IAK. The length must be 33 bytes.	SHA-256 hash of public IAK (32 bytes) with header 0x01

**Notes:**

- Some claims MUST be present in a PSA attestation token.
- The keys `-7500x` were defined in a previous version of the PSA attestation token specification ( `PSA_IOT_PROFILE1` profile) that is still used in the HSE-SVH firmware.
- The actual claims returned from the tokens on the SVH device are HSE firmware version-dependent.
- Key 2394: In PSA, a security domain is represented by a signed integer where negative values represent callers from the NSPE and positive IDs represent callers from the SPE. The value 0 is not permitted.
- Key 2395 (For the definitions of these lifecycle states, refer to the ARM [PSA Security Model](#)):
  - UNKNOWN ( `0x0000 - 0x00ff` )
  - ASSEMBLY\_AND\_TEST ( `0x1000 - 0x10ff` )
  - PSA\_ROT\_PROVISIONING ( `0x2000 - 0x20ff` )
  - SECURED ( `0x3000 - 0x30ff` )
  - NON\_PSA\_ROT\_DEBUG ( `0x4000 - 0x40ff` )
  - RECOVERABLE\_PSA\_ROT\_DEBUG ( `0x5000 - 0x50ff` )
  - DECOMMISSIONED ( `0x6000 - 0x60ff` )
- Key 2396:
  - Word[0]: Die revision
  - Word[1]: SE OTP version (return 0 for VSE SE firmware < [v1.2.14](#))
  - Word[2]: Security capability (not applicable to HSE-SVH device, always returns 1 in this word)
    - 0: Unknown security capability
    - 1: Security capability not applicable

- 2: Basic security capability
  - 3: Root of Trust security capability
  - 4: HSE-SVM security capability
  - 5: HSE-SVH security capability (run HSE-SVM binary on HSE-SVH device)
- Word[3]: Production version
- Word[4:7]: Reserved (zeros)
- Key 2399: Each software component uses the attributes described in the following table, and some MUST be present in a software component claim.

Key	Attribute (Present)	Description	Value
1	Measurement Type (Optional)	The Measurement Type attribute is a short string representing the role of this software component.	See note below
2	Measurement Value (Must)	The Measurement Value attribute represents a hash of the invariant software component in memory at startup time.	SHA-256 hash (32 bytes) of the firmware
4	Version (Optional)	The Version attribute is the issued software version in the form of a text string.	A string of 8 bytes

The following measurement types may be used for Key 1:

- "BL": a Bootloader
- "PRoT": a component of the PSA Root of Trust
- "ARoT": a component of the Application Root of Trust
- "App": a component of the NSPE application
- "TS": a component of a Trusted Subsystem

The PSA Attestation API allows access to the PSA attestation token, so an external entity can cryptographically verify the identity and trust status of the device.

Table: PSA Attestation API

PSA Attestation API	Usage
psa_initial_attest_get_token	Retrieve the PSA attestation Token.
psa_initial_attest_get_token_size	Calculate the size of a PSA attestation Token.
sl_tz_attestation_get_public_key	Get the public IAK key for PSA attestation token signature verification.

**Note:** The `sl_tz_attestation_get_public_key` is a Silicon Labs custom API.

## SE Manager

SE Manager is the foundation for the [Secure library](#) cryptographic operations on HSE devices. It means that SE Manager has to move into the SPE.

The following SE Manager [core](#) APIs are always available in the NSPE.

SE Manager Core API	VSE-SVM	HSE-SVM	HSE-SVH
sl_se_init	Y	Y	Y
sl_se_deinit	Y	Y	Y
sl_se_init_command_context	Y	Y	Y
sl_se_deinit_command_context	Y	Y	Y
sl_se_set_yield	Y	Y	Y

The following SE Manager [core](#) APIs expose to the NSPE through the NSC interface for the VSE devices.

SE Manager Core API	VSE-SVM	HSE-SVM	HSE-SVH
sl_se_read_executed_command	Y	-	-
sl_se_ack_command	Y	-	-

The following SE Manager [utility](#) APIs expose to the NSPE through the NSC interface for configuring the security features of HSE or VSE devices.

SE Manager Utility API	VSE-SVM	HSE-SVM	HSE-SVH
sl_se_check_se_image	Y	Y	Y
sl_se_apply_se_image	Y	Y	Y
sl_se_get_upgrade_status_se_image	Y	Y	Y
sl_se_check_host_image	Y	Y	Y
sl_se_apply_host_image	Y	Y	Y
sl_se_get_upgrade_status_host_image	Y	Y	Y
sl_se_init_otp_key	Y	Y	Y
sl_se_read_pubkey	Y	Y	Y
sl_se_init_otp	Y	Y	Y
sl_se_read_otp	Y	Y	Y
sl_se_get_se_version	Y	Y	Y
sl_se_get_debug_lock_status	Y	Y	Y
sl_se_apply_debug_lock	Y	Y	Y
sl_se_get_otp_version	Y	Y	Y
sl_se_write_user_data	-	Y (EFR32xG21 only)	Y (EFR32xG21 only)
sl_se_erase_user_data	-	Y (EFR32xG21 only)	Y (EFR32xG21 only)
sl_se_get_reset_cause	-	Y (EFR32xG21 only)	Y (EFR32xG21 only)
sl_se_get_status	-	Y	Y
sl_se_get_serialnumber	-	Y	Y
sl_se_enable_secure_debug	-	Y	Y
sl_se_disable_secure_debug	-	Y	Y

SE Manager Utility API	VSE-SVM	HSE-SVM	HSE-SVH
sl_se_set_debug_options	-	Y	Y
sl_se_erase_device	-	Y	Y
sl_se_disable_device_erase	-	Y	Y
sl_se_get_challenge	-	Y	Y
sl_se_roll_challenge	-	Y	Y
sl_se_open_debug	-	Y	Y
sl_se_disable_tamper	-	-	Y
sl_se_read_cert_size	-	-	Y
sl_se_read_cert	-	-	Y

**Note:** The NSPE cannot access the other [SE Manager APIs](#) for cryptographic and attestation operations.

## Common Vulnerabilities and Exposures (CVE)

At this writing, the following known TrustZone CVE had been fixed in the current implementation.

- [CVE-2020-16273](#): Stack sealing
- [CVE-2021-36465](#): VLLDM instruction/floating-point vulnerability



## Upgrade Existing Application To TrustZone

# Upgrade Existing Application to TrustZone

The main concerns when upgrading existing deployment to the TrustZone solution are:

- The [Secure/Non-secure pair for the bootloader](#) (24 kB) does not fit inside the current allotted bootloader space (16 kB).
- The [Secure/Non-secure pair for the application](#) does not fit inside the current allotted application space.
- The [PSA ITS](#) moves from a non-encrypted to an encrypted format, so the existing stored cryptographic keys in NVM3 cannot be reused after upgrading the current application to TrustZone.

The [Secure Library](#) is based on PSA Crypto, so the existing application cannot integrate with the TrustZone if one of the following conditions is valid.

- Use [SE Manager APIs](#) for cryptographic and attestation operations.
- Use classic Mbed TLS APIs for cryptographic operations (except for X.509 certificate) and Transport Layer Security (TLS) protocol.

## System Requirements

The following table lists the tools and software required for TrustZone development on Series 2 devices.

Tool/Software	Required Version	Description
GCC	v10.3.1	Fix a bug (ID 99271) on cmse_nonsecure_call attribute.
IAR EWARM	v9.20.4	Fix a bug (EWARM-9484) on __cmse_nonsecure_call attribute.
Segger J-Link	≥ v7.6.2c	v7.6.2c is the first version to add basic TrustZone support on Series 2 devices.
Simplicity Studio	≥ v5.6.3.0	v5.6.3.0 is the first version to support TrustZone software development on Series 2 devices.
Simplicity Commander	≥ v1.13.3	v1.13.3 includes a TrustZone-aware flash loader and supports features required for TrustZone development.
GSDK	≥ v4.2.2	GSDK v4.2.2 is the first version to support TrustZone software development on Series 2 devices.
SE Firmware	≥ v1.2.14	v1.2.14 is the first version to fully support TrustZone on xG21 (HSE) and xG22 (VSE) devices.
SE Firmware	≥ v2.2.1	v2.2.1 is the first version to fully support TrustZone on other Series 2 HSE and VSE devices.

### Notes:

- Required GCC and IAR EWARM versions are GSDK-dependent.
- [Bug list of GCC v10.3](#)
- [IAR EWARM release note](#)
- [Segger J-Link release note](#)
- [Simplicity Studio user guide](#)
- [Latest version of Simplicity Commander](#)
- [GSDK release note](#)
- Silicon Labs strongly recommends installing the latest SE firmware on Series 2 devices to support the required TrustZone features. The latest SE firmware image and release notes after installing the GSDK (Windows folder):  
C:\Users<PC USER NAME>\SimplicityStudio\SDKs\gecko\_sdk\util\se\_release\public

## Peripheral Addresses in Device Header Files

The device header files (e.g., efr32mg21b020f1024im32.h) need to be configurable for different situations. The `SL_TRUSTZONE_SECURE` and `SL_TRUSTZONE_NONSECURE` definitions specify whether the compilation is for Secure or Non-secure applications. The `SL_TRUSTZONE_SECURE` and `SL_TRUSTZONE_NONSECURE` should be exclusive. If none of the definitions are true, the state should be similar to the Non-secure configuration, but the [startup code](#) (`SystemInit()` in `system_*.c`) will be responsible for reconfiguring the system.

Define (Software Component)	Default Peripheral Pointer	Startup Code
<code>SL_TRUSTZONE_SECURE</code> (TrustZone Secure)	Point to Secure peripherals ( <code>*_BASE = *_S_BASE</code> )	No effect on <code>SystemInit()</code>
<code>SL_TRUSTZONE_NONSECURE</code> (TrustZone Non-Secure)	Point to Non-secure peripherals ( <code>*_BASE = *_NS_BASE</code> )	No effect on <code>SystemInit()</code>
None of the above (-)	Point to Non-secure peripherals ( <code>*_BASE = *_NS_BASE</code> )	<code>SystemInit()</code> moves peripherals to Non-secure

When building a Secure application (`SL_TRUSTZONE_SECURE` is true), all peripherals shall have their non-suffixed default address pointing to the Secure location of the peripheral (e.g., EMU). But the definitions in `sl_trustzone_secure_config.h` can force the addresses of specific peripherals pointing to the Non-secure location.

```
#ifndef SL_TRUSTZONE_SECURE_CONFIG_H
#define SL_TRUSTZONE_SECURE_CONFIG_H
// Specify security configuration of peripherals. Peripherals that are not
// included here will automatically have their _BASE addresses point to their
// secure address. This might not be true, since most peripherals are configured
// to be non-secure -- but it's also not a problem if the peripheral is not
// accessed from the S app.
// Used in multiple places.
#define SL_TRUSTZONE_PERIPHERAL_CMU_S (0)
// Used by SE Manager service.
#define SL_TRUSTZONE_PERIPHERAL_AHBRADIO_S (0)
// Used by MSC service.
#define SL_TRUSTZONE_PERIPHERAL_LDMA_S (1)
// Used by MSC service.
#define SL_TRUSTZONE_PERIPHERAL_LDMAXBAR_S (1)
#endif // SL_TRUSTZONE_SECURE_CONFIG_H

#if defined(SL_CATALOG_TRUSTZONE_SECURE_CONFIG_PRESENT)
#include "sl_trustzone_secure_config.h"
#endif
#if ((defined(SL_TRUSTZONE_SECURE) && !defined(SL_TRUSTZONE_PERIPHERAL_EMU_S))
    || (defined(SL_TRUSTZONE_PERIPHERAL_EMU_S) && (SL_TRUSTZONE_PERIPHERAL_EMU_S != 0)))
#define EMU_BASE      (EMU_S_BASE)      /* EMU base address */
#else
```

In other cases (`SL_TRUSTZONE_NONSECURE` is true or both `SL_TRUSTZONE_SECURE` and `SL_TRUSTZONE_NONSECURE` are false), all peripherals shall have their non-suffixed default address pointing to the Non-secure location of the peripheral (e.g., EMU).

```
#define EMU_BASE      (EMU_NS_BASE)      /* EMU base address */
```

**Note:** Do not install the **TrustZone Secure** or **TrustZone Non-Secure** software component to the [TrustZone-unaware](#) application.

Startup Code

The startup code moves peripherals from Secure to Non-secure to support the [default peripheral locations](#). In a TrustZone-aware application (either `SL_TRUSTZONE_SECURE` or `SL_TRUSTZONE_NONSECURE` is true), this is the application's responsibility (skip lines 168 to 194 in `SystemInit()`) and is done in the [Secure firmware](#) of the system.

For the TrustZone-unaware application (both `SL_TRUSTZONE_SECURE` and `SL_TRUSTZONE_NONSECURE` are false), the `SystemInit()` in `system_*.c` (e.g., `system_efr32mg21.c`) moves peripherals to the Non-secure location.

- The `SystemInit()` sets the accesses of all peripherals to Non-secure except for the [SMU](#) and HSE SEMAIBOX (lines 172 to 178).
- The `SystemInit()` sets the SAU in [All Non-secure](#) configuration (lines 180 to 187).
  - It ensures Non-secure access to Non-secure peripherals.
  - The device component files (e.g., `efr32mg21b020f1024im32.slc`) enable the [CMSE](#) compiler option ( `-mcmse` for GCC and `--cmse` for IAR) to pass the condition in line 181 to program the SAU.
  - To catch the missing CMSE compiler option, it will generate a preprocessor error (line 186) if the CMSE flag is not set when manually upgrading a project from GSDK v4.0.x to ≥v4.1.x for the TrustZone-unaware application.
- The `SystemInit()` does not program the [ESAU](#) (default Secure flash is 32 MB), so the whole program is run in the **Secure** state.
- The `SystemInit()` also enables the `BMPUSEC` and `PPUSEC` interrupts in the SMU (lines 189 to 193). It ensures the TrustZone-unaware application catches any violations of Bus Master and peripheral security access permissions.

```

144 void SystemInit(void)
145 {
146 #if defined ( VTOR_PRESENT) && ( VTOR_PRESENT == 1U)
149
150 #if defined(UNALIGNED_SUPPORT_DISABLE)
153
154 #if ( FPU_PRESENT == 1) && ( FPU_USED == 1)
158
159 /* Secure app takes care of moving between the security states.
160  * SL_TRUSTZONE_SECURE MACRO is for secure access.
161  * SL_TRUSTZONE_NONSECURE MACRO is for non-secure access.
162  * When both the MACROS are not defined, during start-up below code makes sure
163  * that all the peripherals are accessed from non-secure address except SMU,
164  * as SMU is used to configure the trustzone state of the system. */
165 #if !defined(SL_TRUSTZONE_SECURE) && !defined(SL_TRUSTZONE_NONSECURE) \
166     && defined(__TZ_PRESENT)
167
168 #if ( _SILICON_LABS_32B_SERIES_2_CONFIG >= 2)
169     CMU->CLKEN1_SET = CMU_CLKEN1_SMU;
170 #endif
171
172     /* config SMU to Secure and other peripherals to Non-Secure. */
173     SMU->PPUSATD0_CLR = _SMU_PPUSATD0_MASK;
174 #if defined (SEMAILBOX_PRESENT)
175     SMU->PPUSATD1_CLR = ( _SMU_PPUSATD1_MASK & (~SMU_PPUSATD1_SMU & ~SMU_PPUSATD1_SEMAIBOX) );
176 #else
177     SMU->PPUSATD1_CLR = ( _SMU_PPUSATD1_MASK & ~SMU_PPUSATD1_SMU );
178 #endif
179
180     /* SAU treats all accesses as non-secure */
181 #if defined(__ARM_FEATURE_CMSE) && (__ARM_FEATURE_CMSE == 3U)
182     SAU->CTRL = SAU_CTRL_ALLNS_Msk;
183     __DSB();
184     __ISB();
185 #else
186     #error "The startup code requires access to the CMSE toolchain extension to set proper SAU settings."
187 #endif /* __ARM_FEATURE_CMSE */
188
189     /* Clear and Enable the SMU PPUSEC and BMPUSEC interrupt. */
190     NVIC_ClearPendingIRQ(SMU_SECURE_IRQn);
191     SMU->IF_CLR = SMU_IF_PPUSEC | SMU_IF_BMPUSEC;
192     NVIC_EnableIRQ(SMU_SECURE_IRQn);
193     SMU->IEN = SMU_IEN_PPUSEC | SMU_IEN_BMPUSEC;
194 #endif /*SL_TRUSTZONE_SECURE */
195 }

```

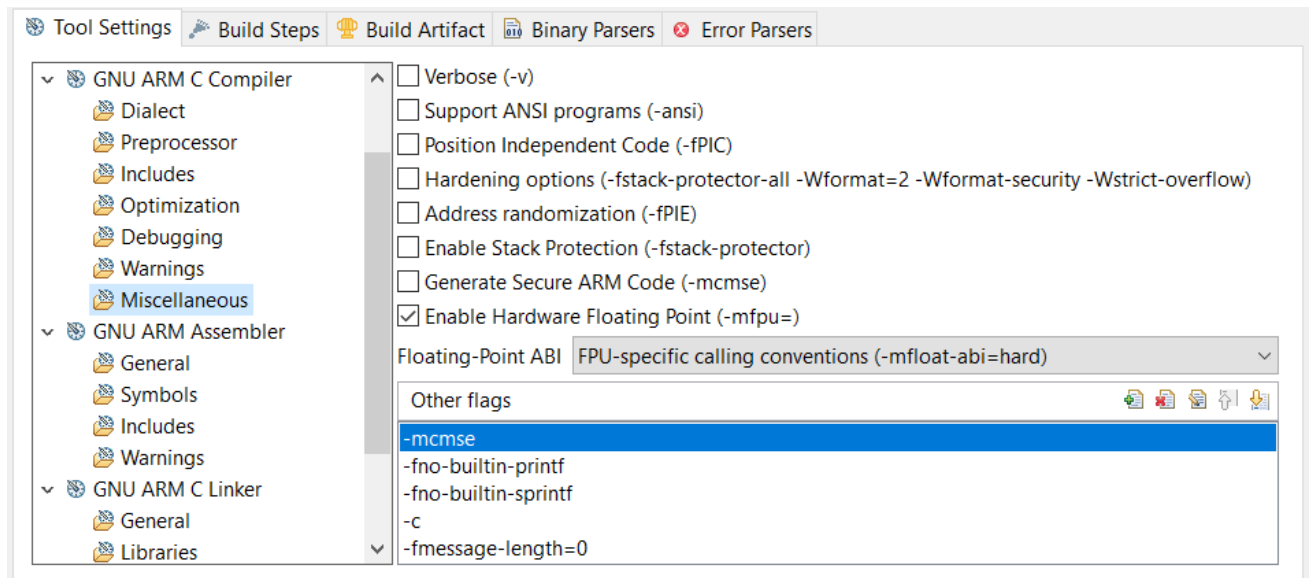
The `SMU_BASE` and HSE `SEMAIBOX_HOST_BASE` in device header files must point to the Secure location regardless of the `SL_TRUSTZONE_SECURE` and `SL_TRUSTZONE_NONSECURE` settings to avoid security violations on peripherals in the TrustZone-unaware application (SMU and HSE SEMAIBOX are set to Secure peripherals).

```
#if ((defined(SL_TRUSTZONE_SECURE) && !defined(SL_TRUSTZONE_PERIPHERAL_SMU_S))
|| (defined(SL_TRUSTZONE_PERIPHERAL_SMU_S) && (SL_TRUSTZONE_PERIPHERAL_SMU_S != 0)))
#define SMU_BASE      (SMU_S_BASE)      /* SMU base address */
#else
#define SMU_BASE      (SMU_S_BASE)      /* SMU base address */

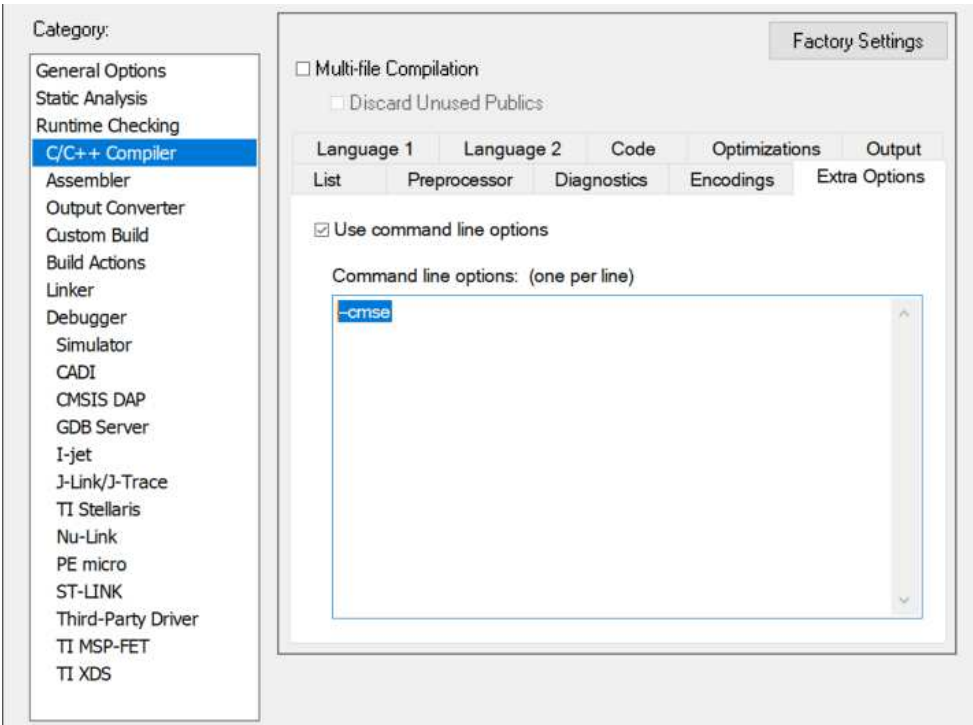
#if ((defined(SL_TRUSTZONE_SECURE) && !defined(SL_TRUSTZONE_PERIPHERAL_SEMAILBOX_HOST_S))
|| (defined(SL_TRUSTZONE_PERIPHERAL_SEMAILBOX_HOST_S) && (SL_TRUSTZONE_PERIPHERAL_SEMAILBOX_HOST_S != 0)))
#define SEMAILBOX_HOST_BASE  (SEMAILBOX_S_HOST_BASE)  /* SEMAILBOX_HOST base address */
#else
#define SEMAILBOX_HOST_BASE  (SEMAILBOX_S_HOST_BASE)  /* SEMAILBOX_HOST base address */
```

## Notes:

- The CMSE compiler option of GCC is in the Other flags window under C/C++ Build → Settings → Tool Settings → GNU ARM C Compiler → Miscellaneous.



- The CMSE compiler option of IAR is in the Command line options: (one per line) window under Options... → C/C++ Compiler → Extra Options.



Linker File

The `template_contribution` defined in the `slcp` files of [Secure and Non-secure projects](#) will override the default memory settings defined in the device component files (e.g., `efr32mg21b020f1024im32.slcc`) to generate the linker files for [Secure](#) and [Non-secure](#) applications.

Memory Region	Default Setting in Device Component File	Override Setting in <code>template_contribution</code>
Flash start address	<code>device_flash_addr</code>	<code>memory_flash_start</code>
Flash size	<code>device_flash_size</code>	<code>memory_flash_size</code>
RAM start address	<code>device_ram_addr</code>	<code>memory_ram_start</code>
RAM size	<code>device_ram_size</code>	<code>memory_ram_size</code>

Default setting (`efr32mg21b020f1024im32.slcc`)

```
- name: device_family
  value: efr32mg21
- name: device_flash_addr
  value: 0
- name: device_flash_size
  value: 1048576
- name: device_flash_page_size
  value: 8192
- name: device_ram_addr
  value: 536870912
- name: device_ram_size
  value: 98304
```

`template_contribution` example (Secure)

```
template_contribution:
- name: memory_flash_start
  value: 0x0
- name: memory_flash_size
  value: 0x2C000
- name: memory_ram_start
  value: 0x20000000
- name: memory_ram_size
  value: 0x3000
```

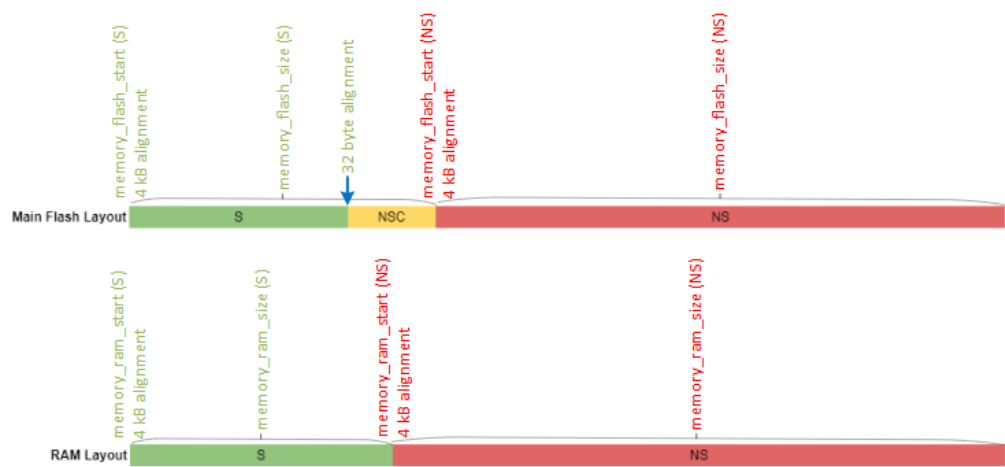
`template_contribution` example (Non-secure)

```
template_contribution:
- name: memory_flash_start
  value: 0x2C000
- name: memory_flash_size
  value: 0x54000
- name: memory_ram_start
  value: 0x20003000
- name: memory_ram_size
  value: 0x5000
```

The [ESAU](#) sets the flash and RAM start address, so these addresses should be alignment at **4 kB** (0x1000). The Secure project linker file needs to have a section for [NSC](#) (Secure Gateway) at the end of the Secure flash section. The [SAU](#) sets the start address of the NSC section, so this section only needs to be **32 bytes** aligned.

- GCC NSC: The `.gnu.sgstubs` region in the Secure application map file (.map)
- IAR NSC: The `Veneer$$CMSE` region in the Secure application map file (.map)

The Secure and Non-secure flash and RAM sizes are incremented or decremented in **4 kB**. The memory configurations in Secure and Non-secure applications are correlated, so the flash and RAM settings are in pairs.



**Note:** Users should not directly edit the `template_contribution` in the `slcp` file, but rather use the [Memory Editor](#) in Simplicity Studio to update the memory configuration.

## Debugger

- Simplicity Studio supports two [debuggers](#):
- GNU Debugger (GDB) client and SEGGER's GDB server
  - Simplicity Studio Debugger

The [TrustZone-unaware](#) and [TrustZone-aware](#) applications enable the `PPUSEC` interrupts in the SMU. The debugger will trigger the `SMU_SECURE_IRQHandler` if the **[Registers]** or **[Peripherals]** view feature violates peripheral security access permission.

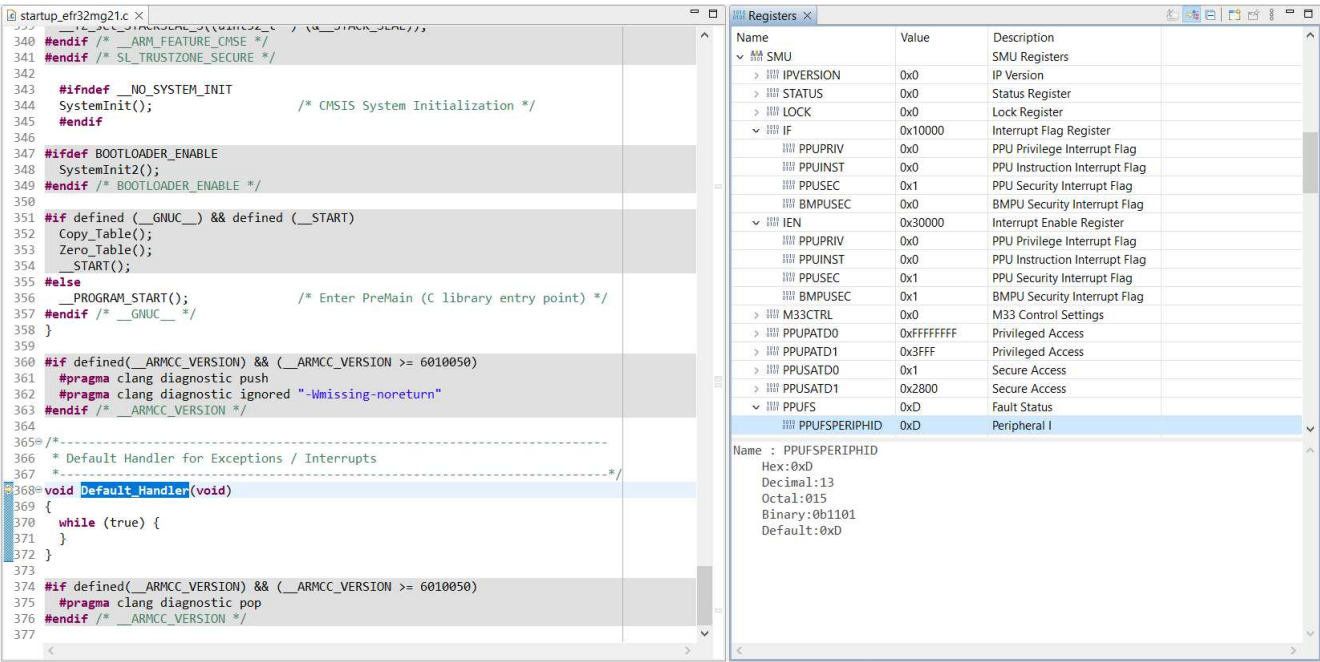
### Simplicity Studio Debugger

The **[Registers]** view of Simplicity Studio Debugger can only access the Secure location of a peripheral. The following figure demonstrates the `Default_Handler` (`SMU_SECURE_IRQHandler` not defined) is triggered (`PPUSEC` in `SMU->IF` = 1) when viewing the registers of GPIO peripheral (`PPUFSPERIPHID` = 13) that is set to Non-secure access in the SMU.

The debugger can access the registers of the SMU since this peripheral is set to Secure access in the SMU.

This limitation does not apply to **GSDK < v4.1.0** since no peripherals are configured for Non-secure access.





The Simplicity Studio Debugger is not the preferred choice for TrustZone debugging since it has limitations on viewing Non-secure access peripherals.

GNU Debugger (GDB)

The [Peripherals] view of GNU Debugger can access either the Secure or Non-secure location of the peripheral to avoid conflicts on security access permission. The following figure shows the registers of GPIO on Secure ( GPIO at 0x4003C000 ) and Non-secure ( GPIO\_NS at 0x5003C000 ) addresses. The GPIO peripheral is set to Non-secure access in the SMU, so the registers in the Secure address are displayed as zero.

Peripherals

Peripheral	Address	Description
<input type="checkbox"/> CMU_NS	0x50008000	CMU Registers
<input type="checkbox"/> DPLL0	0x4001C000	DPLL Registers
<input type="checkbox"/> DPLL0_NS	0x5001C000	DPLL Registers
<input type="checkbox"/> EMU	0x40004000	EMU Registers
<input type="checkbox"/> EMU_NS	0x50004000	EMU Registers
<input type="checkbox"/> FRC	0xA8004000	FRC Registers
<input type="checkbox"/> FRC_NS	0xB8004000	FRC Registers
<input type="checkbox"/> FSRCO	0x40018000	FSRCO Registers
<input type="checkbox"/> FSRCO_NS	0x50018000	FSRCO Registers
<input type="checkbox"/> GPCRC	0x40088000	GPCRC Registers
<input type="checkbox"/> GPCRC_NS	0x50088000	GPCRC Registers
<input checked="" type="checkbox"/> GPIO	0x4003C000	GPIO Registers
<input type="checkbox"/> GPIO_NS	0x5003C000	GPIO Registers
<input type="checkbox"/> HFRCO0	0x40010000	HFRCO Registers
<input type="checkbox"/> HFRCO0_NS	0x50010000	HFRCO Registers
<input type="checkbox"/> HFRCOEM23	0x4A014000	HFRCO Registers
<input type="checkbox"/> HFRCOEM23_NS	0x5A014000	HFRCO Registers
<input type="checkbox"/> HFX00	0x4000C000	SYXO Registers
<input type="checkbox"/> HFX00_NS	0x5000C000	SYXO Registers
<input type="checkbox"/> I2C0	0x4A010000	I2C Registers
<input type="checkbox"/> I2C0_NS	0x5A010000	I2C Registers

No details to display for the current selection.

GPIO: 0x4003C000

Register	Address	Value
GPIO	0x4003C000	
PORTA_CTRL	0x4003C000	0x00000000
SLEWRATE	[6:4]	0x0
DINDIS	[12]	0x0
SLEWRATEALT	[22:20]	0x0
DINDISALT	[28]	0x0
PORTA_MODEL	0x4003C004	0x00000000
PORTA_DOUT	0x4003C010	0x00000000
PORTA_DIN	0x4003C014	0x00000000
PORTB_CTRL	0x4003C030	0x00000000
PORTB_MODEL	0x4003C034	0x00000000
PORTB_DOUT	0x4003C040	0x00000000
PORTB_DIN	0x4003C044	0x00000000
PORTC_CTRL	0x4003C060	0x00000000
PORTC_MODEL	0x4003C064	0x00000000
PORTC_DOUT	0x4003C070	0x00000000
PORTC_DIN	0x4003C074	0x00000000

Peripherals

Peripheral	Address	Description
<input type="checkbox"/> CMU_NS	0x50008000	CMU Registers
<input type="checkbox"/> DPLL0	0x4001C000	DPLL Registers
<input type="checkbox"/> DPLL0_NS	0x5001C000	DPLL Registers
<input type="checkbox"/> EMU	0x40004000	EMU Registers
<input type="checkbox"/> EMU_NS	0x50004000	EMU Registers
<input type="checkbox"/> FRC	0xA8004000	FRC Registers
<input type="checkbox"/> FRC_NS	0xB8004000	FRC Registers
<input type="checkbox"/> FSRCO	0x40018000	FSRCO Registers
<input type="checkbox"/> FSRCO_NS	0x50018000	FSRCO Registers
<input type="checkbox"/> GPCRC	0x40088000	GPCRC Registers
<input type="checkbox"/> GPCRC_NS	0x50088000	GPCRC Registers
<input type="checkbox"/> GPIO	0x4003C000	GPIO Registers
<input checked="" type="checkbox"/> GPIO_NS	0x5003C000	GPIO Registers
<input type="checkbox"/> HFRCO0	0x40010000	HFRCO Registers
<input type="checkbox"/> HFRCO0_NS	0x50010000	HFRCO Registers
<input type="checkbox"/> HFRCOEM23	0x4A014000	HFRCO Registers
<input type="checkbox"/> HFRCOEM23_NS	0x5A014000	HFRCO Registers
<input type="checkbox"/> HFX00	0x4000C000	SYXO Registers
<input type="checkbox"/> HFX00_NS	0x5000C000	SYXO Registers
<input type="checkbox"/> I2C0	0x4A010000	I2C Registers
<input type="checkbox"/> I2C0_NS	0x5A010000	I2C Registers

No details to display for the current selection.

GPIO\_NS: 0x5003C000

Register	Address	Value
GPIO_NS	0x5003C000	
PORTA_CTRL	0x5003C000	0x00400040
SLEWRATE	[6:4]	0x4
DINDIS	[12]	0x0
SLEWRATEALT	[22:20]	0x4
DINDISALT	[28]	0x0
PORTA_MODEL	0x5003C004	0x00004000
PORTA_DOUT	0x5003C010	0x00000008
PORTA_DIN	0x5003C014	0x00000008
PORTB_CTRL	0x5003C030	0x00400040
PORTB_MODEL	0x5003C034	0x00000000
PORTB_DOUT	0x5003C040	0x00000000
PORTB_DIN	0x5003C044	0x00000000
PORTC_CTRL	0x5003C060	0x00400040
PORTC_MODEL	0x5003C064	0x00000000
PORTC_DOUT	0x5003C070	0x00000000
PORTC_DIN	0x5003C074	0x00000000

The GNU Debugger is the preferred choice for TrustZone debugging and is the default debugger for Simplicity Studio ≥ v5.5.0.0.



# TrustZone Platform Examples

# TrustZone Platform Examples

The following TrustZone platform examples located in the C:\Users<PC USER NAME>\SimplicityStudio\SDKs\gecko\_sdk\app\common\example folder (Windows) demonstrate the TrustZone implementation on Series 2 devices. All TrustZone platform examples do not include [Gecko Bootloader](#).

## TrustZone PSA Attestation

**tz\_psa\_attestation\_ws**  
This example workspace demonstrates TrustZone for PSA Attestation.  
[View Project Documentation](#)

CREATE

SimplicityStudio > SDKs > gecko\_sdk > app > common > example > tz\_psa\_attestation

Name

- tz\_psa\_attestation\_ns
- readme.md
- tz\_psa\_attestation\_s.slcw
- tz\_psa\_attestation\_ws.slcw

Example Folder	Description
tz_psa_attestation	The workspace description file (tz_psa_attestation_ws.slcw) creates the TrustZone PSA Attestation example. The project description file (tz_psa_attestation_s.slcw) configures a Secure application that provides the Secure Library functionality required by the Non-secure application.
tz_psa_attestation_ns	The project description file (tz_psa_attestation_ns.slcw) configures a Non-secure application for the TrustZone PSA Attestation example.

Notes:

- This example cannot run if the `SECURE_BOOT_ENABLE` (root of trust of the attestation) option in SE OTP is disabled.
- The combined image of Secure and Non-secure applications is signed by the `example_signing_key.pem` (private key) in C:\Users<PC USER NAME>\SimplicityStudio\SDKs\gecko\_sdk\platform\common\example folder (Windows). The `example_signing_pubkey.pem` (public key) in the same folder is installed to the SE OTP to verify the image signature during [Secure Boot](#).

## TrustZone PSA Crypto ECDH

tz\_psa\_crypto\_ecdh\_ws

This example workspace demonstrates TrustZone for ECDH key agreement.

CREATE

[View Project Documentation](#)

SimplicityStudio > SDKs > gecko\_sdk > app > common > example > tz\_psa\_crypto\_ecdh >

Name
tz_psa_crypto_ecdh_ns
readme.md
tz_psa_crypto_ecdh_s.slcw
tz_psa_crypto_ecdh_ws.slcw

Example Folder	Description
tz_psa_crypto_ecdh	The workspace description file (tz_psa_crypto_ecdh_ws.slcw) upgrades the existing Platform - PSA Crypto ECDH example to TrustZone-aware. The project description file (tz_psa_crypto_ecdh_s.slcw) configures a Secure application that provides the Secure Library functionality required by the Non-secure application.
tz_psa_crypto_ecdh_ns	The project description file (tz_psa_crypto_ecdh_ns.slcw) configures the existing Platform - PSA Crypto ECDH example as a Non-secure application. The source code can be reused without changes.

The following sections use Simplicity Studio v5.6.3.0 and GSDK v4.2.2. The procedures and pictures may be different if using higher versions of Simplicity Studio 5 and GSDK.

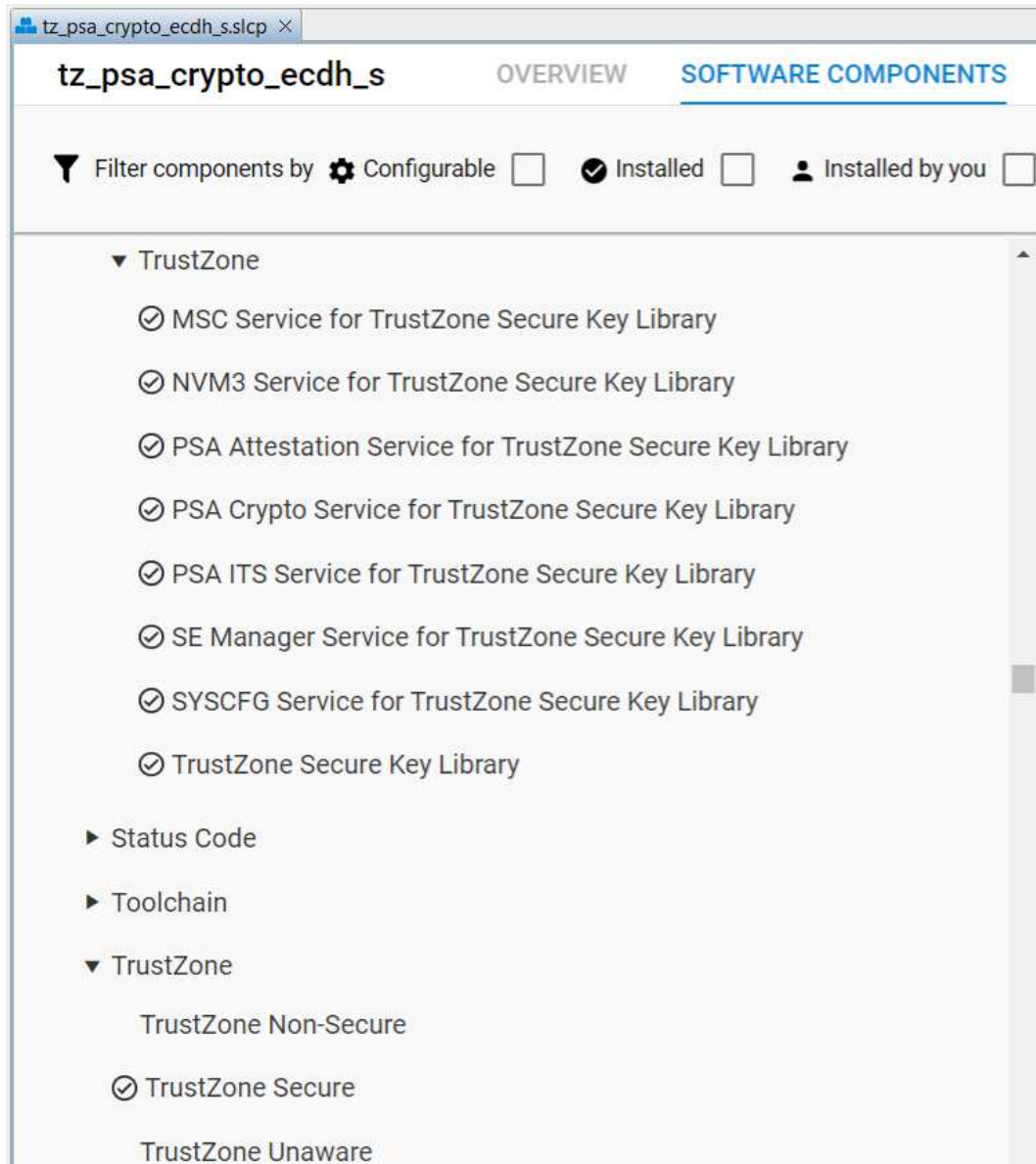
Project Description File

The project description file ( .slcw ) contains references to the GSDK used and a list of components to use from these. The TrustZone-aware application requires separate slcw files for the Secure and Non-secure applications.

Users should not directly edit the slcw files, but rather use the [Memory Editor](#) and Post Build Editor in Simplicity Studio to update the [memory configuration](#) and post-build actions.

Secure Application

The following figure describes which TrustZone software components are installed for the TrustZone Secure library of the [TrustZone PSA Crypto ECDH](#) example.

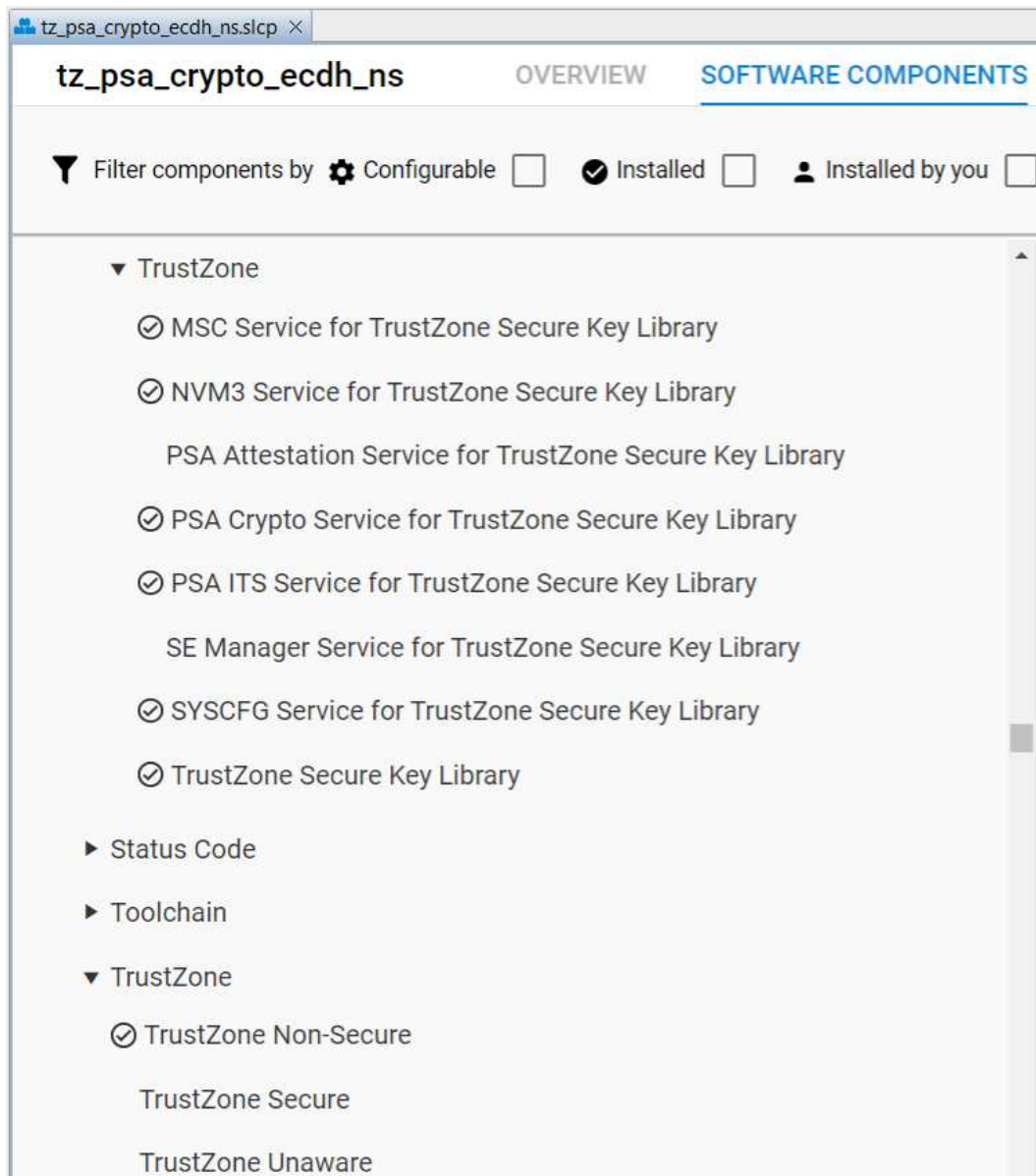


#### Notes:

- The services provided by the Secure library are standardized.
- The source files for the Secure library will be automatically added to the application when generating the Secure project from the `slcp` file. For the current TrustZone implementation, modifications of the source files of the Secure library are not recommended.

## Non-secure Application

The following figure describes which TrustZone software components are installed for the Non-secure application of the [TrustZone PSA Crypto ECDH](#) example.

**Notes:**

- The following software components are automatically installed when [PSA Crypto and ITS](#) services are used on the Non-secure application.
  - MSC Service for TrustZone Secure Key Library
  - NVM3 Service for TrustZone Secure Key Library
  - PSA Crypto Service for TrustZone Secure Key Library
  - PSA ITS Service for TrustZone Secure Key Library
  - SYSCFG Service for TrustZone Secure Key Library
- The following software components can be installed to the Non-secure application when those services are required.
  - [PSA Attestation Service for TrustZone Secure Key Library](#)
  - [SE Manager Service for TrustZone Secure Key Library](#)

## Workspace

A workspace is a structure that can contain multiple projects. 'Workspace' is a generic term for this construct. In the context of Simplicity Studio, where workspace has a different, Eclipse-based, meaning, workspaces are referred to as [Solutions](#).

The workspace description file ( `.slcw` ) contains references to projects ( `.slcp` ) that make up the workspace. Users should not directly edit the `slcw` file, but rather use the Post Build Editor in Simplicity Studio to update the post-build actions.

## Memory Configuration

The [memory configurations](#) in the TrustZone platform examples are based on the Series 2 radio board with minimum flash (512 kB) and RAM (32 kB), so these configurations can run on all Series 2 radio boards. Users can customize the settings when more flash and RAM are available on the selected device.

- Memory flash size (total) = `memory_flash_size` (S) + `memory_flash_size` (NS) = 512 kB
- Memory RAM size (total) = `memory_ram_size` (S) + `memory_ram_size` (NS) = 32 kB

## Secure Application

The project description file of the Secure application ( `*_s.slcp` ) uses the default memory setting below to generate the [Secure linker file](#) (linkerfile.ld for GCC and linkerfile.icf for IAR in the project autogen folder).

The actual memory usage during software development is unknown, so it needs to reserve enough flash ( `memory_flash_size` : 176 kB) and RAM ( `memory_ram_size` : 12 kB) for the Secure part of all TrustZone platform examples. The bigger RAM size (including stack and heap) is mainly for the software fallback on cryptographic operations in PSA Crypto.

Default Memory Setting (Secure)	xG21 and xG22 Devices	Other Series 2 Devices
memory_flash_start	0x00000000	0x08000000
memory_flash_size	0x0002C000 (176 kB)	0x0002C000 (176 kB)
memory_ram_start	0x20000000	0x20000000
memory_ram_size	0x00003000 (12 kB)	0x00003000 (12 kB)

```
MEMORY
{
  FLASH (rx) : ORIGIN = 0x0, LENGTH = 0x2c000
  RAM (rwx) : ORIGIN = 0x20000000, LENGTH = 0x3000
}
```

## Non-secure Application

The project description files of the Non-secure application ( `*_ns.slcp` ) use the default memory setting below to generate the [Non-secure linker file](#) (linkerfile.ld for GCC and linkerfile.icf for IAR in the project autogen folder).

The actual memory usage during software development is unknown, so the remaining flash ( `memory_flash_size` : 336 kB) and RAM ( `memory_ram_size` : 20 kB) should be big enough for the Non-secure part of all TrustZone platform examples.

Default Memory Setting (Non-secure)	xG21 and xG22 Devices	Other Series 2 Devices
memory_flash_start	0x0002C000 (176 kB)	0x0802C000 (176 kB)
memory_flash_size	0x00054000 (336 kB)	0x00054000 (336 kB)
memory_ram_start	0x20003000 (12 kB)	0x20003000 (12 kB)
memory_ram_size	0x00005000 (20 kB)	0x00005000 (20 kB)

```
MEMORY
{
  FLASH (rx) : ORIGIN = 0x2c000, LENGTH = 0x54000
  RAM (rwx) : ORIGIN = 0x20003000, LENGTH = 0x5000
}
```

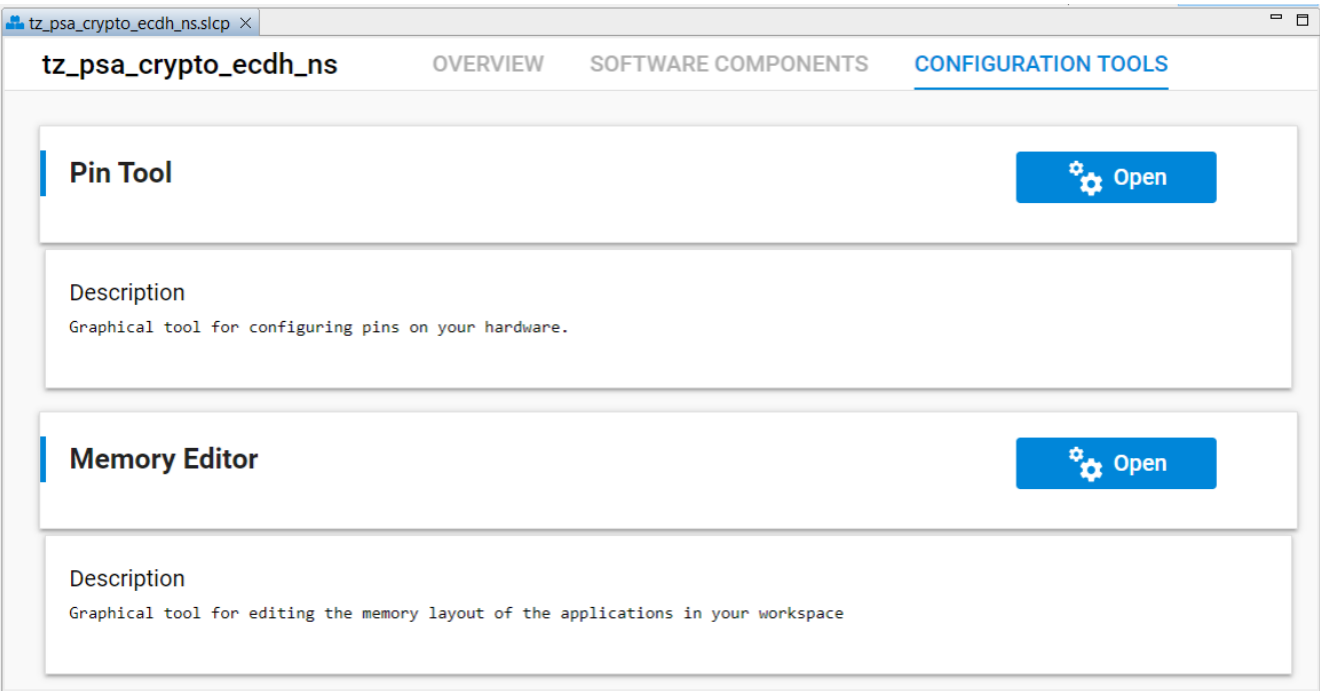
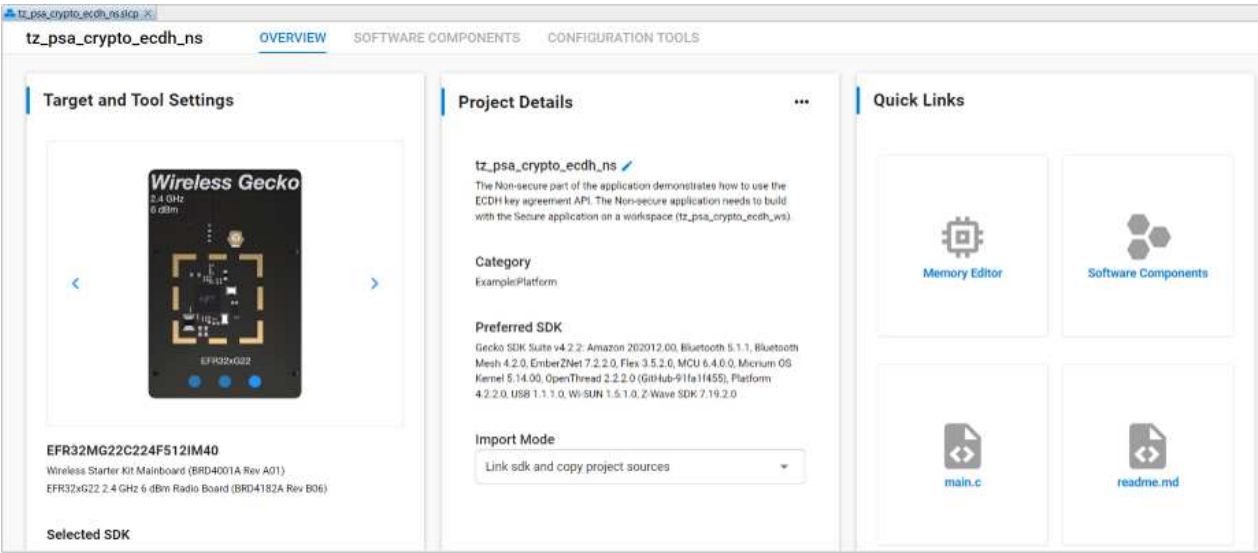
**Note:** The usable flash for Non-secure code should be equal to `memory_flash_size` - NVM size (default is 40 kB) if NVM3 storage is required.

### Memory Editor

The default memory setting of [Secure](#) and [Non-secure](#) applications are good enough for software development and debugging. The final memory layouts of Secure and Non-secure projects are deduced by inspecting the flash and RAM usage in the Secure application memory map file (.map).

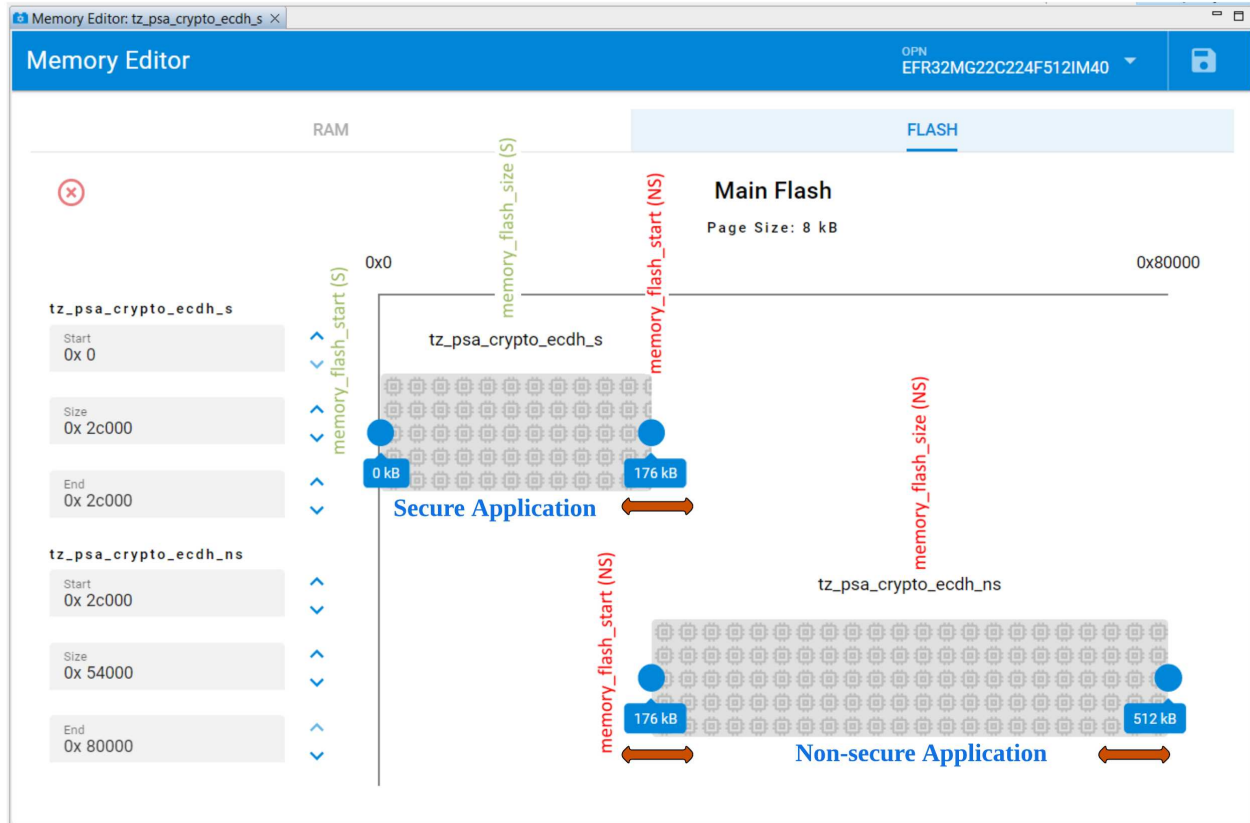
The [Memory Editor](#) in Simplicity Studio 5 is a graphical tool for editing the memory layout (flash and RAM) of the applications in the workspace. The Memory Editor will update the linker file in the project `autogen` folder with the custom settings. [Rebuild](#) the projects to use the new memory configurations in the linker files.

The Memory Editor is located at the **Quick Links** and **CONFIGURATION TOOLS** of Secure or Non-secure `slcp` file.



The following items will be determined by the flash usage in the Secure application memory map file:

- memory\_flash\_size (S)
- memory\_flash\_start (NS)
- memory\_flash\_size (NS)

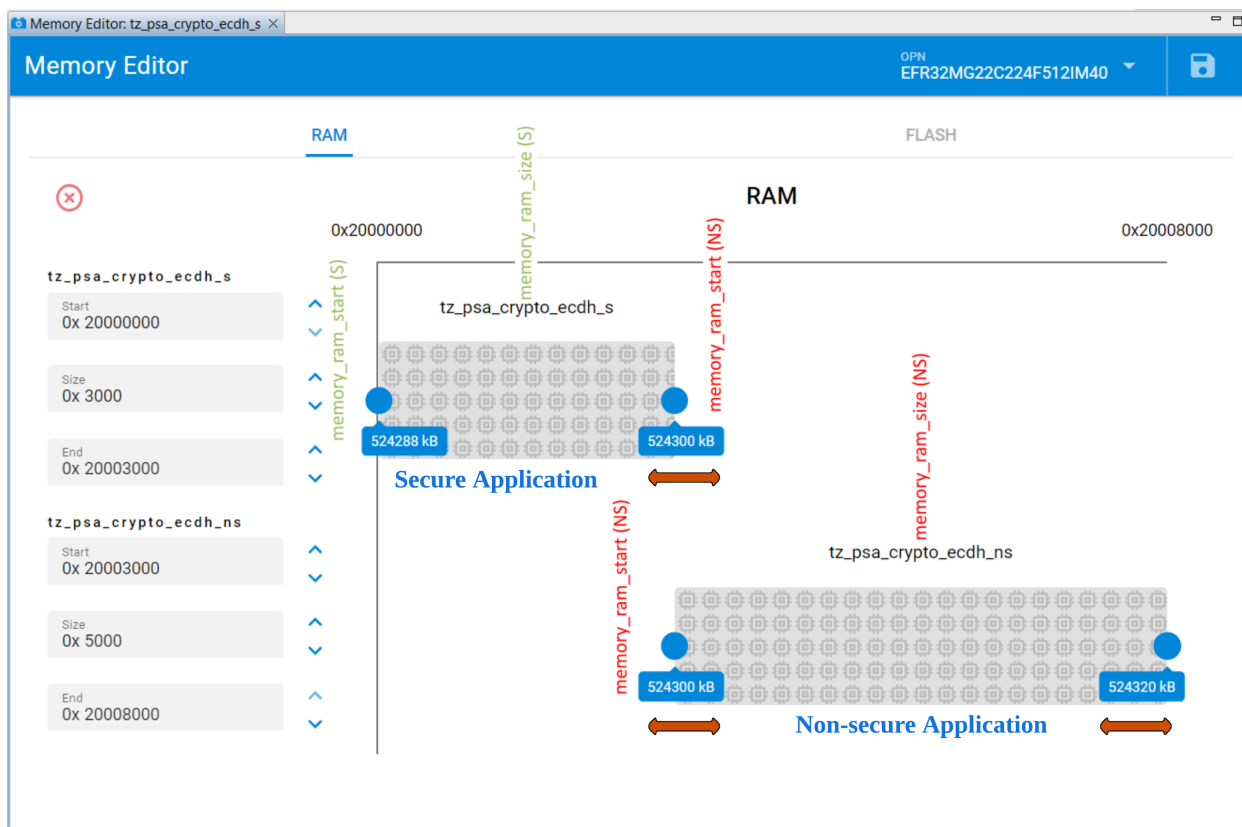


**Note:** The Memory Editor in Simplicity Studio v5.6.3.0 can only adjust the flash size in **8 kB** (page size) alignment, which may not fit the **4kB alignment** between the Secure and Non-secure flash boundary.

The following items will be determined by the RAM usage in the Secure application memory map file:

- memory\_ram\_size (S)
- memory\_ram\_start (NS)
- memory\_ram\_size (NS)





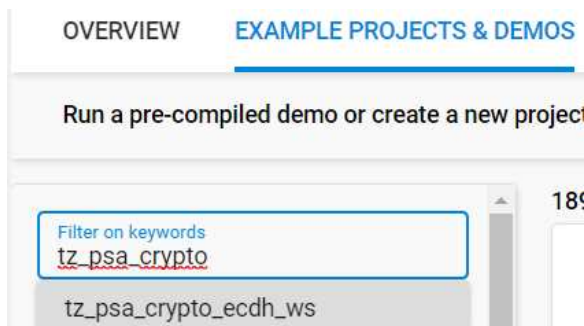
## Build

The Secure project must be built first to create the Secure object library (trustzone\_secure\_library.o) with function entries for the Non-secure project. Both projects need to be rebuilt if any changes in the Secure project. Users can use Simplicity IDE in Simplicity Studio 5 or IAR EWARM v9.20.4 to build the TrustZone platform examples.

## Simplicity IDE

The following procedures are based on the **TrustZone PSA Crypto ECDH** example on BRD4182A Radio Board (EFR32MG22C224F512IM40).

1. Use the `tz_psa_crypto` keyword to search in **EXAMPLE PROJECTS & DEMOS** tab. Select the `tz_psa_crypto_ecdh_ws` example.



2. Click **[CREATE]** to generate the [solution](#).



### tz\_psa\_crypto\_ecdh\_ws

This example workspace demonstrates TrustZone for ECDH key agreement.

[View Project Documentation](#)

CREATE

3. The Project Configuration dialog shows the Secure and Non-secure projects in the target solution. Click **[FINISH]** to start the creation process.

New Project Wizard

Project Configuration

Select the project name and location.

✓ Target, SDK

✓ Examples

✎ Configuration

Solution name:tz\_psa\_crypto\_ecdh\_ws

Project name:tz\_psa\_crypto\_ecdh\_s

Project name:tz\_psa\_crypto\_ecdh\_ns

✓ Use default location

Location:C:\Users\amleung\SimplicityStudio\v5\_workspace

BROWSE

With project files:

○ Link to sources

● Link sdk and copy project sources

○ Copy contents

CANCEL

BACK

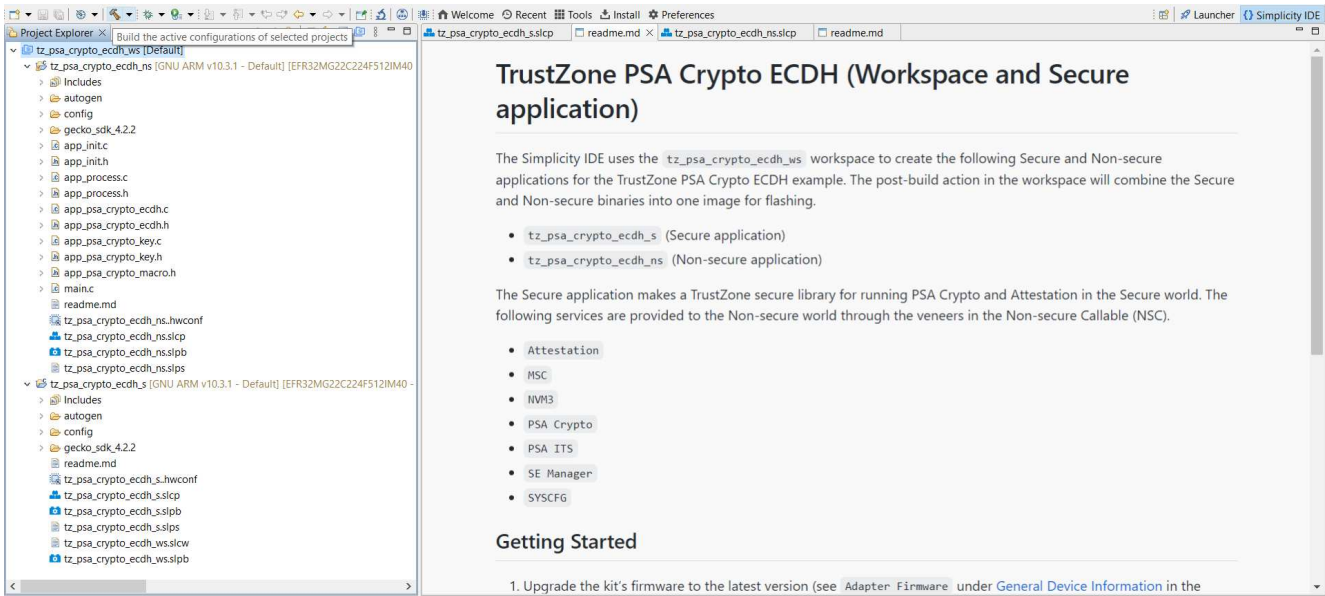
NEXT

FINISH

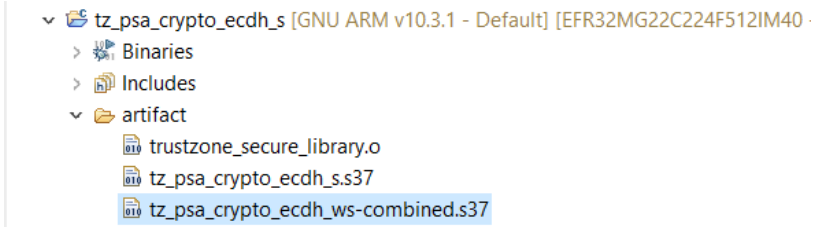
4. The Simplicity IDE perspective opens after finishing the solution creation. Click **Build** on the Simplicity IDE perspective toolbar to build the projects of a selected solution in order (Secure then Non-secure).

Copyright © 2025 Silicon Laboratories. All rights reserved.

141/280



5. The post-build actions ( .slpb files) of the Secure project, Non-secure project, and workspace will be processed in sequence if the solution is successfully built. The combined image (tz\_psa\_crypto\_ecdh\_ws-combined.s37) in the Secure project artifact folder can be used for programming the device or debugging.



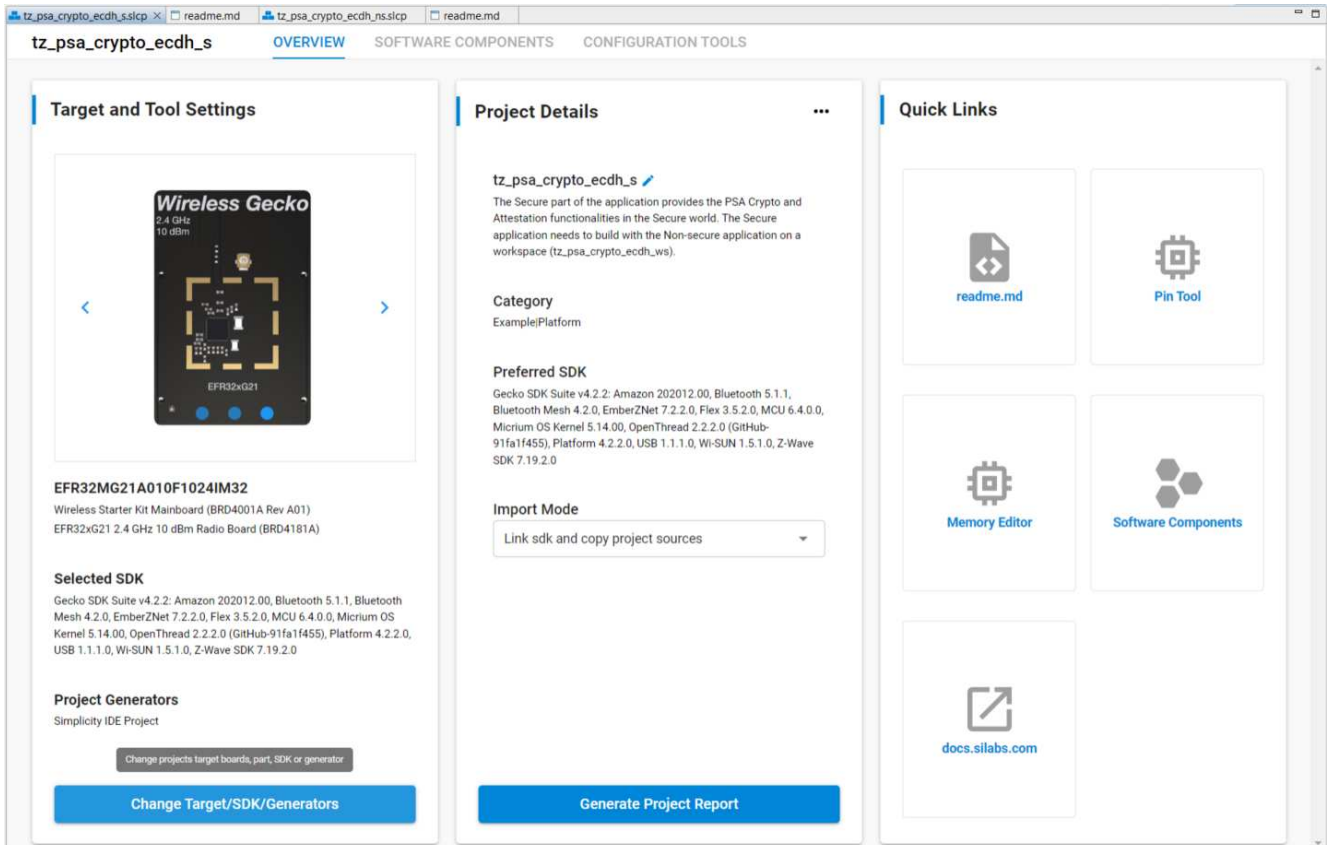
6. Use [Memory Editor](#) to finalize the memory layouts of Secure and Non-secure applications and rebuild the solution to update the memory configurations.

**Note:** The Simplicity IDE can only apply the post-build action to a particular project if multiple Secure or Non-secure projects exist in the solution.

IAR EWARM

The following procedures are based on the **TrustZone PSA Crypto ECDH** example on BRD4181A Radio Board (EFR32MG21A010F1024IM32).

- 1. Follow steps 1 to 3 in [TrustZone PSA Crypto ECDH](#) to generate the solution for the `tz_psa_crypto_ws` . Select the `tz_psa_crypto_ecdh_s.slpb` file.
- 2. The **Overview** tab shows the **Target and Tool Settings** card on the left side. Scroll down if necessary and click **[ChangeTarget/SDK/Generators]**.



**Target and Tool Settings**

**Wireless Gecko**  
2.4 GHz  
10 dBm

**EFR32MG21A010F1024IM32**  
Wireless Starter Kit Mainboard (BRD4001A Rev A01)  
EFR32xG21 2.4 GHz 10 dBm Radio Board (BRD4181A)

**Selected SDK**  
Gecko SDK Suite v4.2.2: Amazon 202012.00, Bluetooth 5.1.1, Bluetooth Mesh 4.2.0, EmberZNet 7.2.2.0, Flex 3.5.2.0, MCU 6.4.0.0, Micrium OS Kernel 5.14.00, OpenThread 2.2.2.0 (GitHub-91fa1f455), Platform 4.2.2.0, USB 1.1.1.0, Wi-SUN 1.5.1.0, Z-Wave SDK 7.19.2.0

**Project Generators**  
Simplicity IDE Project

Change projects target boards, part, SDK or generator

**Change Target/SDK/Generators**

**Project Details**

**tz\_psa\_crypto\_ecdh\_s**

The Secure part of the application provides the PSA Crypto and Attestation functionalities in the Secure world. The Secure application needs to build with the Non-secure application on a workspace (tz\_psa\_crypto\_ecdh\_ws).

**Category**  
Example|Platform

**Preferred SDK**  
Gecko SDK Suite v4.2.2: Amazon 202012.00, Bluetooth 5.1.1, Bluetooth Mesh 4.2.0, EmberZNet 7.2.2.0, Flex 3.5.2.0, MCU 6.4.0.0, Micrium OS Kernel 5.14.00, OpenThread 2.2.2.0 (GitHub-91fa1f455), Platform 4.2.2.0, USB 1.1.1.0, Wi-SUN 1.5.1.0, Z-Wave SDK 7.19.2.0

**Import Mode**  
Link sdk and copy project sources

**Generate Project Report**

**Quick Links**

[README.md](#)

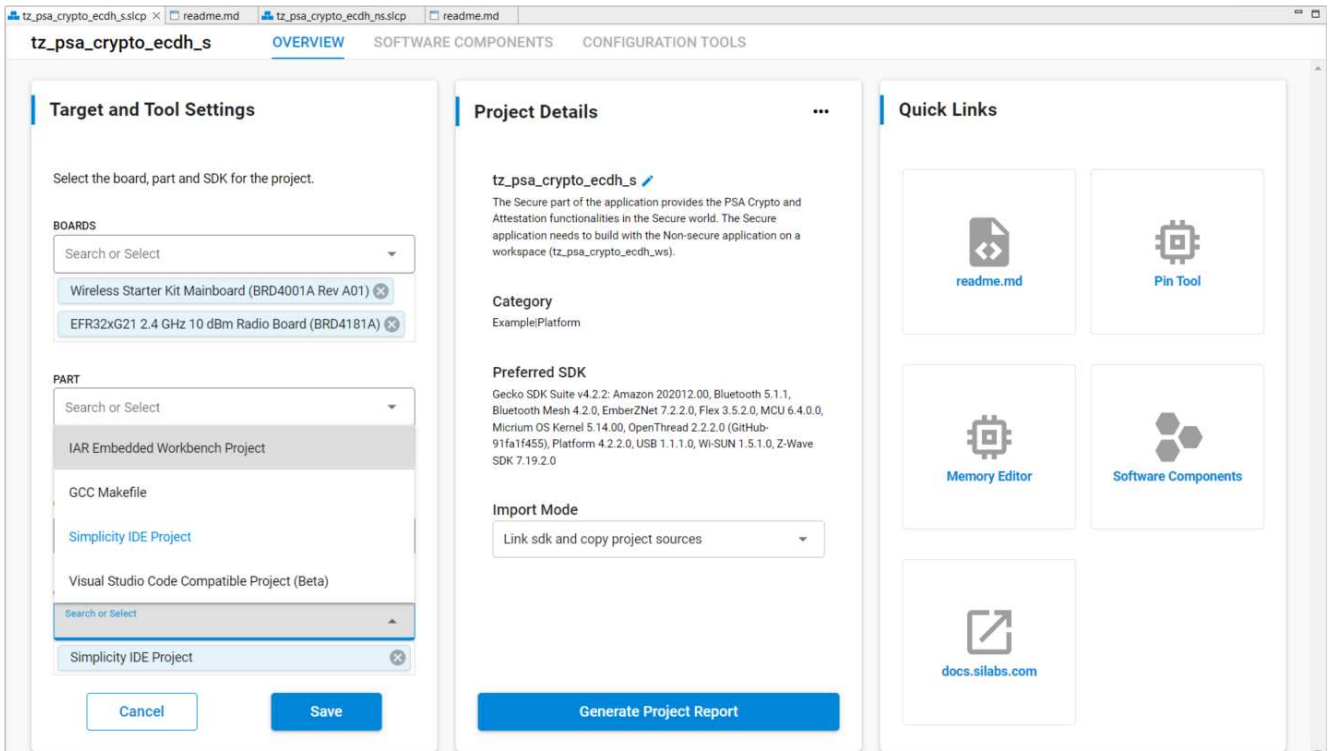
[Pin Tool](#)

[Memory Editor](#)

[Software Components](#)

[docs.silabs.com](#)

3. Drop down the **CHANGE PROJECT GENERATORS** list and select **IAR Embedded Workbench Project**.



**Target and Tool Settings**

Select the board, part and SDK for the project.

**BOARDS**

Search or Select

Wireless Starter Kit Mainboard (BRD4001A Rev A01)

EFR32xG21 2.4 GHz 10 dBm Radio Board (BRD4181A)

**PART**

Search or Select

IAR Embedded Workbench Project

GCC Makefile

Simplicity IDE Project

Visual Studio Code Compatible Project (Beta)

Search or Select

Simplicity IDE Project

Cancel Save

**Project Details**

**tz\_psa\_crypto\_ecdh\_s**

The Secure part of the application provides the PSA Crypto and Attestation functionalities in the Secure world. The Secure application needs to build with the Non-secure application on a workspace (tz\_psa\_crypto\_ecdh\_ws).

**Category**  
Example|Platform

**Preferred SDK**  
Gecko SDK Suite v4.2.2: Amazon 202012.00, Bluetooth 5.1.1, Bluetooth Mesh 4.2.0, EmberZNet 7.2.2.0, Flex 3.5.2.0, MCU 6.4.0.0, Micrium OS Kernel 5.14.00, OpenThread 2.2.2.0 (GitHub-91fa1f455), Platform 4.2.2.0, USB 1.1.1.0, Wi-SUN 1.5.1.0, Z-Wave SDK 7.19.2.0

**Import Mode**  
Link sdk and copy project sources

**Generate Project Report**

**Quick Links**

[README.md](#)

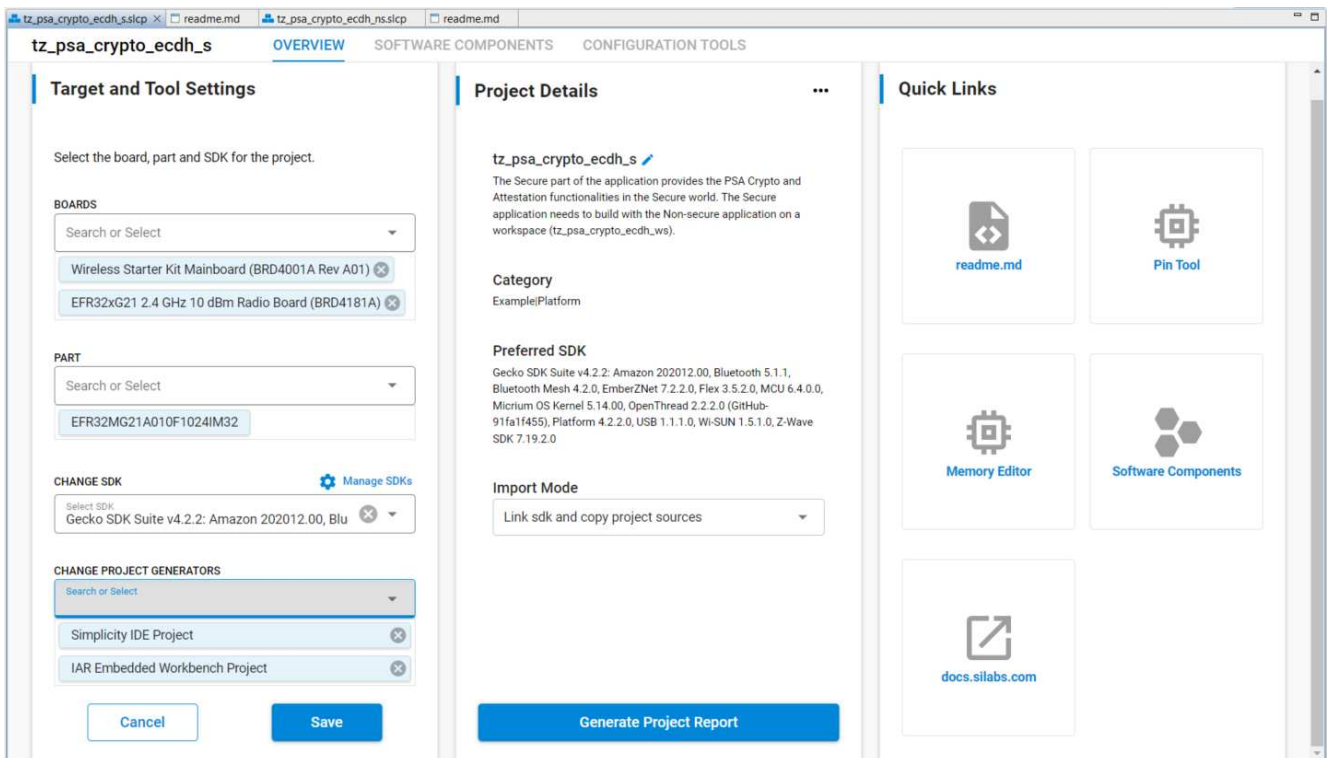
[Pin Tool](#)

[Memory Editor](#)

[Software Components](#)

[docs.silabs.com](#)

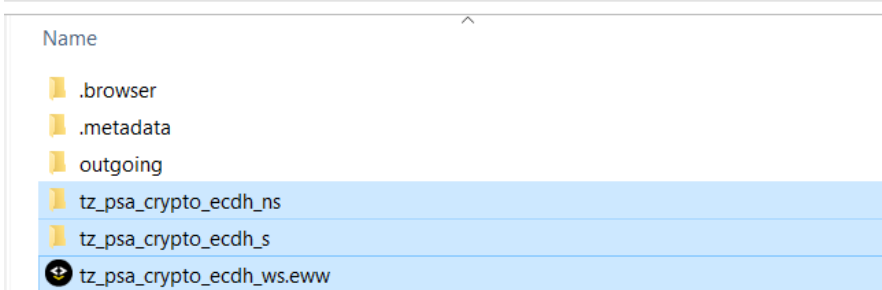
4. Click **[Save]** to generate an IAR Secure project (tz\_psa\_crypto\_ecdh\_s.ewp).



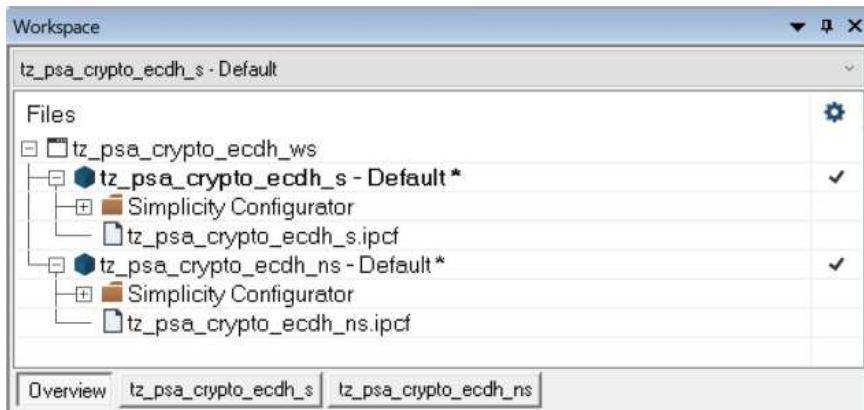
5. Select the `tz_psa_crypto_ecdh_ns.sclp` file. Repeat steps 2 to 4 to generate an IAR **Non-secure\*** project (`tz_psa_crypto_ecdh_ns.ewp`). 6. Use a text editor to create an IAR `tz_psa_crypto_ecdh_ws.eww`file (shown below) to house the projects (`tz_psa_crypto_ecdh_s.ewp` and `tz_psa_crypto_ecdh_ns.ewp`) generated in steps 4 and 5. The location of the `tz_psa_crypto_ecdh_ws.eww` is the directory for *WS<sub>D</sub>IR*.

```
<?xml version="1.0" encoding="iso-8859-1"?>
<workspace>
  <project>
    <path>$WS_DIR$\tz_psa_crypto_ecdh_s\tz_psa_crypto_ecdh_s.ewp</path>
  </project>
  <project>
    <path>$WS_DIR$\tz_psa_crypto_ecdh_ns\tz_psa_crypto_ecdh_ns.ewp</path>
  </project>
  <batchBuild/>
</workspace>
```


› SimplicityStudio › v5\_workspace

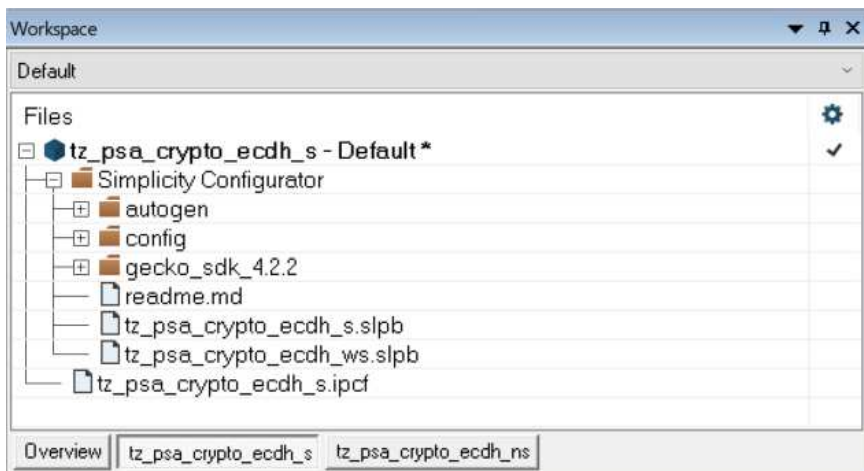


7. Double-click the `tz_psa_crypto_ecdh_ws.eww`file to open the workspace that includes Secure and Non-secure projects.



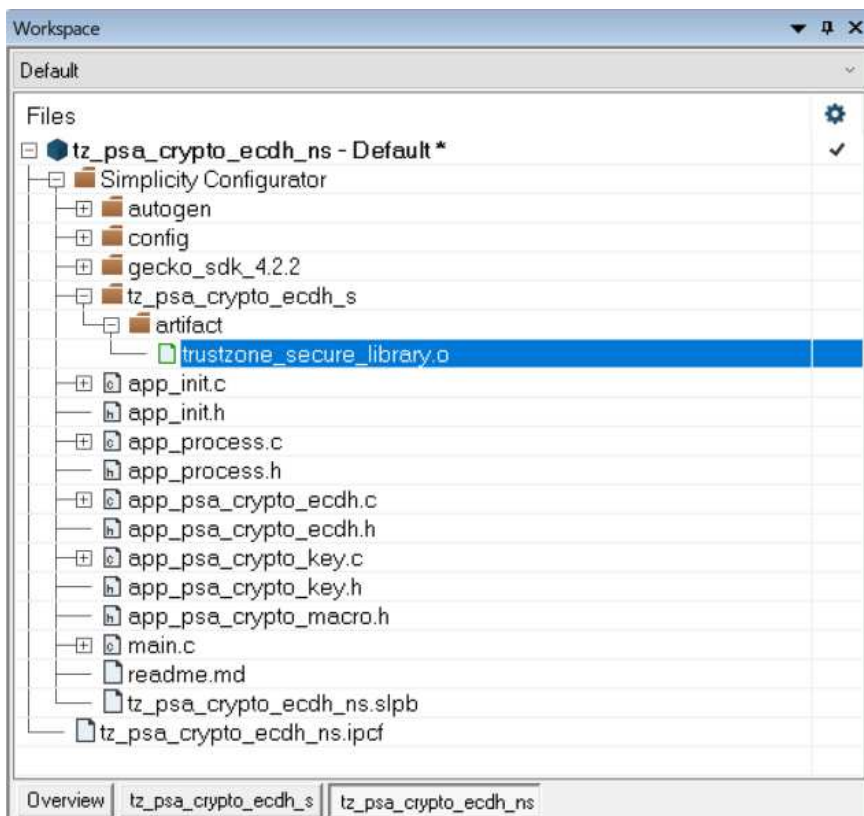
8. Click the `tz_psa_crypto_ecdh_s` tab to

open the Secure project. Click  (Make) to build. It exports the Secure object library (`trustzone_secure_library.o`) for function entries that will be used by the Non-secure project.



9. Click the `tz_psa_crypto_ecdh_ns` tab to

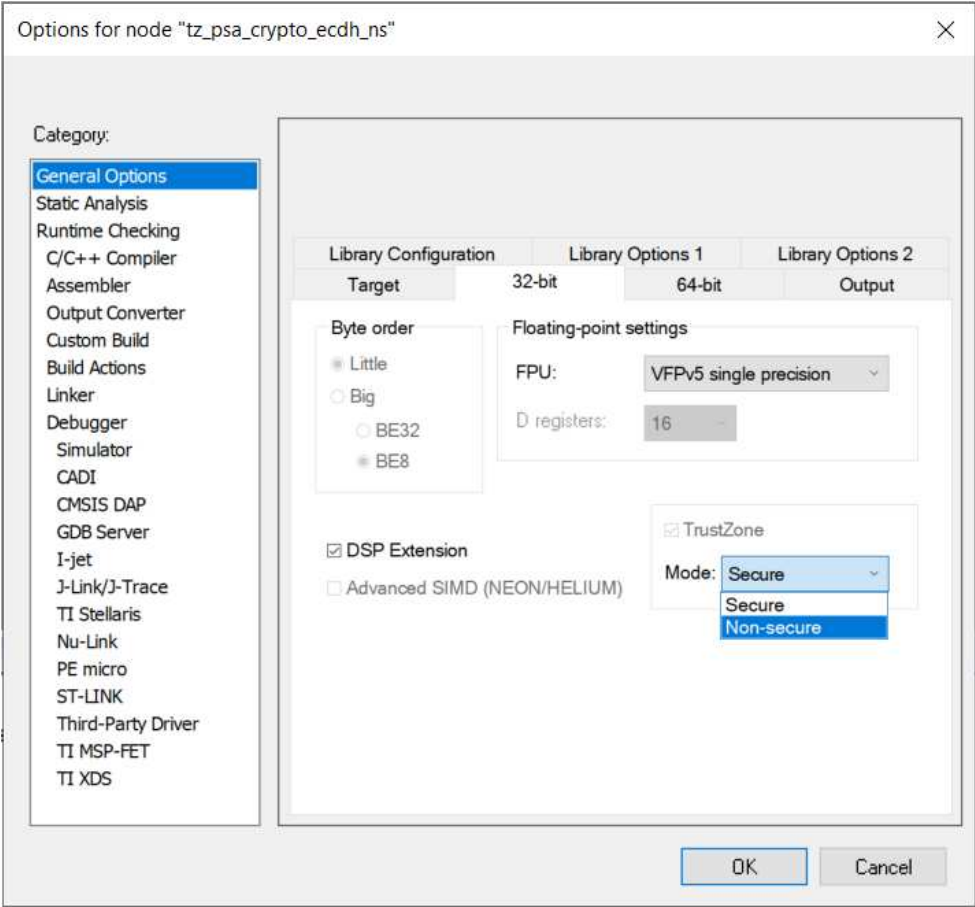
open the Non-secure project.




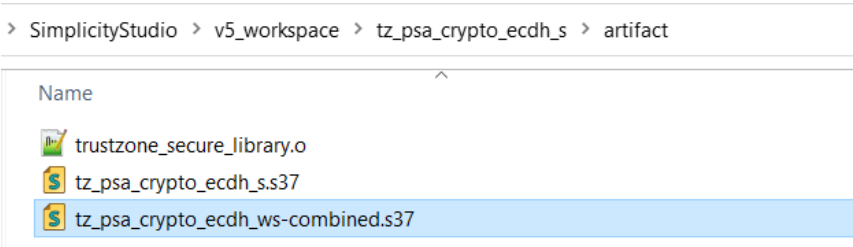
10. The `SL_TRUSTZONE_NONSECURE`

defined in the Non-secure project disables the [CMSE compiler option](#) ( `--cmse` ) regardless of whether the **Project** →

Options... → General Options → 32-bit → TrustZone → Mode: setting is Secure or Non-secure. So changing this configuration from Secure to Non-secure is optional. Click [OK] to exit.



11. Click  (Make) to build the Non-secure project. The post-build actions of the workspace (tz\_psa\_crypto\_ecdh\_ws.slbp) will be triggered in IAR to combine the Secure and Non-secure images (tz\_psa\_crypto\_ecdh\_ws-combined.s37) to the artifact folder of tz\_psa\_crypto\_ecdh\_s for programming the device.



12. Use [Memory Editor](#) to finalize the memory layouts of Secure and Non-secure applications and rebuild the Secure and Non-secure projects to update the memory configurations.

**Note:** The IAR EWARM can only apply the workspace post-build action to a particular project if multiple Secure or Non-secure projects exist in the workspace.

## Debugging

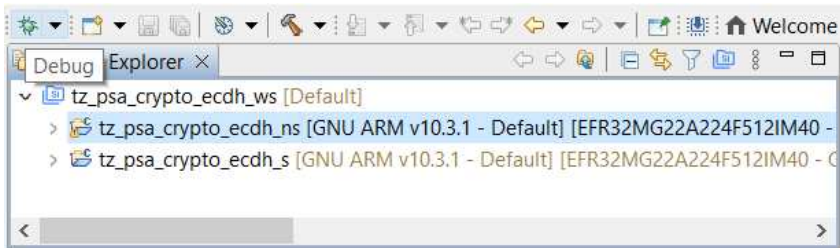
Users can use Simplicity IDE in Simplicity Studio 5 or IAR EWARM v9.20.4 to debug the TrustZone platform examples. Building the projects with Optimization Level None (-OO) is recommended for debugging.

### Simplicity IDE

The TrustZone debugging process on Simplicity IDE is similar to the existing sample projects in Simplicity Studio.



- 2. Flash the combined image (tz\_psa\_crypto\_ecdh\_ws-combined.s37) generated in [Simplicity IDE](#) to the device.
- 3. Select the Secure or Non-secure project and use the **Debug** icon to launch a debug session.



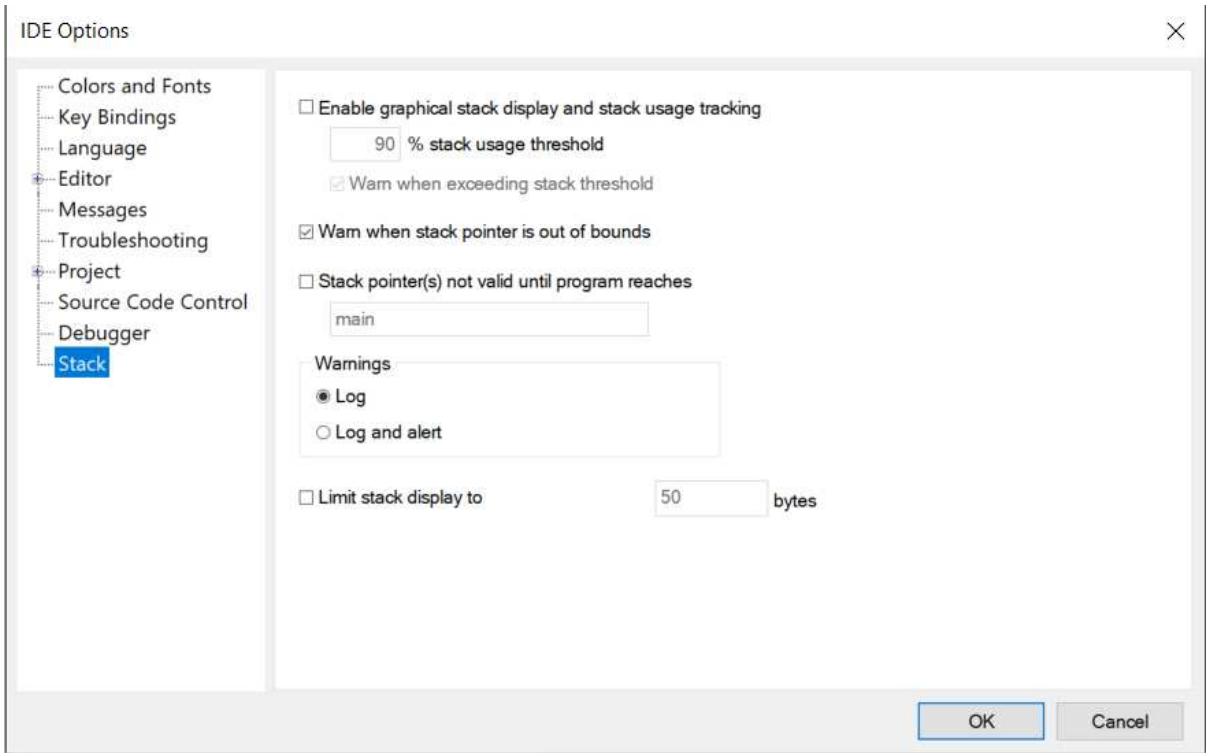
- 4. Follow the instructions in the [Using the Debugger](#) section in Simplicity Studio 5 User's Guide to debug the Secure or Non-secure application. 5. The debugger cannot step into the function in a Non-secure application when debugging the Secure application and vice versa. Use the **Program Counter** (PC in Secure or Non-secure address) in the **Registers** window to determine the program status.

Name	Value	Description
r12	0	
sp	0x20003fe0 <sl_stack+4064>	
lr	185911	
pc	0x2cf98 <app_process_action+1200>	
xpsr	1627586560	

IAR EWARM

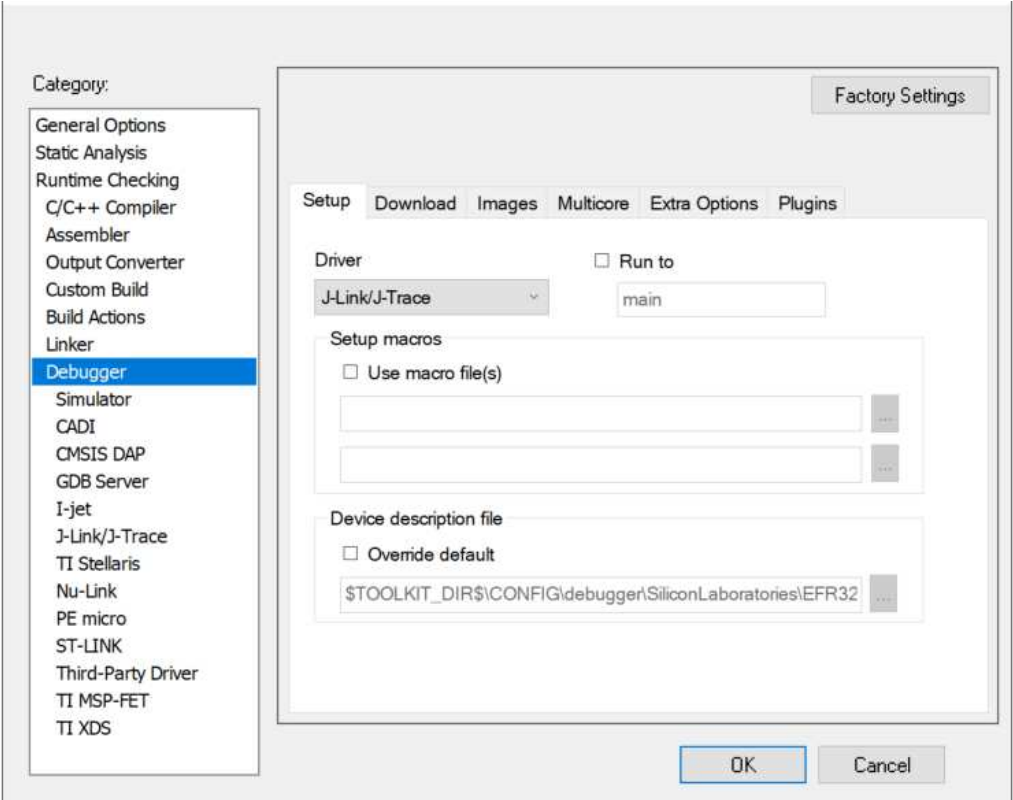
Use the tz\_psa\_crypto\_ecdh\_ws.eww workspace created in [IAR EWARM](#) for the debugger settings. Except for a minor difference in step 3, the following steps are the same as those to set up the Secure (tz\_psa\_crypto\_ecdh\_s) and Non-secure (tz\_psa\_crypto\_ecdh\_ns) projects for debugging.

- 1. Select **Options...** in the **J-Link** **Tools** **Window** context menu of the Secure or Non-secure project and open the **IDE Options** → **Stack** dialog. Uncheck the **Stack pointer(s) not valid until program reaches\*** checkbox. Click **[OK]** to exit.



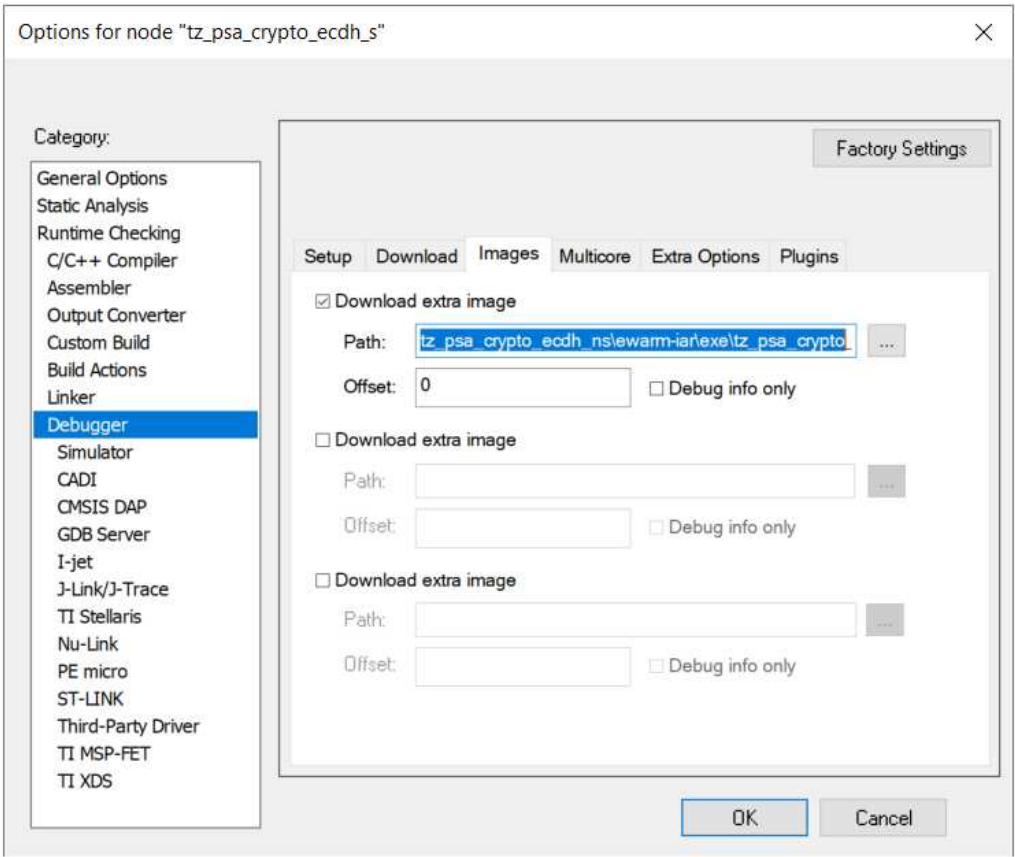
- 2. Select **Options...** in the **View** **Project** **J-Link** context menu of the Secure or Non-secure project and open the window for

Debugger options. Click the **Setup** tab to open a dialog, and uncheck the **Run to → main** checkbox. Click the **Images** tab to set up another option.



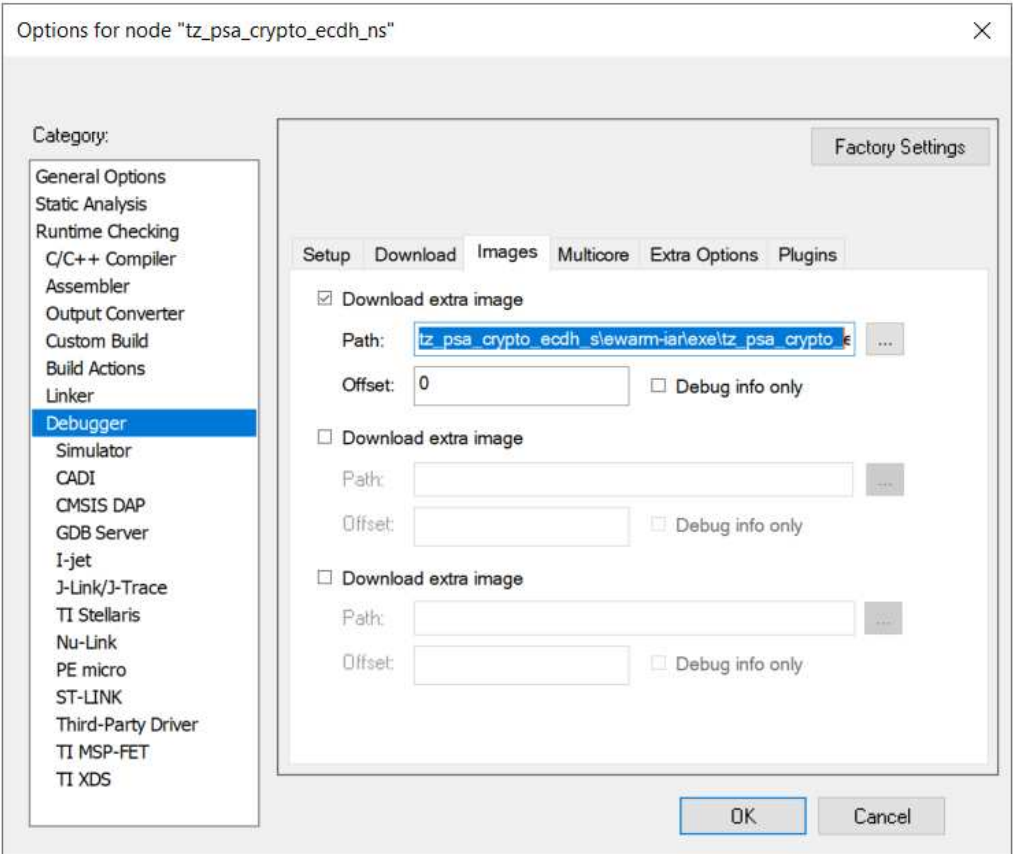
3. Check the **ownload extra image** option. Enter the location of the .out file to **Path**: with **Offset**: set to 0. All project relative paths are resolved from the directory location of the tz\_psa\_crypto\_ecdh\_ws.eww workspace file.

Location of Non-secure .out file for Secure project: tz\_psa\_crypto\_ecdh\_ns\\ewarm-iar\\exe\\tz\_psa\_crypto\_ecdh\_ns.out

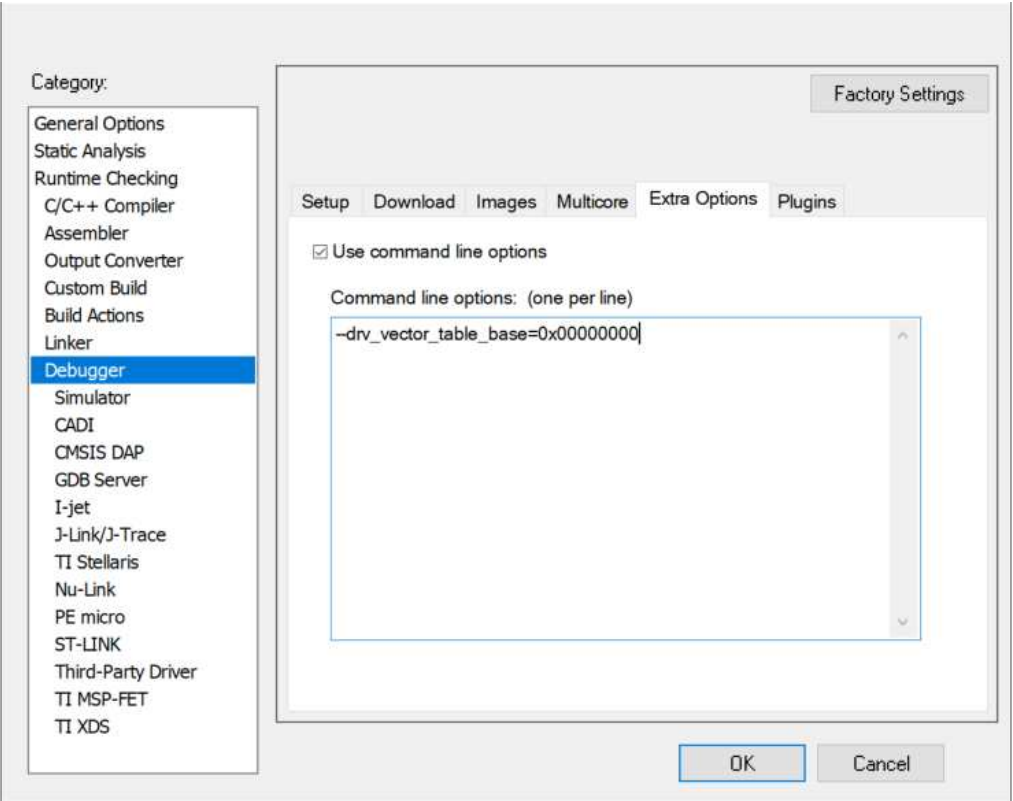






Location of Secure .out file for Non-secure project: tz\_psa\_crypto\_ecdh\_s\ewarm-iar\exe\tz\_psa\_crypto\_ecdh\_s.out

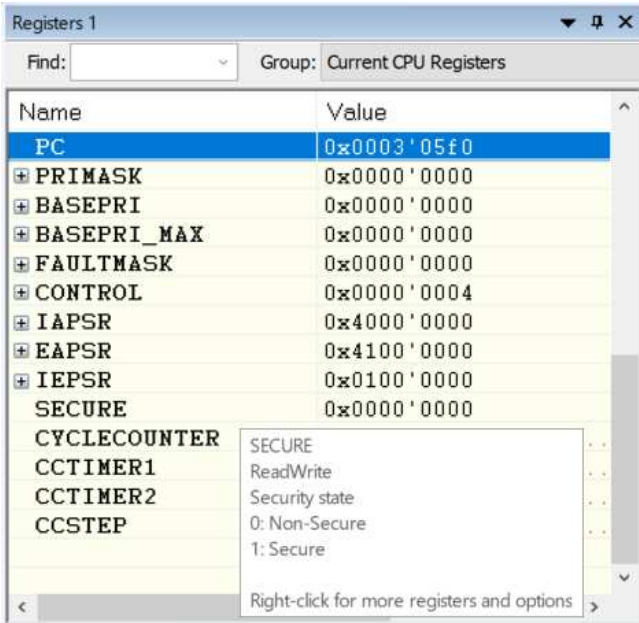



4. Click the **Extra Options** tab to set up another option. 5. Check the **Use command line options**. Enter `--drv_vector_table_base=0x00000000` to Command line options: (one per line) window. Click [OK] to exit.



6. Finish the debug settings in Secure and Non-secure projects, and click  (Download and Debug) in the Secure or Non-secure project to download the Secure and Non-secure images for debugging (assume both projects had successfully [built](#) before). Click  (Go) to

start running the code in a Secure or Non-secure project. 7. The debugger will automatically switch between Secure and Non-secure projects when stepping into a function or hitting a breakpoint in a Secure or Non-secure project. Use the **Program Counter** (PC in Secure or Non-secure address) or **SECURE** (0 or 1) in the **Registers** window to determine the program status.



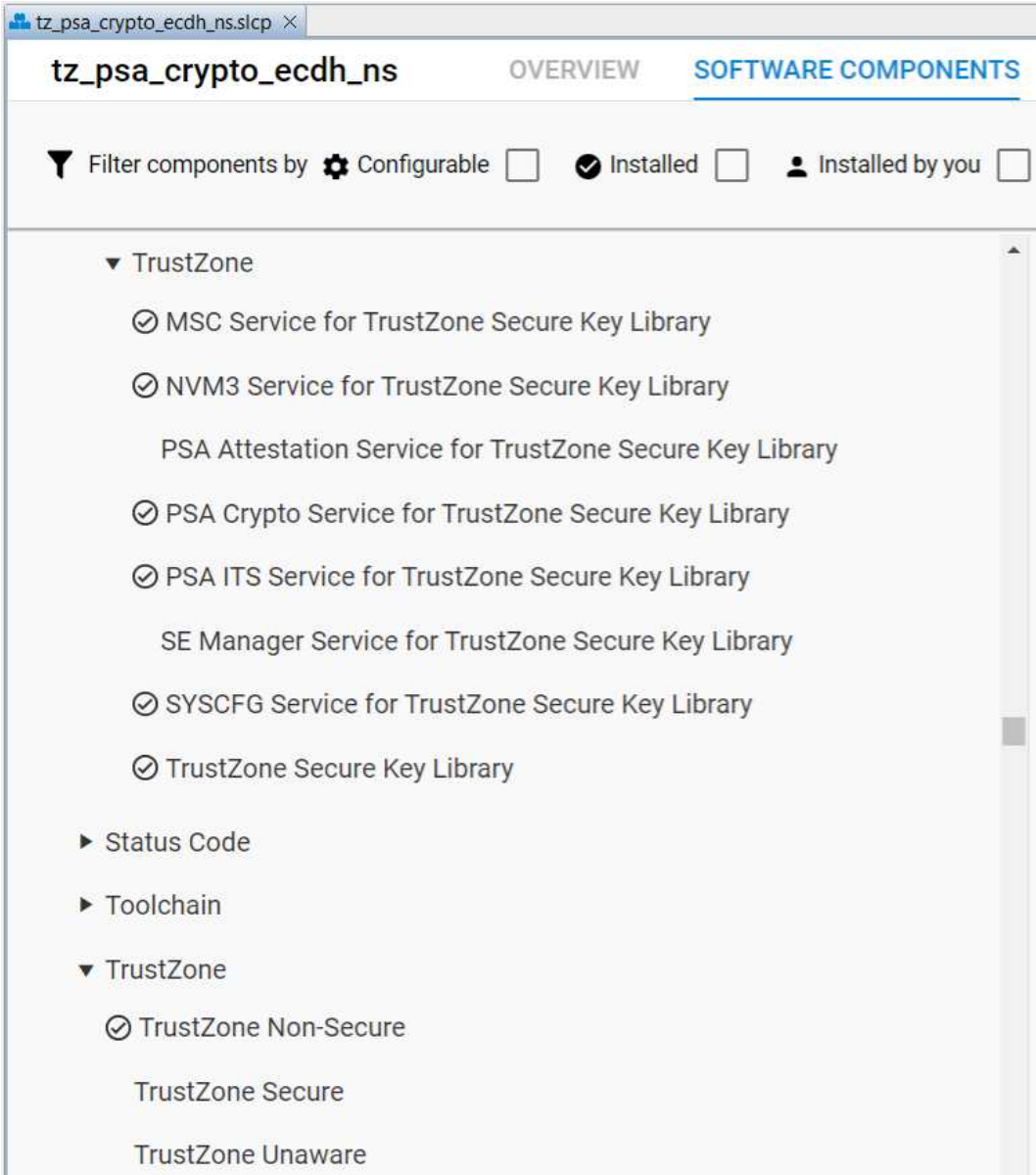
8. Click  (Stop Debugging) to end the debug session.

## Benchmark

The TrustZone implementation will affect the memory footprint and performance of cryptographic operations. The following comparisons are based on the **TrustZone PSA Crypto ECDH** example on BRD4182A Radio Board (EFR32MG22C224F512IM40) with SE firmware v1.2.14.

## Memory Footprint

The memory footprint of a TrustZone project depends on which services (software components in the figure below) provided by the [Secure Library](#) are used in the Non-secure application ( `tz_psa_crypto_ecdh_ns` project).



The following tables compare the memory footprint of the [TrustZone-unaware](#) ( Platform - PSA Crypto ECDH ) and [TrustZone-aware](#) projects ( tz\_psa\_crypto\_ecdh\_ws ) based on the following conditions.

- The tz\_psa\_crypto\_ecdh\_ns reuses the source code from the Platform - PSA Crypto ECDH example without any changes.
- The total size in tz\_psa\_crypto\_ecdh\_ns does not consider the 4 kB alignment on the Secure and Non-secure flash and RAM. The 4 kB alignment requirement will increase the actual usage of flash and RAM.
- All source code is compiled with Optimize for size (-Os) in Simplicity IDE (GNU ARM v10.3.1) of Simplicity Studio 5.

**Table:** Flash Size Comparison

Platform Example	Secure	NSC	Non-secure	Total
Platform - PSA Crypto ECDH	64688 B	-	-	64688 B
tz_psa_crypto_ecdh_ws	79172 B	288 B	29264 B	108724 B

**Note:** The NSC is part of the Secure code, and the total size does not include the flash for NVM3 storage.

Table: RAM Size Comparison

Platform Example	Secure	NSC	Non-secure	Total
Platform - PSA Crypto ECDH	3784 B	-	-	3764 B
tz_psa_crypto_ecdh_ws	2156 B	-	1200 B	3356 B

**Note:** The total size does not include the RAM for the stack and heap. The Secure and Non-secure applications have their independent stack and heap.

## PSA Crypto Performance

The following sections compare the PSA Crypto performance of the [TrustZone-unaware](#) ( Platform - PSA Crypto ECDH ) and [TrustZone-aware](#) projects ( tz\_psa\_crypto\_ecdh\_ws ) based on the following conditions.

- The tz\_psa\_crypto\_ecdh\_ns reuses the source code from the Platform - PSA Crypto ECDH example without any changes.
- All source code is compiled with Optimize most (-O3) in Simplicity IDE (GNU ARM v10.3.1) of Simplicity Studio 5.
- Use ECC curve SECP256R1 on volatile and persistent keys.
- The EFR32MG22C224 runs at 38 MHz HFRCODPLL.

### Volatile key ECDH operation on Platform - PSA Crypto ECDH

```
. ECDH Client
+ Creating a SECP256R1 (256-bit) VOLATILE PLAIN client key... PSA_SUCCESS (cycles: 2928 time: 77 us)
+ Creating a SECP256R1 (256-bit) VOLATILE PLAIN server key... PSA_SUCCESS (cycles: 2960 time: 77 us)
+ Exporting a public key of a SECP256R1 (256-bit) VOLATILE PLAIN server key... PSA_SUCCESS (cycles: 332134 time: 8740 us)
+ Computing client shared secret with a SECP256R1 (256-bit) server public key... PSA_SUCCESS (cycles: 336860 time: 8864 us)
```

### Volatile key ECDH operation on tz\_psa\_crypto\_ecdh\_ws

```
. ECDH Client
+ Creating a SECP256R1 (256-bit) VOLATILE PLAIN client key... PSA_SUCCESS (cycles: 5047 time: 132 us)
+ Creating a SECP256R1 (256-bit) VOLATILE PLAIN server key... PSA_SUCCESS (cycles: 5067 time: 133 us)
+ Exporting a public key of a SECP256R1 (256-bit) VOLATILE PLAIN server key... PSA_SUCCESS (cycles: 333956 time: 8788 us)
+ Computing client shared secret with a SECP256R1 (256-bit) server public key... PSA_SUCCESS (cycles: 338470 time: 8907 us)
```

### Persistent key ECDH operation on Platform - PSA Crypto ECDH

```
. ECDH Client
+ Creating a SECP256R1 (256-bit) PERSISTENT PLAIN client key... PSA_SUCCESS (cycles: 27489 time: 723 us)
+ Creating a SECP256R1 (256-bit) PERSISTENT PLAIN server key... PSA_SUCCESS (cycles: 27587 time: 725 us)
+ Exporting a public key of a SECP256R1 (256-bit) PERSISTENT PLAIN server key... PSA_SUCCESS (cycles: 332949 time: 8761 us)
+ Computing client shared secret with a SECP256R1 (256-bit) server public key... PSA_SUCCESS (cycles: 337803 time: 8889 us)
```

### Persistent key ECDH operation on tz\_psa\_crypto\_ecdh\_ws

```
. ECDH Client
+ Creating a SECP256R1 (256-bit) PERSISTENT PLAIN client key... PSA_SUCCESS (cycles: 46998 time: 1236 us)
+ Creating a SECP256R1 (256-bit) PERSISTENT PLAIN server key... PSA_SUCCESS (cycles: 45962 time: 1209 us)
+ Exporting a public key of a SECP256R1 (256-bit) PERSISTENT PLAIN server key... PSA_SUCCESS (cycles: 334127 time: 8792 us)
+ Computing client shared secret with a SECP256R1 (256-bit) server public key... PSA_SUCCESS (cycles: 338321 time: 8903 us)
```

The overheads on the TrustZone-aware project ( `tz_psa_crypto_ecdh_ws` ) are due to the following operations of [Secure Library](#) implementation.

- Packages the list of input arguments in the appropriate format before calling into the NSC function.
- Switches from a Non-secure to a Secure state.
- Validates all input arguments before calling into the function in SPE.
- Encrypts PSA ITS if using a persistent key.
- Returns to a Non-secure state.

## Anti-Tamper Protection Configuration and Use

# Anti-Tamper Protection Configuration and Use

Note: This section replaces *AN1247: Anti-Tamper Protection Configuration and Use* . Further updates to this application note will be provided here.

This application note describes how to program, provision, and configure the anti-tamper module. Many aspects of the anti-tamper module, including disabling the anti-tamper response when needed, are discussed.

The anti-tamper module is only available on High devices. The external tamper detect module is available on some Secure Vault Mid devices (e.g. xG27) and Secure Vault High devices (e.g. xG25B).

## Key Points

- Tamper responses
- Tamper sources
- Tamper configuration
- Tamper disable
- Examples of provisioning and disabling the anti-tamper module

Series 2 Device Security Features

# Series 2 Device Security Features

Protecting IoT devices against security threats is central to a quality product. Silicon Labs offers several security options to help developers build secure devices, secure application software, and secure paths of communication to manage those devices. Silicon Labs’ security offerings were significantly enhanced by the introduction of the Series 2 products that included a Secure Engine. The Secure Engine is a tamper-resistant component used to securely store sensitive data and keys and to execute cryptographic functions and secure services.

On Series 1 devices, the security features are implemented by the TRNG (if available) and CRYPTO peripherals.

On Series 2 devices, the security features are implemented by the Secure Engine and CRYPTOACC (if available). The Secure Engine may be hardware-based, or virtual (software-based). Throughout this document, the following abbreviations are used:

- HSE - Hardware Secure Engine
- VSE - Virtual Secure Engine
- SE - Secure Engine (either HSE or VSE)

Additional security features are provided by Secure Vault. Three levels of Secure Vault feature support are available, depending on the part and SE implementation, as reflected in the following table:

Level (1)	SE Support	Part (2)
Secure Vault High (SVH)	HSE only (HSE-SVH)	Refer to <a href="#">IoT Endpoint Security Fundamentals</a> for details on supporting devices.
Secure Vault Mid (SVM)	HSE (HSE-SVM)	"
"	VSE (VSE-SVM)	"
Secure Vault Base (SVB)	N/A	"

Notes:

1. The features of different Secure Vault levels can be found in <https://www.silabs.com/security>.
2. [IoT Endpoint Security Fundamentals](#).

Secure Vault Mid consists of two core security functions:

- Secure Boot: Process where the initial boot phase is executed from an immutable memory (such as ROM) and where code is authenticated before being authorized for execution.
- Secure Debug access control: The ability to lock access to the debug ports for operational security, and to securely unlock them when access is required by an authorized entity.

Secure Vault High offers additional security options:

- Secure Key Storage: Protects cryptographic keys by "wrapping" or encrypting the keys using a root key known only to the HSE-SVH.
- Anti-Tamper protection: A configurable module to protect the device against tamper attacks.
- Device authentication: Functionality that uses a secure device identity certificate along with digital signatures to verify the source or target of device communications.

A Secure Engine Manager and other tools allow users to configure and control their devices both in-house during testing and manufacturing, and after the device is in the field.

## User Assistance

In support of these products Silicon Labs offers whitepapers, webinars, and documentation. The following table summarizes the key security documents:

Document	Summary	Applicability
<a href="#">Series 2 Secure Debug</a>	How to lock and unlock Series 2 debug access, including background information about the SE	Secure Vault Mid and High
<a href="#">Series 2 Secure Boot with RTSL</a>	Describes the secure boot process on Series 2 devices using SE	Secure Vault Mid and High
AN1222: Production Programming of Series 2 Devices	How to program, provision, and configure security information using SE during device production	Secure Vault Mid and High
Anti-Tamper Protection Configuration and Use (this document)	How to program, provision, and configure the anti-tamper module	Secure Vault High
<a href="#">Authenticating Silicon Labs Devices using Device Certificates</a>	How to authenticate a device using secure device certificates and signatures, at any time during the life of the product	Secure Vault High
<a href="#">Secure Key Storage</a>	How to securely 'wrap' keys so they can be stored in non-volatile storage.	Secure Vault High

## Key Reference

Public/Private keypairs along with other keys are used throughout Silicon Labs security implementations. Because terminology can sometimes be confusing, the following table lists the key names, their applicability, and the documentation where they are used.

Key Name	Customer Programmed	Purpose
Public Sign key (Sign Key Public)	Yes	Secure Boot binary authentication and/or OTA upgrade payload authentication
Public Command key (Command Key Public)	Yes	Secure Debug Unlock or Disable Tamper command authentication
OTA Decryption key (GBL Decryption key) aka AES-128 Key	Yes	Decrypting GBL payloads used for firmware upgrades
Attestation key aka Private Device Key	No	Device authentication for secure identity

## SE Firmware

Silicon Labs strongly recommends installing the latest SE firmware on Series 2 devices to support the required security features. Refer to [AN1222](#) for the procedure to upgrade the SE firmware and [IoT Endpoint Security Fundamentals](#) for the latest SE Firmware shipped with Series 2 devices and modules.



## Introduction

# Introduction

The HSE-SVH Anti-Tamper module is used to hamper or prevent both reverse engineering and re-engineering of proprietary software systems or applications.

Tamper attacks come from one or more vectors. Common attacks include voltage glitching, magnetic interference, and forced temperature adjustment. The HSE-SVH Anti-Tamper module provides fast hardware detection of external tamper signals such as case opening, glitching, and logical attacks allowing analysis and escalation up to and including bricking the device.

The anti-tamper module connects a number of hardware and software-driven tamper signals to a set of configurable hardware and software responses. This can be used to program the device to automatically respond to external events that could signal that someone is trying to tamper with the device, and very rapidly remove secrets stored in the HSE.

The available tamper signals range from signals based on failed authentication and secure boot to specialized glitch detectors. When any of these signals fire, the tamper block can be configured to trigger several different responses, ranging from triggering an interrupt to erasing the One-Time-Programmable (OTP) memory, removing all HSE secrets and resulting in a permanently destroyed device.

Silicon Labs provides [Custom Part Manufacturing Service \(CPMS\)](#) to protect the users' privacy by configuring the most effective tamper detection features at the Silicon Labs factory. For more information about CPMS, see the [Custom Part Manufacturing Service User's Guide](#).

Some SVM devices (e.g. xG25A and xG27) and SVH devices (e.g. xG25B) feature an External Tamper Detect module which is used to detect signals such as case opening. The ETAMPDET signal on SVH devices is routed to the SE as an Anti-Tamper module tamper source, in addition to being a stand-alone module. For more information about ETAMPDET operation, refer to the device reference manual. Examples demonstrating how to use ETAMPDET can be found on the [Silicon Labs peripheral\\_example github repository](#).

## Secure Engine Manager

# Secure Engine Manager

The Secure Engine Manager provides thread-safe APIs for the SE's mailbox interface. The SE Manager APIs related to tamper operations are listed in the following table.

For the SE's mailbox interface, see section *Secure Engine Subsystem* in [Series 2 Secure Debug](#).

SE Manager API	Usage
sl_se_init_otp	Initialize SE OTP configuration (including tamper configuration on HSE-SVH devices).
sl_se_read_otp	Read SE OTP configuration (including tamper configuration on HSE-SVH devices).
sl_se_init_otp_key	Used during device initialization to upload the Public Command Key.
sl_se_read_pubkey	Read the stored Public Command Key.
sl_se_get_serialnumber	Read out the serial number (16 bytes) of the HSE device.
sl_se_get_challenge	Read out the current challenge value (16 bytes) for tamper disable.
sl_se_roll_challenge	Used to roll the current challenge value (16 bytes) to invalidate the Disable Tamper Token.
sl_se_disable_tamper	Temporarily disable tamper configuration using the Disable Tamper Token.
sl_se_get_status	Read the current HSE status (including recorded tamper status on HSE-SVH devices).
sl_se_get_reset_cause	Read the EMU->RSTCAUSE register from HSE devices after a tamper reset.
sl_se_get_tamper_reset_cause	Read the cached value of the EMU->TAMPERRSTCAUSE register after a tamper reset.
sl_se_enter_active_mode	Force the SE to remain active to enable the detection of glitch tamper events on the host Cortex-M33 core.(see fourth note below)
sl_se_exit_active_mode	Exit active mode and allow the SE to sleep when not performing operations. This will prevent the detection of glitch tamper events when the SE is sleeping. This API should only be used if active mode was entered by calling sl_se_enter_active_mode. If active mode is set through a DCI command, it can only be disabled through a DCI command. (see fourth note below)

### Notes:

- The `sl_se_get_reset_cause` is only available on EFR32xG21B devices. The `EMU->RSTCAUSE` register can be directly read on other HSE-SVH devices.
- The `sl_se_get_tamper_reset_cause` is unavailable on EFR32xG21B devices, and SE firmware  $\geq$  v2.2.1 is required.
- The SE Manager API document can be found at <https://docs.silabs.com/gecko-platform/latest/service/api/group-sl-se-manager>.
- Does not apply to EFR32MG21B parts.

# Tamper Responses

# Tamper Responses

A [tamper source](#) can lead to a series of different autonomous responses from the HSE. These responses are listed in the following table.

Level	Response	Description
0	Ignore	No action is taken
1	Interrupt	Triggers the SETAMPERHOST interrupt on the host
2	Filter	Increases a counter in the tamper filter
4	Reset	Resets the device
7	Erase OTP	Erases the device's OTP configuration

- Notes:
- Level 3, 5, and 6 are reserved.
  - These responses are cumulative:
    - If a filter response is triggered, it will also trigger an interrupt.
    - If a reset response is triggered, it will supersede the interrupt. The [filter counter](#) and interrupt flag are clear at reset.
    - If an erase OTP response is triggered, it will erase the OTP and reset the device. The device will fail to boot and become unusable.

## Interrupt

If a tamper source is configured to respond with the interrupt response or higher ( $\geq$  level 1), the `SETAMPERHOST` interrupt line to the host Cortex-M33 will be pulsed and make the NVIC trigger the corresponding interrupt handler (`SETAMPERHOST_IRQHandler`).

After the interrupt has been handled, the tamper status can be found by reading the HSE status (using `sl_se_get_status` in the [SE Manager](#)), which contains a list of all the tamper sources that have been triggered since the last time the status was read. Reading HSE status clears the registered tamper sources.

**Note:** Enabling the `SEMAILBOXHOST` clock for the tamper source is required to trigger the `SETAMPERHOST` interrupt in most HSE-SVH devices. EFR32xG21B does not require this.

## Filter

The HSE has a filter to debounce spurious tamper events. The filter has a counter that is periodically reset. If a tamper source is configured to the filter response (level 2), when it is triggered, the counter is increased. If the counter value reaches a configurable threshold, the `Filter counter` tamper source ([number 1](#)) is triggered, which can configure to lead to any other responses (1, 4, or even 7).

Only a single shared filter counter is available, so the cumulative triggering of all tamper sources configured to the filter level will increase the same counter. The filter can be programmed to use one of the trigger thresholds and reset periods provided below. The filter counter is zero upon a tamper or normal reset.

### Filter Trigger Threshold

- Value (n): 0 to 7
- Filter Trigger Threshold:  $256/2^n$  (256 to 2)

### Filter Reset Period

- Value (n): 0 to 31
- Filter Reset Period:  $32 \text{ ms} * 2^n$  (32 ms to ~795.4 days)

### Example Filter Configuration

For example, consider a device with a Filter Trigger Threshold of 3 and Filter Reset Period of 5. If that device detects 32 ( $256/2^3$ ) Filter response events in 1.024 seconds ( $32 \text{ ms} * 2^5$ ), the `Filter counter` tamper source ([number 1](#)) will trigger.

## Reset

The reset response resets the HSE and Cortex-M33. After a tamper reset, the last reset cause can be directly read from `EMU->RSTCAUSE` register or using `sl_se_get_rstcause` in the [SE Manager](#). In cases where the reset was caused by a tamper response, the source of the tamper can be determined by calling `sl_se_get_tamper_reset_cause` in the [SE Manager](#). (Note that this API is not available for EFR32xG21B-based parts). See Table Tamper Sources on Other HSE-SVH Devices for the list of tamper sources. Tamper reset occurs when the HSE sends a request to the Cortex-M33's EMU, which issues a hard reset.

If a tamper reset is triggered during boot, this can lead to a boot loop. To debug such a scenario, the HSE has a tamper reset counter and enters diagnostic mode if the counter reaches a [programmable threshold](#). Users can issue a non-tamper reset to clear the tamper reset counter before the programmable threshold is reached.

In diagnostic mode, the Cortex-M33 is held in reset and only DCI commands are available. The device will remain in diagnostic mode until a power-on or pin reset occurs.

For more information on the SE's DCI, see section *Secure Engine Subsystem* in [Series 2 Secure Debug](#).

## Erase OTP

The Erase OTP response is the strongest reaction the HSE can take, and it will make the device and all wrapped secrets unrecoverable. After this response, the device will no longer be able to boot or connect to a debugger.

This response should typically only be used in situations where the device believes that it is under an actual attack, for instance through the detection of several voltage or digital glitches in a short time window.

## Tamper Sources

# Tamper Sources

The following tables list the available tamper sources and the default level on the EFR32xG21B and other HSE-SVH devices. The tamper sources with the default level higher than 0 ( Ignore ) are always in effect even if the user does not initialize the tamper configuration in HSE OTP. Users can keep or escalate the default tamper responses ( $\geq 0$  for Ignore and  $\geq 4$  for Reset ) of any sources when initially configuring the part.

**Table:** Tamper Sources on the EFR32xG21B Devices

Type	Number	Name	Description	Default Level
SE Hardware	0	Reserved	-	-
"	1	Filter counter	Filter counter reached the configured threshold value	0 (Ignore)
"	2	SE watchdog	Internal SE watchdog expired	4 (Reset)
"	3	Reserved	-	-
"	4	SE RAM CRC	A 2-bit, non-correctable error in the SE RAM has occurred.	4 (Reset)
"	5	SE hard fault	The SE core has encountered a hard fault exception indicating that an invalid memory access was attempted.	4 (Reset)
"	6	Reserved	-	-
SE Software	7	SE software assertion	SE firmware has triggered an assertion, indicating that one of several sanity checks has failed and that normal operation cannot continue without a reset.	4 (Reset)
"	8	SE secure boot	Secure boot of SE firmware failed	4 (Reset)
"	9	User secure boot	Secure boot of host firmware failed	0 (Ignore)
"	10	Mailbox authorization	Unauthorized command received over the Mailbox interface. This can be triggered by either (1) an incorrectly signed debug unlock or tamper disable token or (2) attempting to export a non-exportable key.	0 (Ignore)
"	11	DCI authorization	Unauthorized command received over the DCI interface. This can be triggered by either (1) an incorrectly signed debug un-lock or tamper disable token or (2) attempting to export a non-exportable key.	0 (Ignore)
"	12	OTP read	OTP or flash content could not be properly authenticated.	4 (Reset)
"	13	Reserved	-	-
"	14	Self test	A check of the integrity of the SE's internal storage failed during boot up.	4 (Reset)
"	15	TRNG monitor	The TRNG monitor performs a number of tests on the collected entropy data. If any of these tests fail, this tamper source is triggered.	0 (Ignore)
Hardware	16 - 23	PRS0 - 7 [1]	PRS consumer for SE Tamper 0 - 7 asserted	0 (Ignore)

Type	Number	Name	Description	Default Level
"	24	Decouple BOD [1]	Decouple Brown-Out-Detector threshold alert	4 (Reset)
"	25	Temperature sensor [1]	SE temperature is monitored to be within 5 degrees C of the limits for the device. If the limit is exceeded, this tamper source will be triggered.	0 (Ignore)
"	26	Voltage glitch falling	Voltage glitch detector detected a falling glitch	0 (Ignore)
"	27	Voltage glitch rising	Voltage glitch detector detected a rising glitch	0 (Ignore)
"	28	Secure lock	This tamper source indicates that the guarding mechanism (comparing the locks with their logical complement) of the debug port locks has failed	4 (Reset)
"	29	SE debug	Debug access to SE	0 (Ignore)
"	30	Digital glitch	Digital glitch detector detected an event	0 (Ignore)
"	31	SE ICACHE	The SE's instruction cache uses a checksum to verify the integrity of the data. This tamper source is triggered if the checksum is invalid.	4 (Reset)

**Table:** Tamper Sources on Other HSE-SVH Devices

Type	Number	Name	Description	Default Level
SE Hardware	0	Reserved	-	-
"	1	Filter counter	Filter counter reached the configured threshold value	0 (Ignore)
"	2	SE watchdog	Internal SE watchdog expired	4 (Reset)
"	3	Reserved	-	-
"	4	SE RAM ECC2	A 2-bit, non-correctable error in the SE RAM has occurred.	4 (Reset)
"	5	SE hard fault	The SE core has encountered a hard fault exception indicating that an invalid memory access was attempted.	4 (Reset)
"	6	Reserved	-	-
SE Software	7	SE software assertion	SE firmware has triggered an assertion, indicating that one of several sanity checks has failed and that normal operation cannot continue without a reset.	4 (Reset)
"	8	SE secure boot	Secure boot of SE firmware failed	4 (Reset)
"	9	User secure boot	Secure boot of host firmware failed	0 (Ignore)
"	10	Mailbox authorization	Unauthorized command received over the Mailbox interface. This can be triggered by either (1) an incorrectly signed debug unlock or tamper disable token or (2) attempting to export a non-exportable key.	0 (Ignore)
"	11	DCI authorization	Unauthorized command received over the DCI interface. This can be triggered by either (1) an incorrectly signed debug unlock or tamper disable token or (2) attempting to export a non-exportable key.	0 (Ignore)
"	12	OTP read	OTP or flash content could not be properly authenticated	4 (Reset)
"	13	Reserved	-	-

Type	Number	Name	Description	Default Level
"	14	Self test	A check of the integrity of the SE's internal storage failed during boot up.	4 (Reset)
"	15	TRNG monitor	The TRNG monitor performs a number of tests on the collected entropy data. If any of these tests fail, this tamper source is triggered.	0 (Ignore)
Hardware	16	Secure lock	This tamper source indicates that the guarding mechanism (comparing the locks with their logical complement) of the debug port locks has failed.	4 (Reset)
"	17	Digital glitch	Digital Glitch detector detected an event	0 (Ignore)
"	18	Voltage glitch	Voltage Glitch Detector detected an event	0 (Ignore)
"	19	SE ICACHE	The SE's instruction cache uses a checksum to verify the integrity of the data. This tamper source is triggered if the checksum is invalid.	4 (Reset)
"	20	SE RAM ECC1	SE RAM 1-bit ECC error occurred	0 (Ignore)
"	21	BOD [1]	Brown-Out-Detector threshold alert	4 (Reset)
"	22	Temperature sensor [1]	SE temperature is monitored to be within 5 degrees C of the limits for the device. If the limit is exceeded, this tamper source will be triggered.	0 (Ignore)
"	23	DPLL fall	DPLL lock failed low	0 (Ignore)
"	24	DPLL rise	DPLL lock failed high	0 (Ignore)
"	25	PRS0 or ETAMPDET	PRS consumer for SE Tamper 25 or ETAMPDET asserted.	0 (Ignore)
"	26 - 31	PRS1 - 6 or PRS0 - 5[1]	PRS consumer for SE Tamper 26 - 31 asserted	0 (Ignore)

**Notes:**

- [1] These tamper sources are available down to EM2. Other sources are available in EM1 and above.
- In EFR32xG21B devices, hardware tamper sources 24 to 27 can operate down to Energy Mode 3 (EM3), whereas other hardware tamper sources (16 - 23 and 28 - 31) can be active down to Energy Mode 1 (EM1).
- In other HSE-SVH devices, tamper sources 25 to 31 are used for External Tamper Detect (ETAMPDET) if present and PRS consumers. Devices with ETAMPDET (e.g. EFR32xG25B) will have 6 PRS consumers (26 to 31) and devices without ETAMPDET will have 7 PRS consumers (25 to 31).
- The **ETAMPDET** source gets triggered when any of the ETAMPDET channels are asserted.
- [User configuration](#) or [tamper disable](#) cannot reduce the tamper response below the default Level.
- The **User secure boot** source gets triggered if secure boot is enabled and host image verification fails. It is likely to put the device in the boot loop if users escalate the tamper response of this source to 4 (Reset).
- The **Mailbox and DCI authorizations** get triggered whenever one of the following conditions has occurred. The [mailbox](#) returns `SE_RESPONSE_AUTHORIZATION_ERROR`, and [DCI](#) returns `AUTH_ERROR = 2`.
  1. A mailbox or DCI command tries to exercise a key that it is not allowed to use (e.g., trying to export a non-exportable key).
  2. A [secure debug](#) access or [tamper disable](#) request over the mailbox or DCI is invalidly signed.
  3. A malformed HSE firmware upgrade over the mailbox or DCI is attempted.
- The **OTP read** gets triggered if there is an issue when decrypting and authenticating settings in HSE OTP or flash.
- The HSE has redundancy built into the locking mechanism, and the **Secure lock** source is used to detect errors in that redundancy.
- PRS inputs can allow user applications to implement additional tamper sources and feed them into the tamper response mechanism. The **PRS** tamper sources are under the control of the user application and could be reconfigured or disabled if the user application is compromised.
- The **Temperature sensor** source is not completely accurate and is generally only suitable for systems that expect to stay well within the specified temperature range. Users requiring a tighter temperature limit can implement their temperature

monitor and provide the results as a tamper source via PRS.

- On EFR32xG23B and later devices, the default behavior is to detect tamper events only when the SE core is active. To detect tamper events when the SE is not performing operations, call `sl_se_enter_active_mode()`. This prevents the SE from sleeping and will result in higher current draw.



Anti-Tamper Configuration

# Anti-Tamper Configuration

The user can provision the anti-tamper configuration in HSE OTP detailed in the following table through `sl_se_init_otp` in the [SE Manager](#). [Tamper configurations](#) must be programmed with secure boot settings and are immutable once written.

For more information on enabling the OTP tamper configuration along with the secure boot settings, see *Enabling Secure Boot and Tamper Configuration* in [AN1222: Production Programming of Series 2 Devices](#).

Setting	Description
Tamper response levels	A user response level for each tamper source (1)
Filter settings	The tamper filter counter has two settings: trigger threshold and reset period
Digital Glitch Detector Always On	Bit 1 of tamper flag: 0 – Digital glitch detector runs only when the HSE is executing a command; 1 – Digital glitch detector runs even when the HSE is not performing any operations (note that this leads to increased energy consumption)
Keep Tamper Alive During Sleep(2)	Bit 2 of tamper flag: 0 – The tamper module stops running in sleep mode; 1 – The tamper module keeps running in sleep mode (down to EM3)
Reset threshold	The number of consecutive tamper resets (up to 255) before the part enters diagnostic mode (3)

Notes:

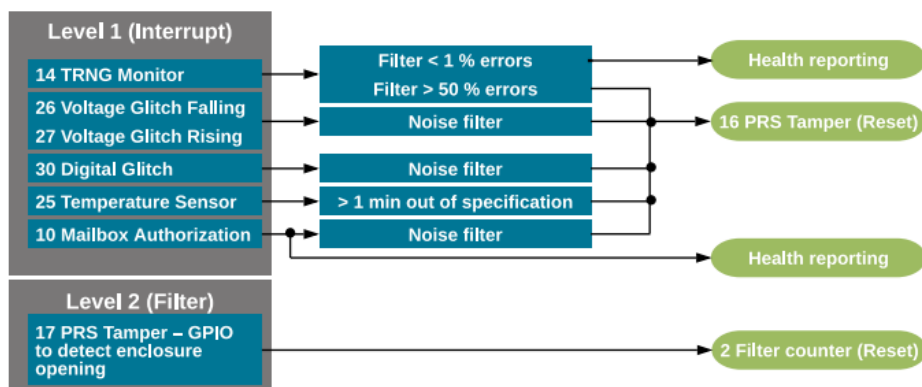
- 1. The effective response of a tamper source is the maximum value between the [default level](#) and user level ( `Active level = MAX(default level, user level)` ). If the user sets the response of a tamper source to a level lower than the default level, the setting will still be saved to HSE OTP but does not take any effect. The HSE returns the user levels instead of active levels of all tamper sources when reading back ( `sl_se_read_otp` ) the tamper configuration from the HSE OTP.
- 2. This flag is not available on EFR32xG21B devices.
- 3. If the threshold is set to 0, the part will never enter the diagnostic mode due to tamper reset.

## Usage Example

# Usage Example

Several of the available [tamper sources](#) report internal HSE errors. A number of these sources are configured to reset the device (level 4) by default. Custom handling of internal and external tamper sources (default level 0) can be configured to trigger an interrupt (level 1) on the Cortex-M33 or trigger an interrupt and increase a counter in the tamper filter (level 2) as in the following figure for EFR32xG21B devices.

**Figure:** Custom Handling of Tamper Sources (EFR32xG21B Devices)



**Note:** The actions for level 1 on the right side are implemented by the tamper interrupt handler.

### Usage example highlights:

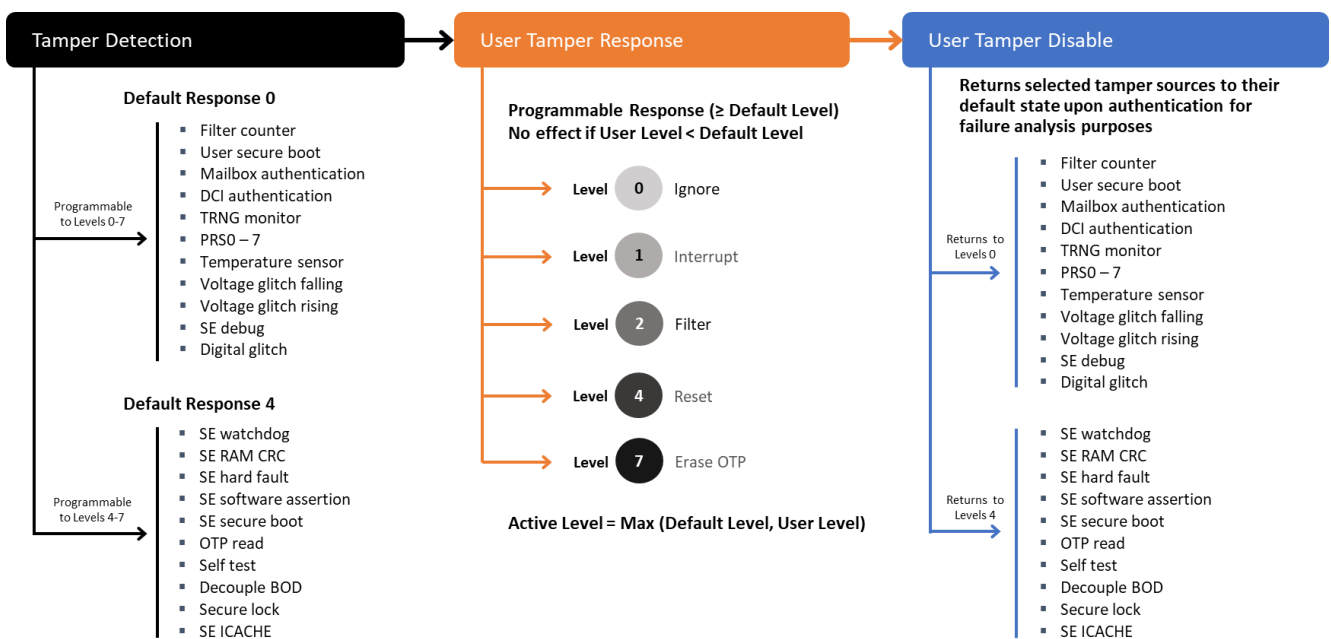
- The response of the TRNG monitor depends on the failure rate due to lack of entropy.
- The voltage and digital glitch detectors can see spurious activations. They should typically not be used to drive a high-level tamper response directly. Instead, they should feed their signals into a tamper interrupt, which activates a high-level action (e.g., Reset in this example) through PRS tamper if a certain number of detections (noise filter) occur in a short time window.
- The operating conditions decide the time out of the specification filter for the temperature sensor. For some systems, any time out of specification should trigger a reset.
- Mailbox authorization is handled similarly for voltage and digital glitch detectors.
- A PRS tamper implements a high-level response for a tamper interrupt, which issues a tamper reset (level 4) to prevent or slow further attacks.
- In extreme cases, if the system identifies an attack with high confidence, a PRS tamper can be configured as [Erase OTP](#) (level 7) to brick the part and prevent further attacks. This is recommended only when the destruction of parts is acceptable and where high confidence of an attack can be achieved.
- Another PRS tamper detects enclosure opening from GPIO. This source feeds into the tamper filter counter (level 2), which will trigger an interrupt ([cumulative effect](#)) and activate a [Filter counter](#) ([number 1](#)) response (Reset in this example) if the filter counter reaches the [trigger threshold](#) within the [filter reset period](#). This filter counter response approach is less flexible than the interrupt response approach since the trigger threshold and filter reset period are one-time programmable.

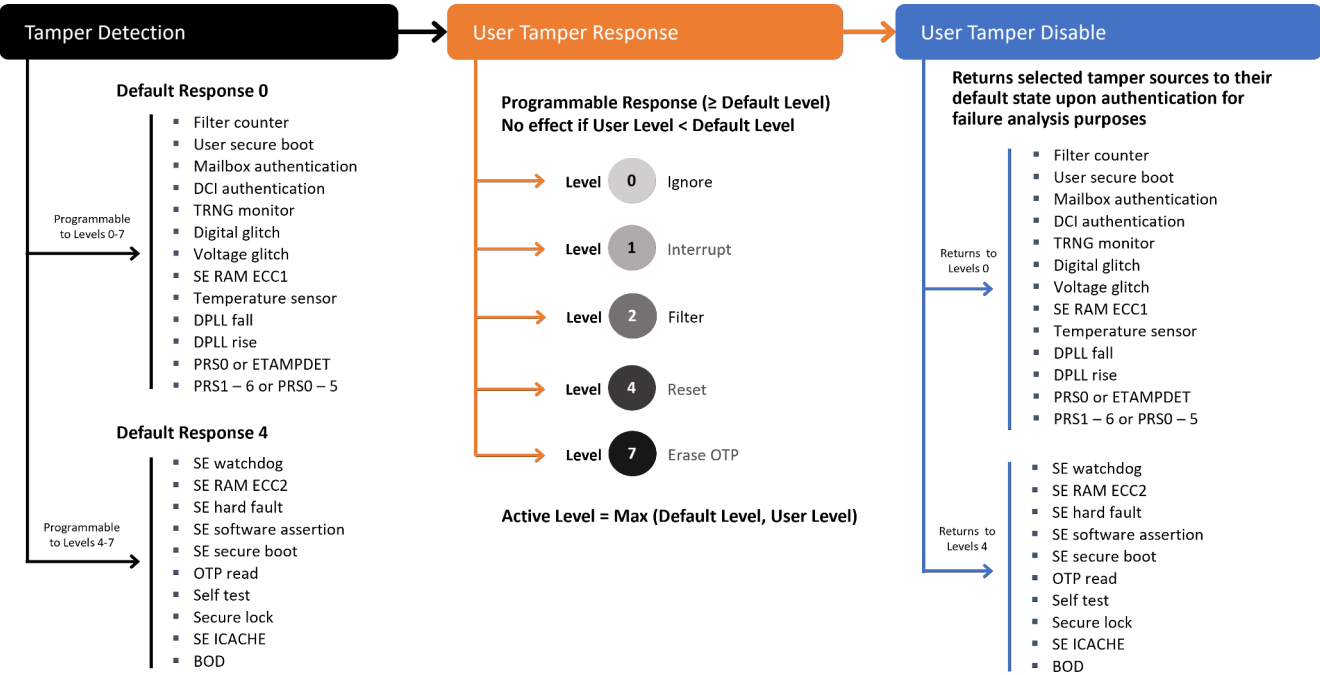
## Tamper Disable

# Tamper Disable

For diagnostic purposes, it may be necessary to disable the tamper response. For example, if a user has configured the part to [Erase OTP](#) on external tamper detection, disabling the tamper response is required to open the unit and perform failure analysis or field service activities.

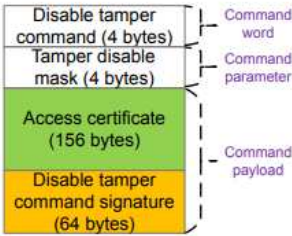
After the [tamper configuration](#) has been initialized, users can temporarily restore the tamper response to default for a set of [tamper sources](#) via a [Disable Tamper Token](#) authenticated against the Public Command Key in HSE OTP (similar to secure debug unlock). This is only possible if the [Public Command Key](#) has been provisioned in the device.





Disable Tamper Token

The elements of the Disable Tamper Token are described in the following figures and table.



Element	Value	Description
Disable tamper command	0xfd020001	The command word of the Disable Tamper Token.
Tamper disable mask	Device-dependent	The command parameter of the Disable Tamper Token.
Access certificate (1)	Device-dependent	See section Access Certificate.
Disable tamper command signature (1)	Device-dependent	See section Challenge Response.

Note:

1. The disable tamper command payload consists of an [access certificate](#) and a [disable tamper command signature](#).

Tamper Disable Mask		
Name	Bit	
Tamper source 31	31	
Tamper source 30	30	
Tamper source 29	29	
Tamper source 28	28	
Tamper source 27	27	
Tamper source 26	26	
Tamper source 25	25	
Tamper source 24	24	
Tamper source 23	23	
Tamper source 22	22	
Tamper source 21	21	
Tamper source 20	20	
Tamper source 19	19	
Tamper source 18	18	
Tamper source 17	17	
Tamper source 16	16	
Tamper source 15	15	
Tamper source 14	14	
Tamper source 13	13	
Tamper source 12	12	
Tamper source 11	11	
Tamper source 10	10	
Tamper source 9	9	
Tamper source 8	8	
Tamper source 7	7	
Tamper source 6	6	
Tamper source 5	5	
Tamper source 4	4	
Tamper source 3	3	
Tamper source 2	2	
Tamper source 1	1	
Tamper source 0	0	

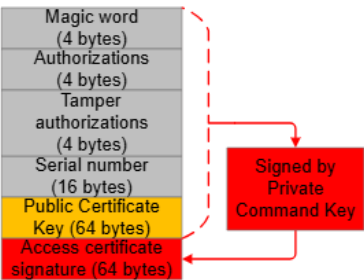
**Note:** Set bit to restore the default response of the corresponding tamper source.

The Disable Tamper Token temporarily reverts all masked tamper sources in the figure above to the hard-coded configuration ([Figure 8.1 Tamper Disable on the EFR32xG21B Devices on page 13](#) and [Figure 8.2 Tamper Disable on Other HSE-SVH Devices on page 13](#)).

The Disable Tamper Token can only restore the escalated user-level configuration to default. It cannot degrade the default level of a tamper source.

Access Certificate

The elements of the access certificate are described in the following figure and table.



Element	Value	Description
Magic word	0xe5ecce01	A constant value used to identify the access certificate.
Authorizations	0x0000003e (1)	A value used to authorize which bit in the debug mode request can be enabled for secure debug.
Tamper Authorizations	0xfffffb6 (2)	A value used to authorize which bit in the tamper disable mask can be set to disable the tamper response.
Serial number	Device-dependent	A number used to compare against the on-chip serial number for secure debug or tamper disable.
Public Certificate Key (3)	Device-dependent	The public key corresponding to the Private Certificate Key (3) used to generate the signature (ECDSA-P256-SHA256) in a challenge response.

Element	Value	Description
Access certificate signature	Device-dependent	All the content above is signed (ECDSA-P256-SHA256) by the Private Command Key corresponding to the Public Command Key in the HSE OTP.

Notes:

- 1. The value allows all debug options to be reset for secure debug. Note that the commands for debug unlock and tamper disable are separate, so the secure debug lock will not be disabled when issuing a tamper disable command.
- 2. Value that sets available bits in the tamper disable mask for tamper disable.
- 3. The Private/Public Certificate Key is a randomly generated key pair. It can be ephemeral or retainable.

The Private Certificate Key can be used repeatedly to generate the signature in a [challenge response](#) on one device until the Private/Public Certificate Key pair is discarded. This can reduce the frequency of access to the Private Command Key, allowing more restrictive access control on that key.

For more information about secure debug, see [Series 2 Secure Debug](#).

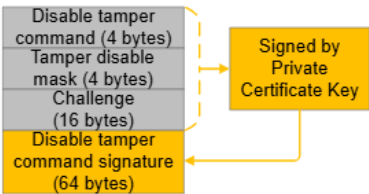
Tamper Authorizations	
Name	Bit
Tamper disable mask 31	31
Tamper disable mask 30	30
Tamper disable mask 29	29
Tamper disable mask 28	28
Tamper disable mask 27	27
Tamper disable mask 26	26
Tamper disable mask 25	25
Tamper disable mask 24	24
Tamper disable mask 23	23
Tamper disable mask 22	22
Tamper disable mask 21	21
Tamper disable mask 20	20
Tamper disable mask 19	19
Tamper disable mask 18	18
Tamper disable mask 17	17
Tamper disable mask 16	16
Tamper disable mask 15	15
Tamper disable mask 14	14
Tamper disable mask 13	13
Tamper disable mask 12	12
Tamper disable mask 11	11
Tamper disable mask 10	10
Tamper disable mask 9	9
Tamper disable mask 8	8
Tamper disable mask 7	7
Tamper disable mask 6	6
Tamper disable mask 5	5
Tamper disable mask 4	4
Tamper disable mask 3	3
Tamper disable mask 2	2
Tamper disable mask 1	1
Tamper disable mask 0	0

Notes:

- Set the bit to enable the corresponding bit in the [tamper disable mask](#).
- The Disable Tamper Token will restore the default response of the corresponding [tamper source](#) if the same bit is set in the tamper disable mask and tamper authorizations.

Challenge Response

The elements of the challenge response are described in the following figure and table.



Element	Value	Description
Disable tamper command	0xfd020001	The command word of Disable Tamper Token.
Tamper disable mask	Device-dependent	The command parameter of Disable Tamper Token.
Challenge	Device-dependent (1)	A random value generated by the HSE.

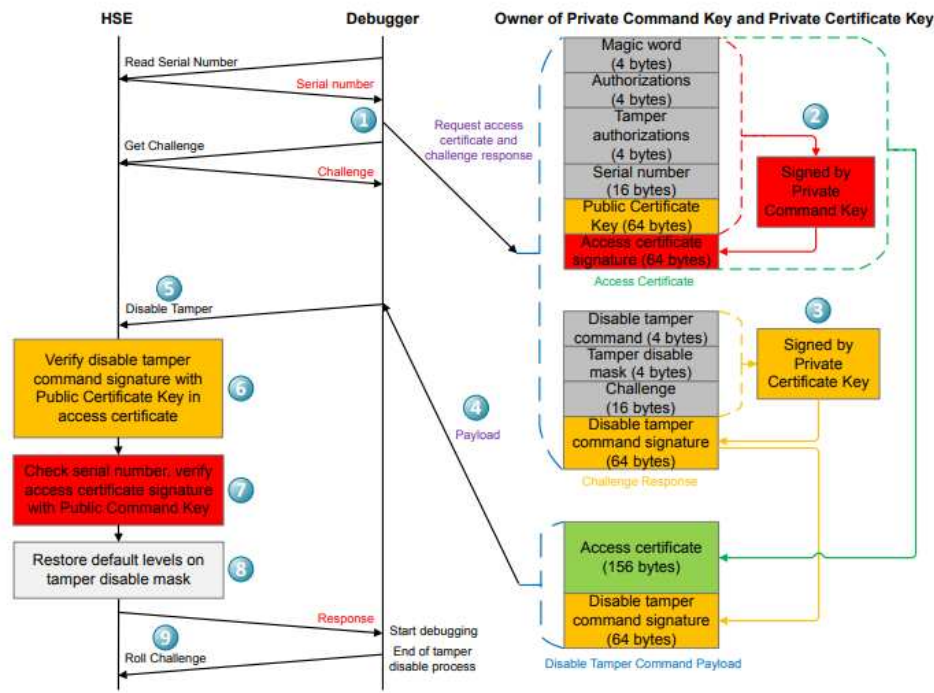
Element	Value	Description
Disable tamper command signature	Device-dependent (2)	All the content above is signed (ECDSA-P256-SHA256) by the Private Certificate Key corresponding to the Public Certificate Key in the access certificate.

Notes:

- 1. The challenge remains unchanged until it is updated to a new random value by [rolling the challenge](#). The Private Certificate Key can be reused for signing when device challenge is refreshed.
- 2. This signature is the final argument of the [Disable Tamper Token](#).

Tamper Disable Flow

The tamper disable flow is described in the following figure.



1. Get the serial number and challenge from the HSE.
2. Generate the [access certificate](#) with device serial number.
3. Generate the [challenge response](#) with device challenge.
4. Generate the disable tamper command payload with access certificate and disable tamper command signature.
5. Send the [Disable Tamper Token](#) to the HSE.
6. Verify the disable tamper command signature using the Public Certificate Key in the access certificate.
7. Verify the serial number and the access certificate signature using the on-chip serial number and Public Command Key in the HSE OTP.
8. Restore default levels on [tamper disable mask](#) until the next power-on or pin reset.
9. [Roll the challenge](#) to invalidate the current Disable Tamper Token.

**Note:** Refer to the [Simplicity Commander example](#) for details on how to follow this flow using Simplicity Commander.



Examples

# Examples

## Overview

The examples for HSE-SVH Anti-Tamper module are described in the following table.

Example	Device (Radio Board)	HSE Firmware	Tool
Provision Tamper configuration	EFR32MG21B010F1024IM32 (BRD4181C)	Version 1.2.9	SE Manager
Provision Public Command Key & Tamper configuration	EFR32MG21B010F1024IM32 (BRD4181C)	Version 1.2.9	Simplicity Commander
"	EFR32MG21B010F1024IM32 (BRD4181C)	Version 1.2.9	Simplicity Studio 5
Tamper disable and Roll challenge	EFR32MG21B010F1024IM32 (BRD4181C)	Version 1.2.9	SE Manager
"	EFR32MG21B010F1024IM32 (BRD4181C)	Version 1.2.9	Simplicity Commander
Roll challenge	EFR32MG21B010F1024IM32 (BRD4181C)	Version 1.2.9	Simplicity Studio 5

**Note:** Unless specified in the example, these examples can be applied to other HSE-SVH devices.

## Using a Platform Example

Simplicity Studio 5 includes the [SE Manager platform example](#) for tamper. This application note uses platform examples of GSDK v4.1.0. The console output may be different on the other version of GSDK.

Refer to the corresponding readme file for details about each SE Manager platform example. This file also includes the procedures to create the project and run the example.

## Using Simplicity Commander

1. This application note uses Simplicity Commander v1.14.6. The procedures and console output may be different on other versions of Simplicity Commander. The latest version of Simplicity Commander can be downloaded from .

```
commander --version

Simplicity Commander 1v14p6b1289

JLink DLL version: 7.70d
Qt 5.12.10 Copyright (C) 2017 The Qt Company Ltd.
EMDLL Version: 0v18p9b677
mbed TLS version: 2.16.6

DONE
```
2. The Simplicity Commander's Command Line Interface (CLI) is invoked by `commander.exe` in the Simplicity Commander folder. The location for Simplicity Studio 5 in Windows is `C:\SiliconLabs\SimplicityStudio\v5\developer\adapter_packs\commander` . For ease of use, it is highly recommended to add the path of `commander.exe` to the system `PATH` in Windows.



If more than one Wireless Starter Kit (WSTK) is connected via USB, the target WSTK must be specified using the `--serialno \<J-Link serial number>` option.

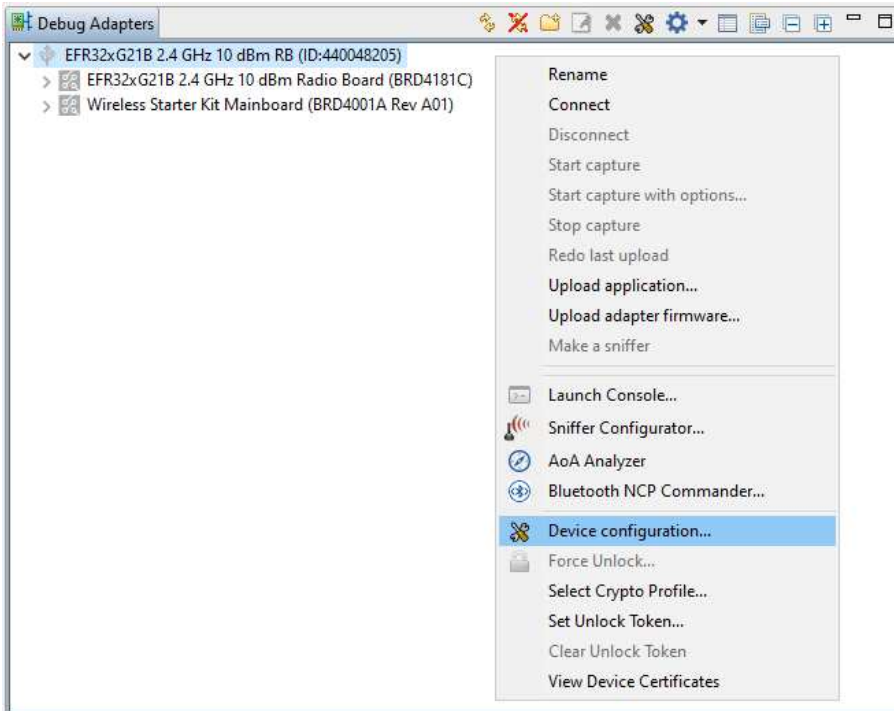
4. If the WSTK is in debug mode OUT, the target device must be specified using the `--device \<device name>` option.

For more information about Simplicity Commander, see the [Simplicity Commander Reference Guide](#).

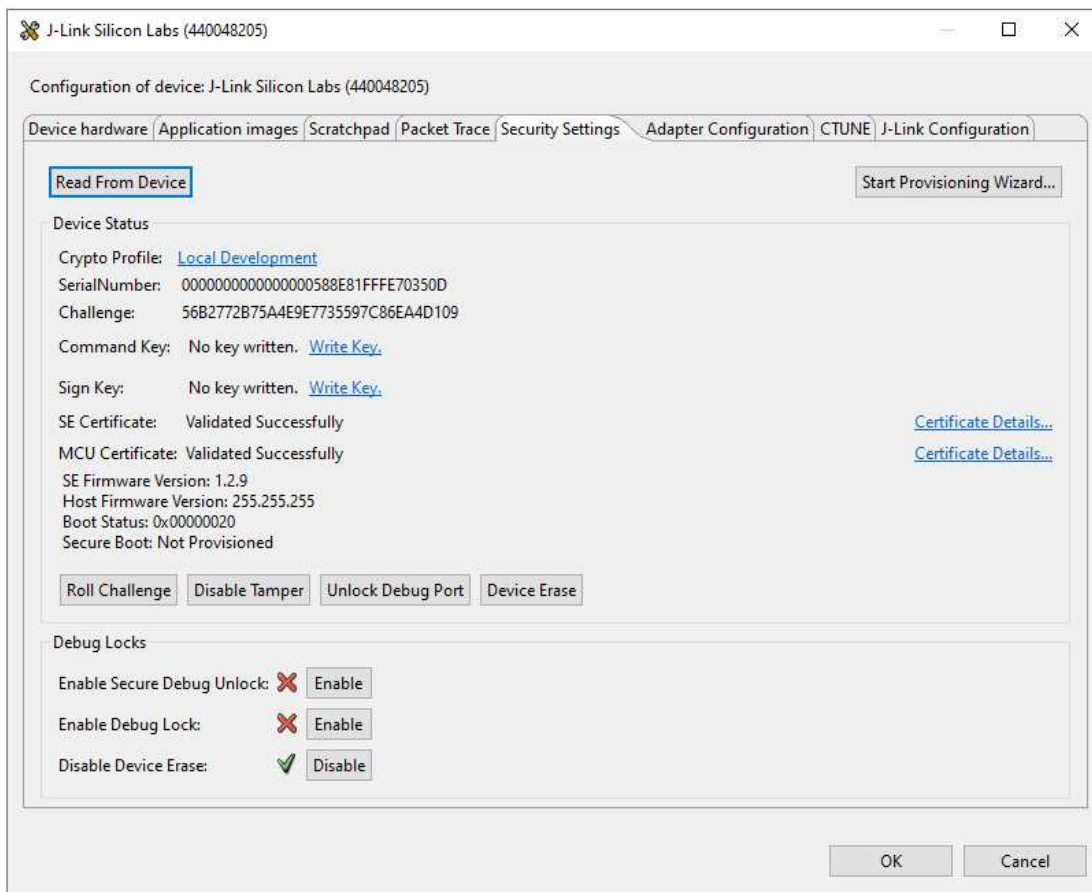
## Using Simplicity Studio

The security operations are performed in the Security Settings of Simplicity Studio. This application note uses Simplicity Studio v5.4.0.0. The procedures and pictures may be different on the other version of Simplicity Studio 5.

1. Right-click the selected debug adapter **RB (ID:J-Link serial number)** to display the context menu.



2. Click **Device configuration...** to open the **Configuration of device: J-Link Silicon Labs (serial number)** dialog box. Click the **Security Settings** tab to get the selected device configuration.



## Using an External Tool

The [tamper disable example](#) uses OpenSSL to sign the [access certificate](#) and [challenge response](#). The Windows version of OpenSSL can be downloaded from [Shining Light Productions](#). This application note uses OpenSSL Version 1.1.1h (Win64).

openssl version

OpenSSL 1.1.1h 22 Sep 2020

The OpenSSL's Command Line Interface (CLI) is invoked by `openssl.exe` in the OpenSSL folder. The location in Windows (Win64) is `C:\Program Files\OpenSSL-Win64\bin`. For ease of use, it is highly recommended to add the path of `openssl.exe` to the system `PATH` in Windows.

## Provision Public Command Key and Tamper Configuration

The Public Command Key pair can be generated from the "unsafe" private key delivered with Simplicity Studio, by Simplicity Commander, or by a Hardware Security Module (HSM). Using an HSM is recommended for production systems.

### Generated from "Unsafe" Key

External tools such as openssl can be used to generate a public key from the reference private key provided in Simplicity Studio. Note that this private key is well known and should not be used in production devices.

Run the `openssl ec` command to generate the Public Command Key from the Private Command Key.

```
openssl ec -in /c/SiliconLabs/SimplicityStudio/v5/developer/adapter_packs/secmgr/scripts/offline/cmd-unsafe-privkey.pem -pubout -out cmd-unsafe-pubkey.pem
```

Generated Using Simplicity Commander

Run the util genkey command to generate the Public Command Key pair (command\_key.pem and command\_pubkey.pem) and Public Command Key token file (command\_pubkey.txt).

```
commander util genkey --type ecc-p256 --privkey command_key.pem --pubkey command_pubkey.pem --tokenfile command_pubkey.txt
```

```
Generating ECC P256 key pair...
Writing private key file in PEM format to command_key.pem
Writing public key file in PEM format to command_pubkey.pem
Writing EC tokens to command_pubkey.txt...
DONE
```

## SE Manager - Tamper Platform Example

Click the [View Project Documentation](#) link to open the [readme](#) file for instructions on creating the project and running the example.

### Platform - SE Manager Tamper

This example project demonstrates the tamper feature of Secure Vault High device.

[CREATE](#)[View Project Documentation](#)

1. Press **ENTER** two times to program the secure boot and tamper configuration to the HSE OTP of an uninitialized device.

```
SE Manager Tamper Example - Core running at 38000 kHz.
. SE manager initialization... SL_STATUS_OK (cycles: 7 time: 0 us)
. Read EMU RSTCAUSE register... SL_STATUS_OK (cycles: 3728 time: 98 us)
+ The EMU RSTCAUSE register (MSB..LSB): 00000043
. Read SE OTP configuration... SL_STATUS_NOT_INITIALIZED (cycles: 7487 time: 197 us)
+ Cannot read SE OTP configuration.
+ Press ENTER to initialize SE OTP for tamper configuration or press SPACE to abort.
+ Warning: The OTP configuration cannot be changed once written!
+ Press ENTER to confirm or press SPACE to abort if you are not sure.
+ Initialize SE OTP for tamper configuration... SL_STATUS_OK (cycles: 267256 time: 7033 us)
+ Issue a power-on or pin reset to activate the new tamper configuration.
. SE manager deinitialization... SL_STATUS_OK (cycles: 9 time: 0 us)
```

**Note:** This example does not enable the secure boot.

2. Press the **RESET** button on the WSTK to restart the program. It will display the current tamper configuration of the device.

```
SE Manager Tamper Example - Core running at 38000 kHz.
. SE manager initialization... SL_STATUS_OK (cycles: 10 time: 0 us)
. Read EMU RSTCAUSE register... SL_STATUS_OK (cycles: 3736 time: 98 us)
+ The EMU RSTCAUSE register (MSB..LSB): 00000043
. Read SE OTP configuration... SL_STATUS_OK (cycles: 7174 time: 188 us)
+ Secure boot: Disabled
+ Tamper source level
  Filter counter      : 1
  SE watchdog        : 4
  SE RAM CRC         : 4
  SE hard fault      : 4
  SE software assertion : 4
  SE secure boot     : 4
  User secure boot   : 0
  Mailbox authorization : 1
  DCI authorization  : 0
  OTP read          : 4
  Self test         : 4
  TRNG monitor      : 1
  PRS0              : 1
  PRS1              : 1
  PRS2              : 2
  PRS3              : 2
  PRS4              : 4
  PRS5              : 4
  PRS6              : 7
  PRS7              : 7
  Decouple BOD      : 4
  Temperature sensor : 2
  Voltage glitch falling : 2
  Voltage glitch rising : 2
  Secure lock       : 4
  SE debug          : 0
  Digital glitch    : 2
  SE ICACHE         : 4
+ Reset period for the tamper filter counter: ~32 ms x 1024
+ Activation threshold for the tamper filter: 4
+ Digital glitch detector always on: Disabled
+ Tamper reset threshold: 5

. Current tamper test is NORMAL.
+ Press SPACE to select NORMAL or TAMPER DISABLE, press ENTER to run.
```

## Simplicity Commander

1. Run the security writekey command to provision the Public Command Key (e.g., command\_pubkey.pem).

```
commander security writekey --command **command_pubkey.pem** --device EFR32MG21B010F1024 --serialno 440030580
```

```
Device has serial number 000000000000000014b457ffe0f77ce
```

```
=====
Please look through any warnings before proceeding.
THIS IS A ONE-TIME command which permanently ties debug and tamper access to certificates signed by this key.
Type 'continue' and hit enter to proceed or Ctrl-C to abort:
=====
continue
DONE
```

**Note:** The Public Command Key cannot be changed once written.

2. Run the security readkey command to read the Public Command Key from the HSE OTP for verification with the key in step 1.

```
commander security readkey --command --device EFR32MG21B010F1024 --serialno 440030580
```

```
B1BC6F6FA56640ED522B2EE0F5B3CF7E5D48F60BE8148F0DC08440F0A4E1DCA4  
7C04119ED6A1BE31B7707E5F9D001A659A051003E95E1B936F05C37EA793AD63  
DONE
```

3. Run the security genconfig command to generate a default user\_configuration.jsonfile for secure boot and tamper configuration.

```
commander security genconfig --nostore -o user_configuration.json --device EFR32MG21B010F1024 --serialno 440030580
```

```
Configuration file written to user_configuration.json  
DONE
```

**Note:** Simplicity Commander Version 1.14.6 or above is required to support tamper configuration for all HSE-SVH devices.

4. Use a text editor to modify the default tamper responses in `user_configuration.json` to the desired configuration as below.

```
{
  "OPN": "EFR32MG21B010F1024",
  "VERSION": "1.0.0",
  "mcu_flags": {
    "SECURE_BOOT_ANTIROLLBACK": false,
    "SECURE_BOOT_ENABLE": false,
    "SECURE_BOOT_PAGE_LOCK_FULL": false,
    "SECURE_BOOT_PAGE_LOCK_NARROW": false,
    "SECURE_BOOT_VERIFY_CERTIFICATE": false
  },
  "tamper_filter": {
    "FILTER_PERIOD": 10,
    "FILTER_THRESHOLD": 6,
    "RESET_THRESHOLD": 5
  },
  "tamper_flags": {
    "DGLITCH_ALWAYS_ON": false
  },
  "tamper_levels": {
    "DCLAUTH": 0,
    "DECOUPLE_BOD": 4,
    "DGLITCH": 2,
    "FILTER_COUNTER": 1,
    "MAILBOX_AUTH": 1,
    "OTP_READ": 4,
    "PRS0": 1,
    "PRS1": 1,
    "PRS2": 2,
    "PRS3": 2,
    "PRS4": 4,
    "PRS5": 4,
    "PRS6": 7,
    "PRS7": 7,
    "SECURE_LOCK": 4,
    "SELF_TEST": 4,
    "SE_CODE_AUTH": 4,
    "SE_DEBUG": 0,
    "SE_HARDFULT": 4,
    "SE_ICACHE": 4,
    "SE_RAM_CRC": 4,
    "SOFTWARE_ASSERTION": 4,
    "TEMP_SENSOR": 2,
    "TRNG_MONITOR": 1,
    "USER_CODE_AUTH": 0,
    "VGLITCH_FALLING": 2,
    "VGLITCH_RISING": 2,
    "WATCHDOG": 4
  }
}
```

**Note:** This example does not enable the secure boot.

- Run the `security writeconfig` command to program the secure boot and tamper configuration to the HSE OTP. This command can be executed once per device.

```
commander security writeconfig --configfile user_configuration.json --device EFR32MG21B010F1024 --serialno 440030580
```

```
=====
THIS IS A ONE-TIME configuration: Please inspect file before confirming:
user_configuration.json
Type 'continue' and hit enter to proceed or Ctrl-C to abort:
=====
continue
DONE
```

6. Run the `security readconfig` command to check the secure boot and tamper configuration of the device.

```
commander security readconfig --serialno 440030580
```

```
MCU Flags
Secure Boot           : Disabled
Secure Boot Verify Certificate : Disabled
Secure Boot Anti Rollback   : Disabled
Secure Boot Page Lock Narrow : Disabled
Secure Boot Page Lock Full  : Disabled

Tamper Levels
FILTER_COUNTER   : 1
WATCHDOG         : 4
SE_RAM_CRC       : 4
SE_HARDFFAULT    : 4
SOFTWARE_ASSERTION : 4
SE_CODE_AUTH     : 4
USER_CODE_AUTH   : 0
MAILBOX_AUTH     : 1
DCLAUTH          : 0
OTP_READ         : 4
SELF_TEST        : 4
TRNG_MONITOR     : 1
PRS0             : 1
PRS1             : 1
PRS2             : 2
PRS3             : 2
PRS4             : 4
PRS5             : 4
PRS6             : 7
PRS7             : 7
DECOUPLE_BOD     : 4
TEMP_SENSOR      : 2
VGLITCH_FALLING  : 2
VGLITCH_RISING   : 2
SECURE_LOCK      : 4
SE_DEBUG         : 0
DGLITCH          : 2
SE_ICACHE        : 4

Tamper Filter
Filter Period : 10
Filter Threshold : 6
Reset Threshold : 5

Tamper Flags
Digital Glitch Detector Always On: Disabled
DONE
```

## Simplicity Studio

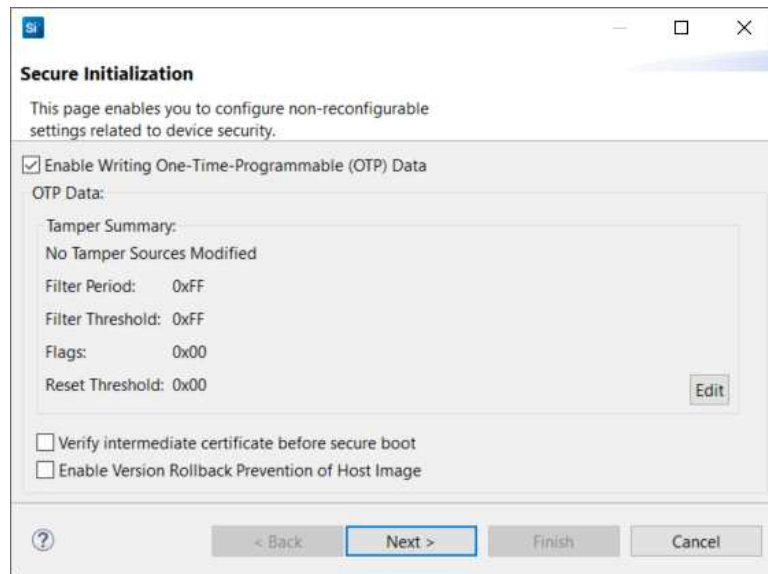
This example focuses on provisioning the Public Command Key and tamper configuration. It skips the procedures for provisioning of the Public Sign Key and Secure Boot Enabling.

1. Run the `util keytotoken` command to convert the Public Command Key file (PEM format) into a text file (`command_pubkey.txt`).

```
commander util keytotoken command_pubkey.pem --outfile command_pubkey.txt
```

```
Writing EC tokens to command_pubkey.txt...  
DONE
```

2. Open the **Security Settings** of the selected device as described in [Using Simplicity Studio](#).
3. Click **[Start Provisioning Wizard...]** in the upper right corner to display the **Secure Initialization** dialog box.



4. Click **[Edit]** to open the **Tamper Source Configuration** dialog box. Use the dropdown menus to modify the default tamper responses to the desired configuration. Click **[OK]** to exit.



Tamper Source Configuration

Select Tamper Response for each Tamper Source

Generate Interrupt	Filter Counter
System Reset	Watchdog
System Reset	SE RAM CRC
System Reset	SE Hardfault
System Reset	Software Assertion
System Reset	SE CodeAuth
Ignore	UserCodeAuth
Generate Interrupt	MailboxAuth
Ignore	DCIAuth
System Reset	OTP Read
Ignore	AutoCodeAuth
System Reset	self-test
Generate Interrupt	TRNG monitor
Generate Interrupt	PRS 0
Generate Interrupt	PRS 1
Increment filter counter	PRS 2
Increment filter counter	PRS 3
System Reset	PRS 4
System Reset	PRS 5
Erase OTP	PRS 6
Erase OTP	PRS 7
System Reset	DECOUPLE BOD
Increment filter counter	TempSensor
Increment filter counter	VGlitch Falling
Increment filter counter	VGlitch Rising
System Reset	SecureLock
Ignore	SE Debug
Increment filter counter	DGlitch
System Reset	SE ICACHE

0x0A

Tamper filter period configuration

0x06

Tamper filter threshold configuration

0x00

Tamper flags

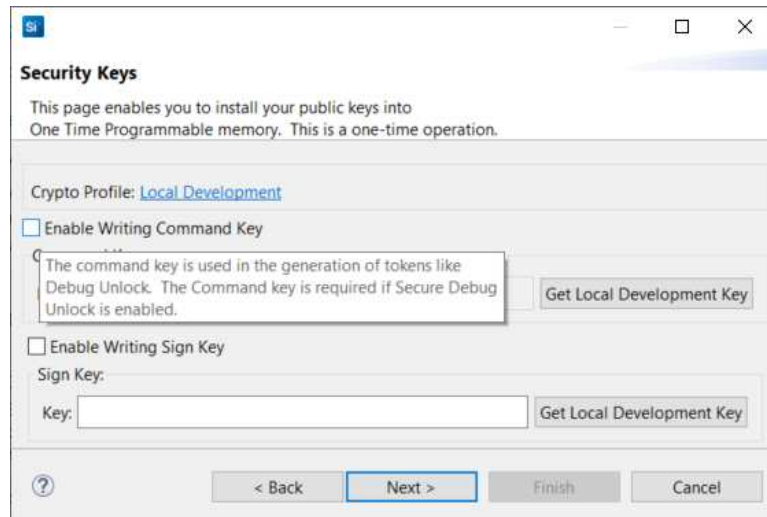
0x05

Tamper reset threshold

OK

Cancel

5. Click [Next >]. The Security Keys dialog box is displayed.



**Security Keys**

This page enables you to install your public keys into One Time Programmable memory. This is a one-time operation.

Crypto Profile: [Local Development](#)

☐ Enable Writing Command Key

The command key is used in the generation of tokens like Debug Unlock. The Command key is required if Secure Debug Unlock is enabled.

☐ Enable Writing Sign Key

Sign Key:

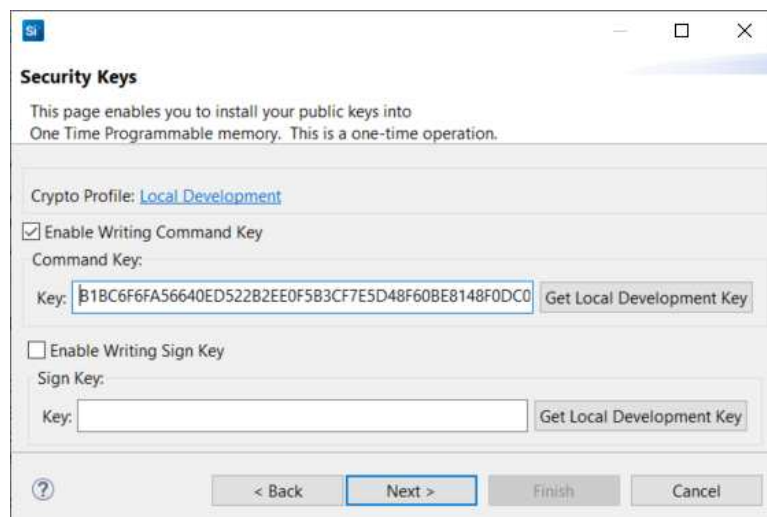
Key:  [Get Local Development Key](#)

[? < Back](#) [Next >](#) [Finish](#) [Cancel](#)

6. Using a text editor, open the `command_pubkey.txt` file generated in step 1.

```
MFG_SIGNED_BOOTLOADER_KEY_X : B1BC6F6FA56640ED522B2EE0F5B3CF7E5D48F60BE8148F0DC08440F0A4E1DCA4
MFG_SIGNED_BOOTLOADER_KEY_Y : 7C04119ED6A1BE31B7707E5F9D001A659A051003E95E1B936F05C37EA793AD63
```

7. Check **Enable Writing Command Key**. Copy the Public Command Key (X-point `B1BC...` first, then Y-point `7C04...`) to the **Key:** box under **Command Key**:



**Security Keys**

This page enables you to install your public keys into One Time Programmable memory. This is a one-time operation.

Crypto Profile: [Local Development](#)

☒ Enable Writing Command Key

Command Key:

Key:  [Get Local Development Key](#)

☐ Enable Writing Sign Key

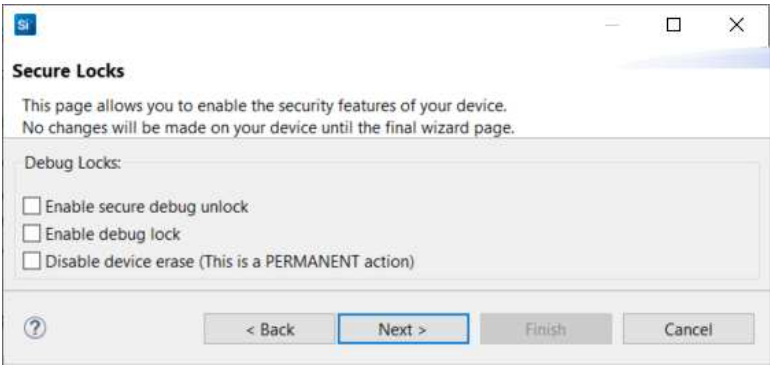
Sign Key:

Key:  [Get Local Development Key](#)

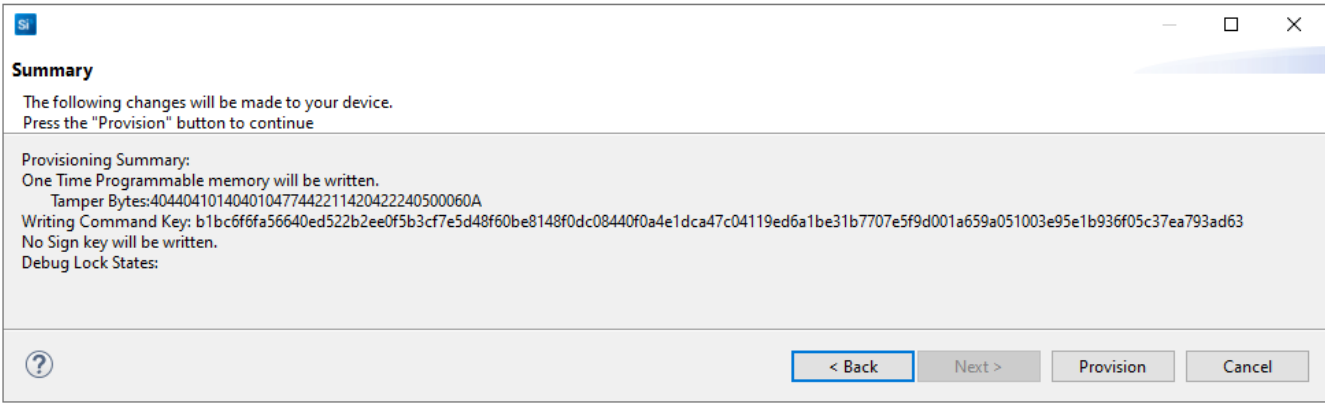
[? < Back](#) [Next >](#) [Finish](#) [Cancel](#)

**Note:** This example does not enable the secure boot (not checking **Enable Writing Sign Key** option).

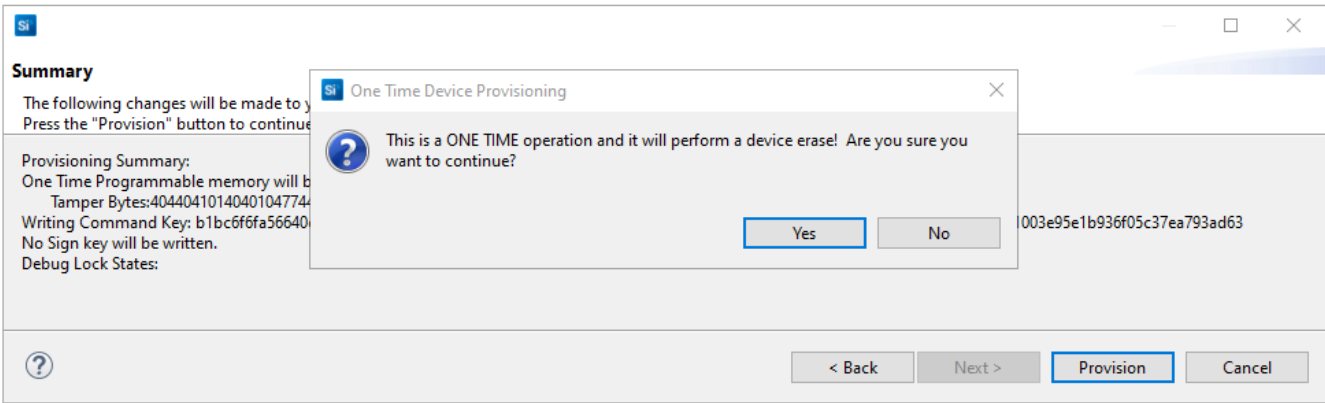
8. Click **[Next >]**. The **Secure Locks** dialog box is displayed. The **Debug locks** are set by default. Uncheck **Enable secure debug unlock** and **Enable debug lock**.



9. Click [Next >] to display the **Summary** dialog box. Verify the tamper configuration and Public Command Key in the Provisioning Summary are correct.

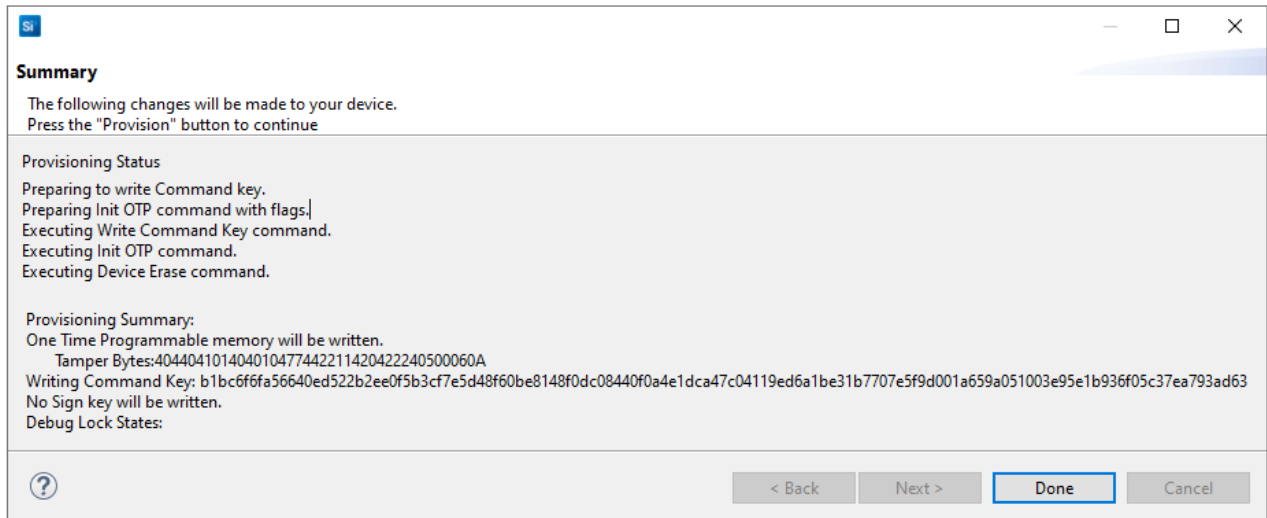


10. If the information displayed is correct, click [Provision]. Click [Yes] to confirm.

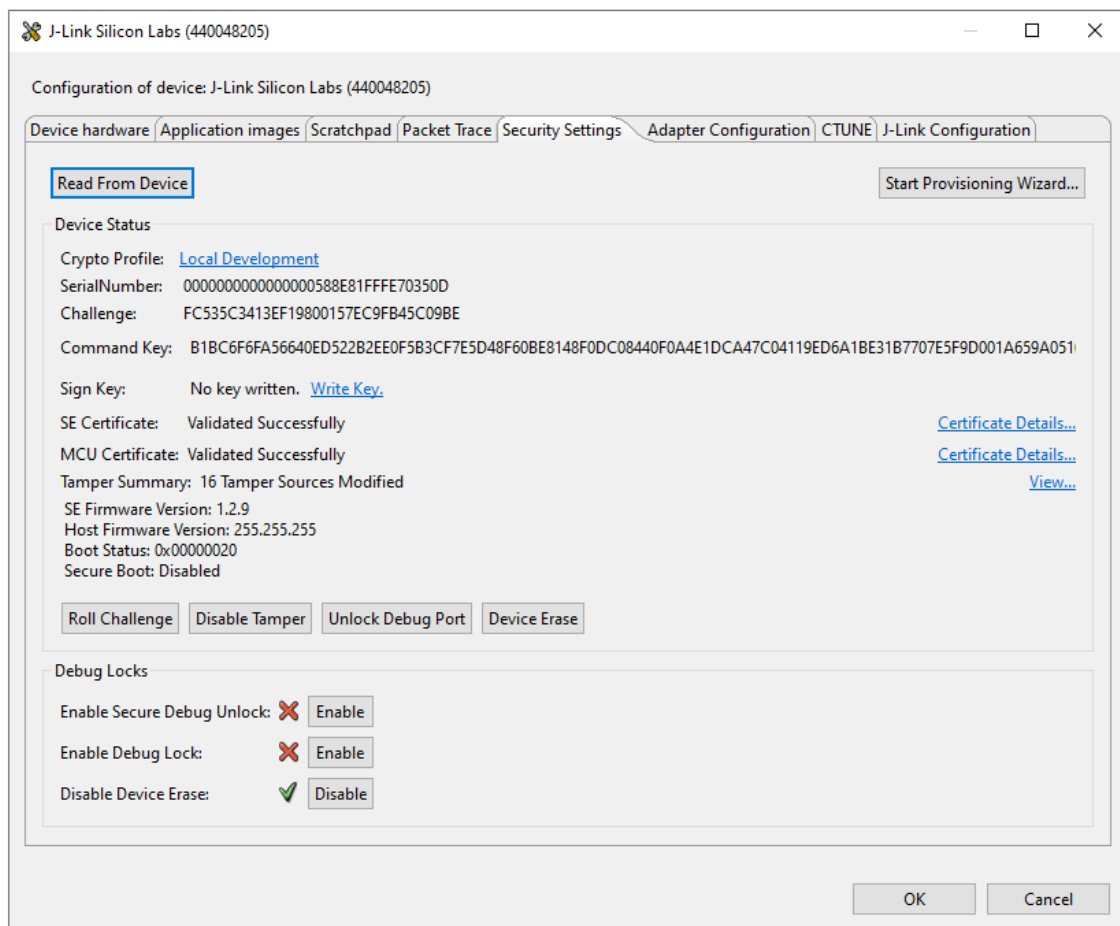


**Note:** The Public Command Key and tamper configuration cannot be changed once written.

11. The **Provisioning Status** is displayed in the **Summary** dialog box.



12. Click [Done] to exit the provisioning process. The device configuration is updated.



13. Click the View... link to check the tamper configuration or click [OK] to exit.

## Tamper Disable and Roll Challenge

### PRS Tamper Sources

The tamper configuration in the [SE Manager Tamper platform example](#) is used to demonstrate the tamper disable on HSE-SVH devices. The following tables list the PRS tamper source usage on EFR32xG21B and other HSE-SVH devices on this example. The push buttons PB0 and PB1 are on the Wireless Starter Kit (WSTK) Mainboard.

**Table:** PRS Tamper Source Usage on EFR32xG21B Devices

Source (Bit)	Default Level (Response)	User Level (Response)	PRS Producer	Tamper Disable Mask (1)
PRS0 (16)	0 (Ignore)	1 (Interrupt)	Push button PB0	0
PRS1 (17)	0 (Ignore)	1 (Interrupt)	-	1
PRS2 (18)	0 (Ignore)	2 (Filter)	Push button PB0	0
PRS3 (19)	0 (Ignore)	2 (Filter)	-	1
PRS4 (20)	0 (Ignore)	4 (Reset)	Push button PB1	1
PRS5 (21)	0 (Ignore)	4 (Reset)	Software (2)	1
PRS6 (22)	0 (Ignore)	7 (Erase OTP)	-	1
PRS7 (23)	0 (Ignore)	7 (Erase OTP)	-	1

**Notes:**

1. The [tamper disable mask](#) is `0x00fa0000` to restore the tamper sources PRS1, PRS3, PRS4, PRS5, PRS6, and PRS7 to default response (Ignore).
2. The Software PRS triggers the tamper source PRS5 to reset the device if the filter counter reaches the [trigger threshold](#) (4) within the [filter reset period](#) (~32 ms x 1024).

**Table:** PRS Tamper Source Usage on Other HSE-SVH Devices

Source (Bit) (1)	Default Level (Response)	User Level (Response)	PRS Producer	Tamper Disable Mask (2)
PRS0 (25 or 26)	0 (Ignore)	1 (Interrupt)	-	1
PRS1 (26 or 27)	0 (Ignore)	1 (Interrupt)	Push button PB0	0
PRS2 (27 or 28)	0 (Ignore)	2 (Filter)	Push button PB0	0
PRS3 (28 or 29)	0 (Ignore)	2 (Filter)	-	1
PRS4 (29 or 30)	0 (Ignore)	4 (Reset)	Push button PB1	1
PRS5 (30 or 31)	0 (Ignore)	4 (Reset)	Software (3)	1
PRS6 (31 or -)	0 (Ignore)	7 (Erase OTP)	-	1

**Notes:**

1. The HSE-SVH devices with ETAMPDET peripheral only have PRS0 (bit 26) to PRS5 (bit 31).
2. The [tamper disable mask](#) depends on whether the HSE-SVH device has an ETAMPDET peripheral.
  1. Without ETAMPDET peripheral, the tamper disable mask is `0xf2000000` to restore the tamper sources PRS0, PRS3, PRS4, PRS5, and PRS6 to default response (Ignore).
  2. With ETAMPDET peripheral, the tamper disable mask is `0xe4000000` to restore the tamper sources PRS0, PRS3, PRS4, and PRS5 to default response (Ignore).
3. The Software PRS triggers the tamper source PRS5 to reset the device if the filter counter reaches the [trigger threshold](#) (4) within the [filter reset period](#) (~32 ms x 1024).

## SE Manager - Tamper Platform Example

Click the [View Project Documentation](#) link to open the readme file for instructions on creating the project and running the example.

### Platform - SE Manager Tamper

This example project demonstrates the tamper feature of Secure Vault High device.

[View Project Documentation](#)

CREATE

Follow the procedures in [SE Manager - Tamper Platform Example](#) if the HSE OTP is uninitialized. The following sections describe an initialized device that runs in Normal and Tamper Disable modes.

## Normal

1. Press `ENTER` to run the `NORMAL` tamper demo. Follow the instructions to go through the example.

```
. Current tamper test is NORMAL.
+ Press SPACE to select NORMAL or TAMPER DISABLE, press ENTER to run.

. Normal tamper test instructions:
+ Press PB0 to increase filter counter and tamper status is displayed.
+ PRS will issue a tamper reset if filter counter reaches 4 within ~32 ms x 1024.
+ Press PB1 to issue a tamper reset.
+ Device will enter diagnostic mode if tamper reset reaches 5.
```

2. Press PB0 to trigger [PRS0 \(Interrupt\)](#) and [PRS2 \(Filter\)](#) to issue an interrupt. The active tamper sources ( `0x00050000` ) of the EFR32xG21B device are PRS0 ( bit 16 ) and PRS2 ( bit 18 ).

```
. Get tamper status... SL_STATUS_OK (cycles: 11937 time: 314 us)
+ Recorded tamper status (MSB..LSB): 00050001
+ Currently active tamper sources (MSB..LSB): 00050000
```

3. Press PB0 (Filter on PRS2) 4 times within ~32 ms x 1024 to trigger an interrupt when reaching the filter counter threshold. The program will use software PRS to issue a tamper reset through the [PRS5](#) tamper source. The active tamper sources ( `0x00050002` ) of the EFR32xG21B device are Filter ( bit 2 ), PRS0 ( bit 16 ), and PRS2 ( bit 18 ).

```
. Get tamper status... SL_STATUS_OK (cycles: 11725 time: 308 us)
+ Recorded tamper status (MSB..LSB): 00050002
+ Currently active tamper sources (MSB..LSB): 00050002
+ Tamper filter threshold is reached, issue a reset through PRS
```

4. Press PB1 to trigger [PRS4 \(Reset\)](#) to issue a tamper reset.
5. After a tamper reset, the `SETAMPER` (bit 13) in `EMU->RSTCAUSE` register is set. Note that bit 1 indicates a pin reset and will also be set.

```
. Read EMU RSTCAUSE register... SL_STATUS_OK (cycles: 4071 time: 107 us)
+ The EMU RSTCAUSE register (MSB..LSB): 00002002
+ The tamper reset is observed
```

6. After five consecutive tamper resets ([reset threshold](#) in this example), the device will enter diagnostic mode until a power-on or pin reset.

## Tamper Disable

This example uses the [tamper disable mask](#) ( `0x00fa0000` ) to restore the tamper sources [PRS1](#), [PRS3](#), [PRS4](#), [PRS5](#), [PRS6](#), and [PRS7](#) of EFR32xG21B device to default response (Ignore).

1. Press `SPACE` to select `TAMPER DISABLE`, press `ENTER` to run.

```
. Current tamper test is NORMAL.
+ Press SPACE to select NORMAL or TAMPER DISABLE, press ENTER to run.
+ Current tamper test is TAMPER DISABLE.
```

2. This example will prompt to program the default Public Command Key in flash to the HSE OTP if this key does not exist. Press **ENTER** two times to confirm and **ENTER** again to restore the default tamper level. Follow the instructions shown in step 3 to go through the example (steps 4 to 6).

```
. Verify the device public command key in SE OTP.
+ Exporting a public command key from a hard-coded private command key... SL_STATUS_OK (cycles: 210999 time: 5552 us)
+ Reading the public command key from SE OTP... SL_STATUS_NOT_INITIALIZED (cycles: 7763 time: 204 us)
+ Press ENTER to program public command key in SE OTP or press SPACE to abort.
+ Warning: The public command key in SE OTP cannot be changed once written!
+ Press ENTER to confirm or press SPACE to skip if you are not sure.
+ Programming a public command key to SE OTP... SL_STATUS_OK (cycles: 79656 time: 2096 us)
+ Press ENTER to disable tamper signals or press SPACE to exit.
```

3. Press **ENTER** to restore the default tamper level if the default Public Command Key in flash matches with the key in the HSE OTP. Follow the instructions to go through the example (steps 4 to 6).

```
. Verify the device public command key in SE OTP.
+ Exporting a public command key from a hard-coded private command key... SL_STATUS_OK (cycles: 200804 time: 5284 us)
+ Reading the public command key from SE OTP... SL_STATUS_OK (cycles: 7134 time: 187 us)
+ Comparing exported public command key with SE OTP public command key... OK
+ Press ENTER to disable tamper signals or press SPACE to exit.

. Start the tamper disable processes.
+ Creating a private certificate key in a buffer... SL_STATUS_OK (cycles: 214059 time: 5633 us)
+ Exporting a public certificate key from a private certificate key... SL_STATUS_OK (cycles: 206545 time: 5435 us)
+ Read the serial number of the SE and save it to access certificate... SL_STATUS_OK (cycles: 7930 time: 208 us)
+ Signing the access certificate with private command key... SL_STATUS_OK (cycles: 222650 time: 5859 us)
+ Request challenge from the SE and save it to challenge response... SL_STATUS_OK (cycles: 4208 time: 110 us)
+ Signing the challenge response with private certificate key... SL_STATUS_OK (cycles: 223559 time: 5883 us)
+ Creating a tamper disable token to disable tamper signals... SL_STATUS_OK (cycles: 946431 time: 24906 us)
+ Success to disable the tamper signals!

. Tamper disable test instructions:
+ Press PB0 to increase filter counter and tamper status is displayed.
+ PRS will NOT issue a tamper reset even filter counter reaches 4 within ~32 ms x 1024.
+ Press PB1 will NOT issue a tamper reset.
+ Issue a power-on or pin reset to re-enable the tamper signals.
+ Press ENTER to roll the challenge to invalidate the current tamper disable token or press SPACE to exit.
```

4. Press PB0 to verify tamper sources **PRS0 (Interrupt)** and **PRS2 (Filter)** of EFR32xG21B device can still issue an interrupt.

```
. Get tamper status... SL_STATUS_OK (cycles: 11259 time: 296 us)
+ Recorded tamper status (MSB..LSB): 00050001
+ Currently active tamper sources (MSB..LSB): 00050000
```

5. The **PRS5** tamper source (configured as Reset) was restored to the default (Ignore), so it cannot issue a tamper reset even if users press PB0 (Filter on PRS2) 4 times within ~32 ms x 1024.
6. The **PRS4** tamper source (configured as Reset) was restored to the default (Ignore), so it cannot issue a tamper reset even if users press PB1.
7. Issue a power-on or pin reset to exit the tamper disable state or press **ENTER** to roll the challenge.



```

. Check and roll the challenge.
+ Request current challenge from the SE... SL_STATUS_OK (cycles: 0 time: 0 us)
+ The current challenge (16 bytes):
  AA C1 79 FC FC C5 78 8E A0 3F 91 AB 5D A9 C5 04
+ Rolling the challenge... SL_STATUS_OK (cycles: 0 time: 0 us)
+ Request rolled challenge from the SE... SL_STATUS_OK (cycles: 0 time: 0 us)
+ The rolled challenge (16 bytes):
  0F 63 9C 44 46 E4 7C B2 C9 CA 66 13 34 34 92 8E
+ Issue a power-on or pin reset to activate the rolled challenge.

. SE manager deinitialization... SL_STATUS_OK (cycles: 0 time: 0 us)

```

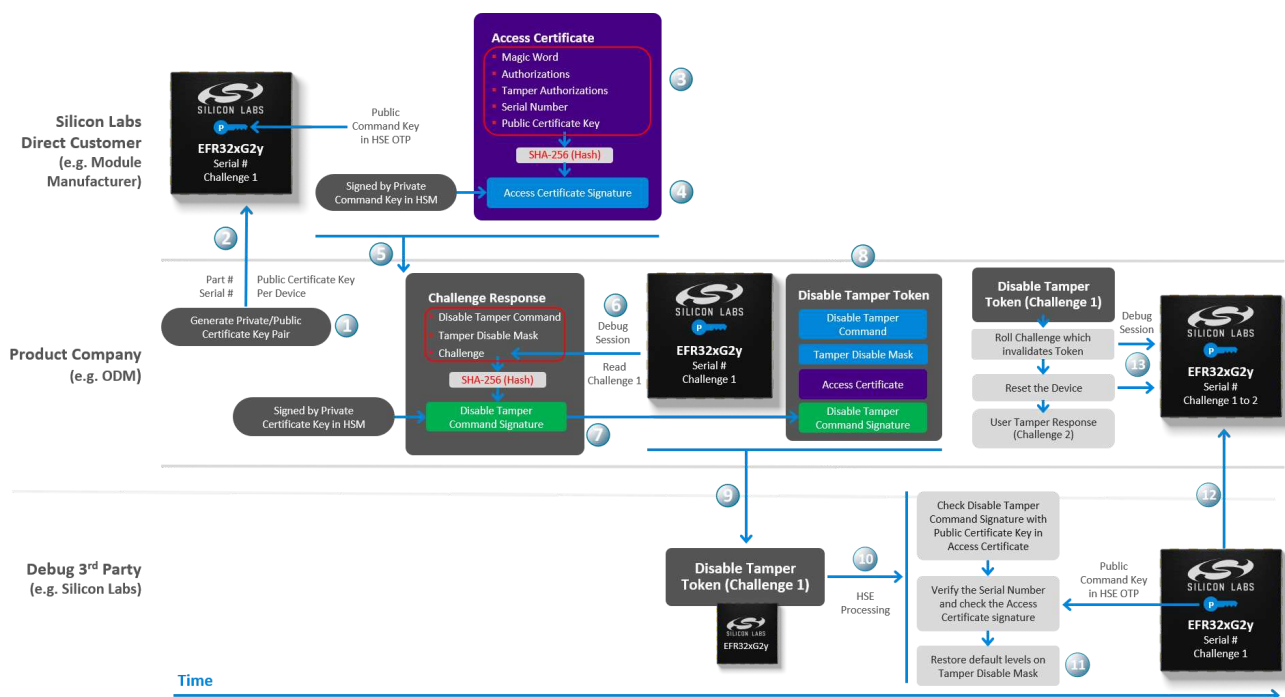
## Simplicity Commander

The tamper disable was designed with three organizations in mind:

1. The Direct Customer to whom Silicon Labs sells the chip. This chip has the Public Command Key installed in the SE OTP.
2. The Product Company is a customer of the Direct Customer. This is applicable if the Direct Customer is creating a white-labeled product for another company or a sub-component that goes into another company's product.
3. The Debug 3<sup>rd</sup> Party could be anyone, internal or external, that the Product Company decides is qualified to debug the device.

Because the Public Command Key is installed into the SE OTP of a large number of devices and cannot be changed, the corresponding Private Command Key must be guarded by a very stringent process. If this Private Command Key is ever leaked, all the devices programmed with the corresponding Public Command Key will be compromised.

A tamper disable use case is described in the following figure, and the signing process is performed by a Hardware Security Module (HSM).



The tamper disable flow moving across the time axis from left to right is explained below:

1. The Product Company creates a **Private/Public Certificate Key pair** for each device. Because the key pair is assigned only to a single device, the company may not need to protect the Private Certificate Key as securely as the Private Command Key by the Direct Customer.  
In this example, the Private/Public Certificate Key pair ( `cert_key.pem` and `cert_pubkey.pem` ) is generated by running the `util genkey` command.



```
commander util genkey --type ecc-p256 --privkey cert_key.pem --pubkey cert_pubkey.pem
```

```
Generating ECC P256 key pair...
Writing private key file in PEM format to cert_key.pem
Writing public key file in PEM format to cert_pubkey.pem
DONE
```

2. The Public Certificate Key ( `cert_pubkey.pem` ) for each device is passed to the Silicon Labs Direct Customer. The part number and serial number are also required if Direct Customer cannot access the device. Run the `security status` command to get the device serial number. The `--serialno` option is for the [J-Link serial number](#) of the WSTK.

```
commander security status --device EFR32MG21B010F1024 --serialno 440030580
```

```
SE Firmware version : 1.2.9
Serial number      : 000000000000000014b457ffe0f77ce
Debug lock        : Disabled
Device erase      : Enabled
Secure debug unlock : Disabled
Tamper status     : OK
Secure boot       : Disabled
Boot status       : 0x20 - OK
Command key installed : True
Sign key installed  : False
DONE
```

3. The Direct Customer then places that Public Certificate Key in the [access certificate](#). The access certificate is per device because it contains the unique device serial number. This certificate is generated once upon creation of the device, and thereafter, is generally only modified when the Private/Public Certificate Key pair is changed by the Product Company. The following two steps are **OPTIONAL** for customization of Authorizations and Tamper Authorizations.
- a. (**Optional**) Run the `security genauth` command to generate the default certificate authorization file ( `certificate_authorization.json` ).

```
commander security genauth -o certificate_authorizations.json --nostore --serialno 440030580
```

```
DONE
```

- b. (**Optional**) Use a text editor to modify the [default Authorizations and Tamper Authorizations](#) in the `json` file. Run the `security gencert` command with the following parameters from the Product Company to generate an unsigned access certificate ( `access_certificate.extsign` ) in Security Store:

- Device part number
- Device serial number
- Public Certificate Key

```
commander security gencert --device EFR32MG21B010F1024 --deviceserialno 000000000000000014b457ffe0f77ce
--cert-pubkey cert_pubkey.pem --extsign
```

```
Authorization file written to Security Store:
C:/Users/<username>/AppData/Local/SiliconLabs/commander/SecurityStore/device_000000000000000014b457ffe0f77ce/certificate_authoriz

Cert key written to Security Store:
C:/Users/<username>/AppData/Local/SiliconLabs/commander/SecurityStore/device_000000000000000014b457ffe0f77ce/cert_pubkey.pem

Created an unsigned certificate in Security Store:
C:/Users/<username>/AppData/Local/SiliconLabs/commander/SecurityStore/device_000000000000000014b457ffe0f77ce/access_certificate.

DONE
```

#### Notes:

- The `--extsign` option to create an unsigned access certificate is only available in Simplicity Commander Version 1.11.2 or above.

[Elements of the Access Certificate on page](#)).

- **(Optional)** Use `--authorization` option if the customized `json` file generated in the above optional steps (a) and (b) is used.

```
commander security gencert --device EFR32MG21B010F1024 --authorization certificate_authorizations.json
--deviceserialno 000000000000000014b457ffe0f77ce --cert-pubkey cert_pubkey.pem --extsign
```

- The signing of the access certificate can be done by passing an unsigned access certificate to a Hardware Security Module (HSM) containing the Private Command Key.

In this example, the OpenSSL is used to sign the access certificate ( `access_certificate.extsign` ) in Security Store with the Private Command Key ( `command_key.pem` ). The [access certificate signature](#) is in the `cert_signature.bin` file.

```
openssl dgst -sha256 -binary -sign command_key.pem -out cert_signature.bin access_certificate.extsign
```

Run the `util signcert` command with the following parameters to verify the signature and generate the signed access certificate ( `access_certificate.bin` ):

- Unsigned access certificate
- Access certificate signature
- Public Command Key

```
commander util signcert access_certificate.extsign --cert-type access --signature cert_signature.bin
--verify command_pubkey.pem --outfile access_certificate.bin
```

```
R = 76CDC5BA18E5248FDA5418002F250F149B449829A005D6F0726268016CC53ED4
S = E4B8ABA2CF742B0E6CC5BA2C1023D76BEEF3C4A11DA97CC4D23459F32237A206
Successfully verified signature
Successfully signed certificate
DONE
```

#### Notes:

- Put the required files in the same folder to run the command.
  - The `util signcert` command for access certificate is only available in Simplicity Commander Version 1.11.2 or above.
  - The access certificate signature can be in a Raw or Distinguished Encoding Rules (DER) format.
- The access certificate is passed to the Product Company. The purpose of the access certificate is to grant overall debug access capabilities to the Product Company and authorize them to allow third parties to debug the device. The Product Company can now use the access certificate to generate the [Disable Tamper Token](#). The same access certificate can be used to generate as many Disable Tamper Tokens as necessary without having to ever go back to the Direct Customer.
  - To create the Disable Tamper Token, a debug session must be started with the device and the challenge value (which is a random number `Challenge 1` in this example) should be read out to generate the [challenge response](#). Run the `security gencommand` to generate the challenge response without [disable tamper command signature](#) and store it in a file ( `command_unsign.bin` ).

```
commander security gencommand --action disable-tamper --disable-param 0x00fa0000 -o command_unsign.bin
--nostore --device EFR32MG21B010F1024 --serialno 440030580
```

```
Unsigned command file written to:
command_unsign.bin
DONE
```

- The [tamper disable mask](#) ( `0x00fa0000` ) is based on the Tamper platform example on EFR32xG21B devices ([Table PRS Tamper Source Usage on EFR32xG21B Devices](#)).
  - If the `--disable-param` option is not provided, it will restore all tamper sources ( `0xffffffffb6` ) by default.
- The challenge response is then cryptographically hashed (SHA-256) to create a digest. The digest is then signed by the Private Certificate Key to generate the disable tamper command signature. The signing of the challenge response can be done by passing an unsigned challenge response to a Hardware Security Module (HSM) containing the Private Certificate Key. In this example, the OpenSSL is used to sign the challenge response ( `command_unsign.bin` ) with the Private Certificate Key ( `cert_key.pem` ). The disable tamper command signature is in the `command_signature.bin` file.

```
commander security disabletamper --disable-param 0x00fa0000 --cert access_certificate.bin  
--command-signature command_signature.bin EFR32MG21B010F1024 --serialno 440030580
```

191/280

C:/Users/<username>/AppData/Local/SiliconLabs/commander/SecurityStore/device\_0000000000000000588e81fffe70350d  
DONE

11. The Disable Tamper Token and the device are now delivered to the Debug **3<sup>rd</sup>** Party.  
Run the `security gencommand` command to create the Security Store to place the Disable Tamper Token file.

```
commander security gencommand --action disable-tamper --disable-param 0x00fa0000 --device EFR32MG21B010F1024 --serialno 440030580
```

```
Unsigned command file written to Security Store:  
C:/Users/<username>/AppData/Local/SiliconLabs/commander/SecurityStore/device_000000000000000014b457ffe0f77ce/challenge_8e7f73e  
DONE
```

Copy the Disable Tamper Token file ( tamper\_payload\_111101000000000000000000.bin ) from Product Company to the Windows Security Store challenge\_<Challenge value> folder located in C:\Users\<PC username>\AppData\Local\SiliconLabs\commander\SecurityStore\device\_<Serial number>\challenge\_<Challenge value> .

- The device compares the Disable Tamper Token contents with its internal serial number, challenge value, and Public Command Key to determine the token's authenticity. If authentic, it will execute the `disable tamper command` to restore the default levels on the `tamper disable mask` ( 0xfa000000 ) ; otherwise, it will ignore the command.  
Run the `security disabletamper` command to disable the tamper.

```
commander security disabletamper --disable-param 0x00fa0000 --device EFR32MG21B010F1024 --serialno 440030580
```

```
Disabling tamper with tamper payload:  
C:/Users/<username>/AppData/Local/SiliconLabs/commander/SecurityStore/device_000000000000000014b457ffe0f77ce/challenge_8e7f73e  
successfully disabled.  
DONE
```

**Note:** Users can verify the Disable Tamper Token by following steps 4 to 6 in [Tamper Disable](#) if the EFR32xG21B device is running in the [Normal](#) mode of the SE Manager Tamper platform example.

13. The Debug **3<sup>rd</sup>** Party can now use this same Disable Tamper Token to disable the tamper (step 12), over and over again after each power-on or pin reset, until they have finished debugging the device.
14. Once the Debug **3<sup>rd</sup>** Party has finished debugging, they will send the device back to the Product Company.
15. Once the Product Company receives the device, they will immediately start a debug session to roll the challenge (from Challenge 1 to Challenge 2 in this example). Rolling the challenge will effectively invalidate any Disable Tamper Token that has been previously given to any third party.

Run the `security rollchallenge` command and reset the device to invalidate the current Disable Tamper Token. The challenge cannot be rolled before it has been used at least once — that is, by running the `security disabletamper` or `security unlock` command.

```
commander security rollchallenge --device EFR32MG21B010F1024 --serialno 440030580
```

Challenge was rolled successfully.  
DONE

The unlock token is invalidated after rolling the challenge because any previously issued Disable Tamper Token now contains a different challenge value ( Challenge 1 ) than the challenge value currently in the device ( Challenge 2 ).

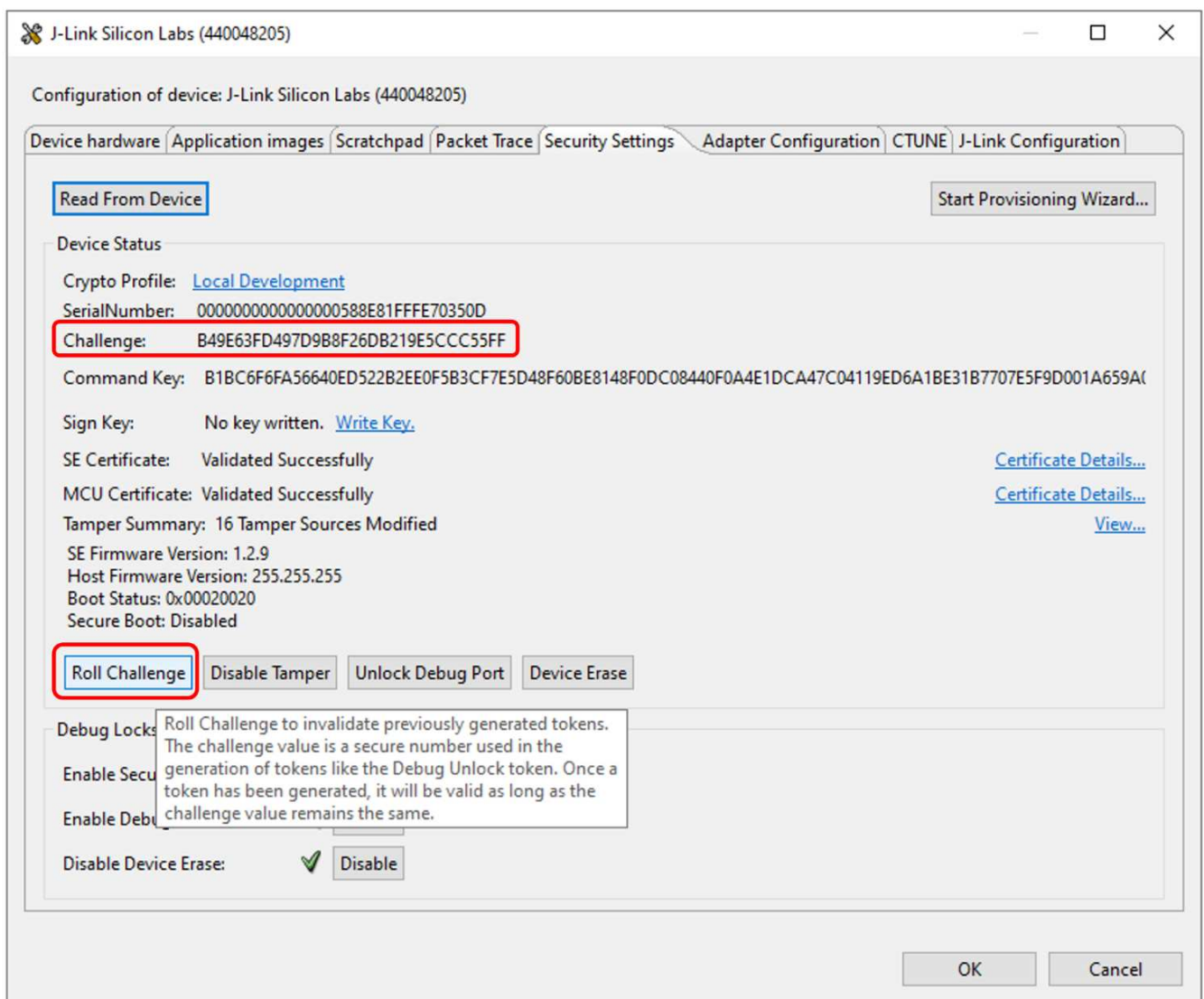
The validation process of any previously issued Disable Tamper Token will always fail until a new Disable Tamper Token is issued with a current matching challenge value ( Challenge 2 ).

**Note:** Direct Customer can directly use the Private Command Key on the connected chip to generate the Disable Tamper Token in Security Store. But it has a high risk (cannot use HSM) to leak the Private Command Key to a 3<sup>rd</sup> party when using this approach.

```
commander security disabletamper --disable-param 0x00fa0000 --command-key command_key.pem  
--device EFR32MG21B010F1024 --serialno 440030580
```

## Simplicity Studio

1. Open Security Settings of the selected device as described in [Using Simplicity Studio](#).
2. Click [Roll Challenge] to generate a new challenge value to invalidate the Disable Tamper Token for tamper disable. Click [OK] to exit.



## Authenticating Silicon Labs Devices using Device Certificates

# Authenticating Silicon Labs Devices Using Device Certificates

Note: This section replaces *AN1268: Authenticating Silicon Labs Devices Using Device Certificates*. Further updates to this application note will be provided here.

This application note describes how to authenticate a device as a genuine Silicon Labs product at any time during its life. The digital certificates for secure identity are stored in the device and the Silicon Labs Server. This secure identity feature is only available on Secure Vault High devices.

## Key Points

- Secure identity on Secure Vault High devices
- Device certificate options
- Entity Attestation Token (EAT)
- Remote authentication process
- Examples for certificate chain verification and remote authentication

Series 2 Device Security Features

# Series 2 Device Security Features

Protecting IoT devices against security threats is central to a quality product. Silicon Labs offers several security options to help developers build secure devices, secure application software, and secure paths of communication to manage those devices. Silicon Labs’ security offerings were significantly enhanced by the introduction of the Series 2 products that included a Secure Engine. The Secure Engine is a tamper-resistant component used to securely store sensitive data and keys and to execute cryptographic functions and secure services.

On Series 1 devices, the security features are implemented by the TRNG (if available) and CRYPTO peripherals.

On Series 2 devices, the security features are implemented by the Secure Engine and CRYPTOACC (if available). The Secure Engine may be hardware-based, or virtual (software-based). Throughout this document, the following abbreviations are used:

- HSE - Hardware Secure Engine
- VSE - Virtual Secure Engine
- SE - Secure Engine (either HSE or VSE)

Additional security features are provided by Secure Vault. Three levels of Secure Vault feature support are available, depending on the part and SE implementation, as reflected in the following table:

Level (1)	SE Support	Part (2)
Secure Vault High (SVH)	HSE only (HSE-SVH)	EFR32xG2yB (3)
Secure Vault Mid (SVM)	HSE (HSE-SVM)	EFR32xG2yA (3)
"	VSE (VSE-SVM)	EFR32xG2y, EFM32PG2y (4)
Secure Vault Base (SVB)	N/A	MCU Series 1 and Wireless SoC Series 1

Notes:

1. The features of different Secure Vault levels can be found in <https://www.silabs.com/security>.
2. The x is a letter (B, F, M, or Z).
3. At the time of this writing, the y is a digit (1, 3, or 4).
4. At the time of this writing, the y is a digit (2).

Secure Vault Mid consists of two core security functions:

- **Secure Boot:** Process where the initial boot phase is executed from an immutable memory (such as ROM) and where code is authenticated before being authorized for execution.
- **Secure Debug access control:** The ability to lock access to the debug ports for operational security, and to securely unlock them when access is required by an authorized entity.

Secure Vault High offers additional security options:

- **Secure Key Storage:** Protects cryptographic keys by "wrapping" or encrypting the keys using a root key known only to the HSE-SVH.
- **Anti-Tamper protection:** A configurable module to protect the device against tamper attacks.
- **Device authentication:** Functionality that uses a secure device identity certificate along with digital signatures to verify the source or target of device communications.

A Secure Engine Manager and other tools allow users to configure and control their devices both in-house during testing and manufacturing, and after the device is in the field.

## User Assistance



In support of these products Silicon Labs offers whitepapers, webinars, and documentation. The following table summarizes the key security documents:

Document	Summary	Applicability
<a href="#">Series 2 Secure Debug</a>	How to lock and unlock Series 2 debug access, including background information about the SE	Secure Vault Mid and High
<a href="#">Series 2 Secure Boot with RTSL</a>	Describes the secure boot process on Series 2 devices using SE	Secure Vault Mid and High
<a href="#">Anti-Tamper Protection Configuration and Use</a>	How to program, provision, and configure the anti-tamper module	Secure Vault High
Authenticating Silicon Labs Devices using Device Certificates (this document)	How to authenticate a device using secure device certificates and signatures, at any time during the life of the product	Secure Vault High
<a href="#">Secure Key Storage</a>	How to securely 'wrap' keys so they can be stored in non-volatile storage.	Secure Vault High
AN1222: Production Programming of Series 2 Devices	How to program, provision, and configure security information using SE during device production	Secure Vault Mid and High

## Key Reference

Public/Private keypairs along with other keys are used throughout Silicon Labs security implementations. Because terminology can sometimes be confusing, the following table lists the key names, their applicability, and the documentation where they are used.

Key Name	Customer Programmed	Purpose
Public Sign key (Sign Key Public)	Yes	Secure Boot binary authentication and/or OTA upgrade payload authentication
Public Command key (Command Key Public)	Yes	Secure Debug Unlock or Disable Tamper command authentication
OTA Decryption key (GBL Decryption key) aka AES-128 Key	Yes	Decrypting GBL payloads used for firmware upgrades
Attestation key aka Private Device Key	No	Device authentication for secure identity

## Device Compatibility

This application note applies to Series 2 HSE-SVH device families. Refer to [IoT Endpoint Security Fundamentals](#) for details on supporting devices.



## Introduction

# Introduction

One of the biggest challenges for connected devices is post-deployment authentication. Silicon Labs' factory trust provisioning service with optional secure programming provides a secure device identity certificate, analogous to a birth certificate, for each individual silicon die during integrated circuit (IC) manufacturing. This enables post-deployment security, authenticity, and attestation-based health checks. The device certificate guarantees the authenticity of the device for its lifetime. When the certificate is checked, a digital signature confirms that the certificate received has not been tampered with.

Certificates can now be used to authenticate Internet of Things (IoT) devices as well as Internet servers, now that Silicon Labs' HSE-SVH devices have both cryptographic acceleration in hardware and tamper-resistant storage to handle digital certificate operations.

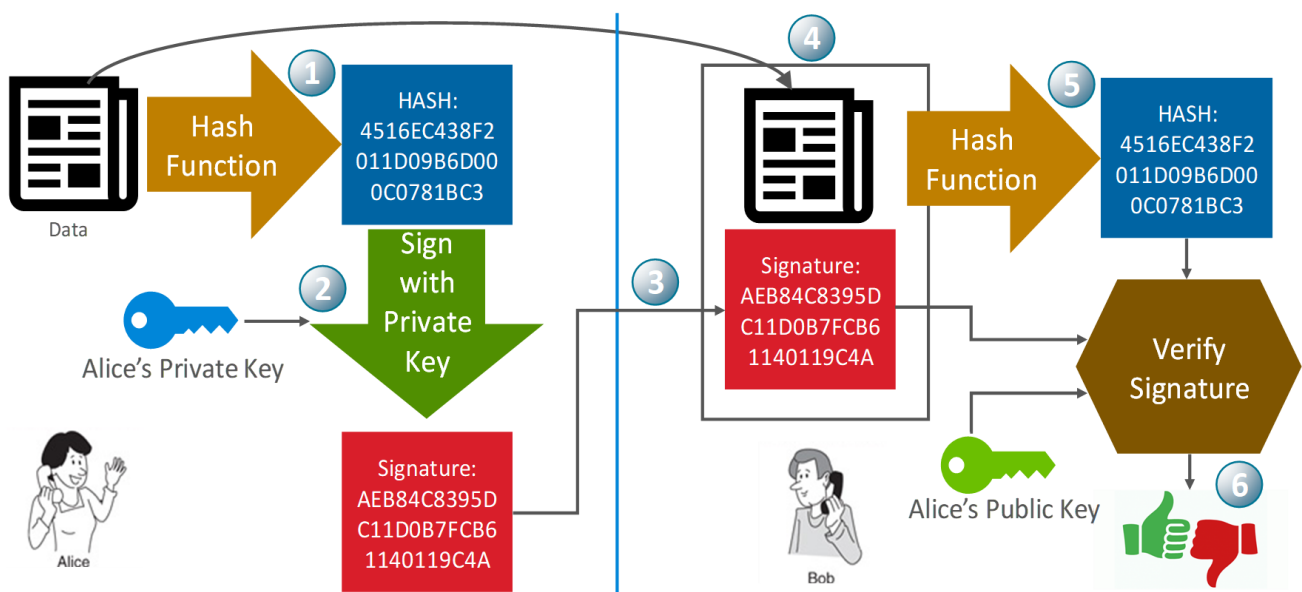
The digital signature and certificates are major cryptographic tools to verify the device is authentic. These tools are described in the following sections.

## Digital Signature

The digital signature is used to protect integrity and authenticity of an electronic message.

### Digital Signature Example

Alice wants to give data to Bob, and Bob wants to make sure that the data came from Alice and has not been tampered with. Alice has a private/public key pair, and has previously given Bob her public key.



1. Alice generates the hash (for example SHA256) of the data.
2. Alice's private key is used to sign the hash to create a signature. The hash is signed instead of the data itself because the signing operation is slow. Therefore it is more efficient to sign the hash instead of the arbitrarily large data.
3. The signature is attached to the end of the data.
4. The data and signature are given to Bob.

5. Bob independently generates the hash of the data.
6. The signature is verified with the hash and Alice's public key, which results in a true or false outcome indicating if the data is valid.

**Note:** This scheme requires distribution of Alice's public key.

## Digital Certificates and Chain of Trust

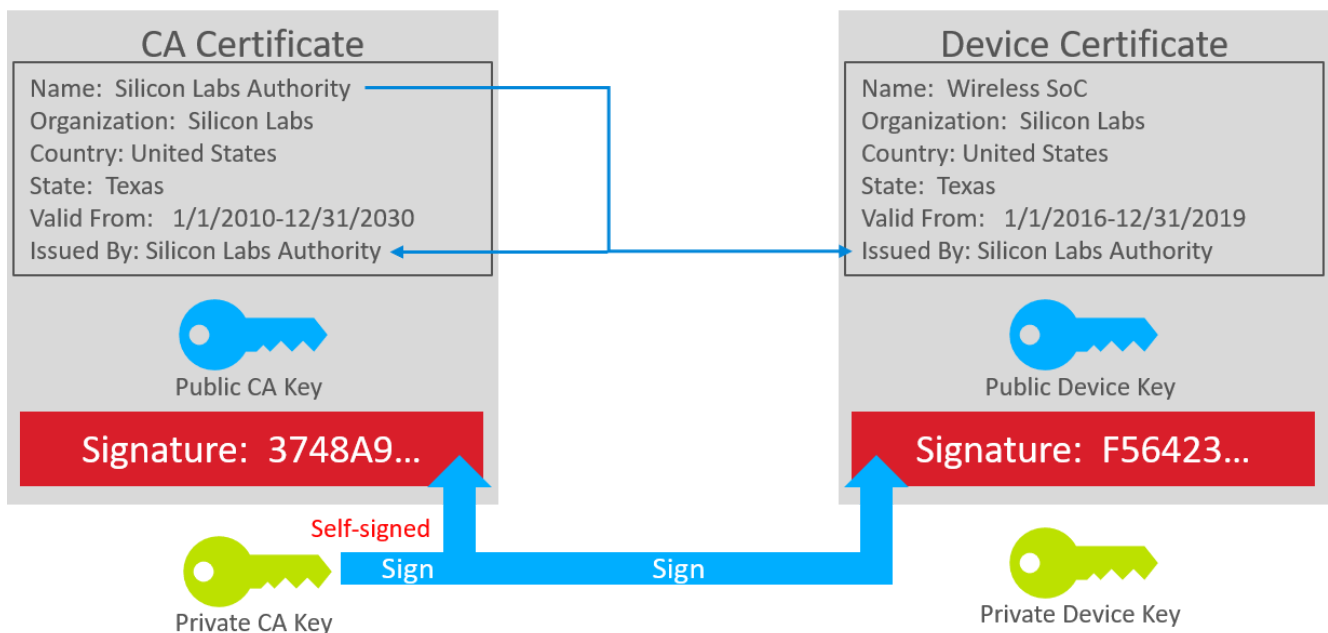
In [Digital Signature Example](#), Bob already had access to Alice's public key, which he trusted. However, it is not always feasible to pre-share a public key with everyone for secure identity verification, and no mechanism exists to revoke or inactivate the public key in case it gets stolen.

A digital certificate is simply a small, verifiable data file that contains identity credentials and a public key. That data is then signed either with the corresponding private key, or a different private key. The digital certificate can be used to prove the ownership of a public key.

- If it is signed using the corresponding private key, it is called a self-signed certificate.
- If it is signed by another private key, the owner of that private key is acting as a Certificate Authority (CA).
- A Certificate Authority (CA) is a trusted third party by both the owner and party relying on the certificate.

Concatenation of digital certificates builds a chain of trust.

- At the root of the chain is a self-signed certificate called a root certificate or a CA certificate.
- The root or CA certificate can be used to sign another certificate.



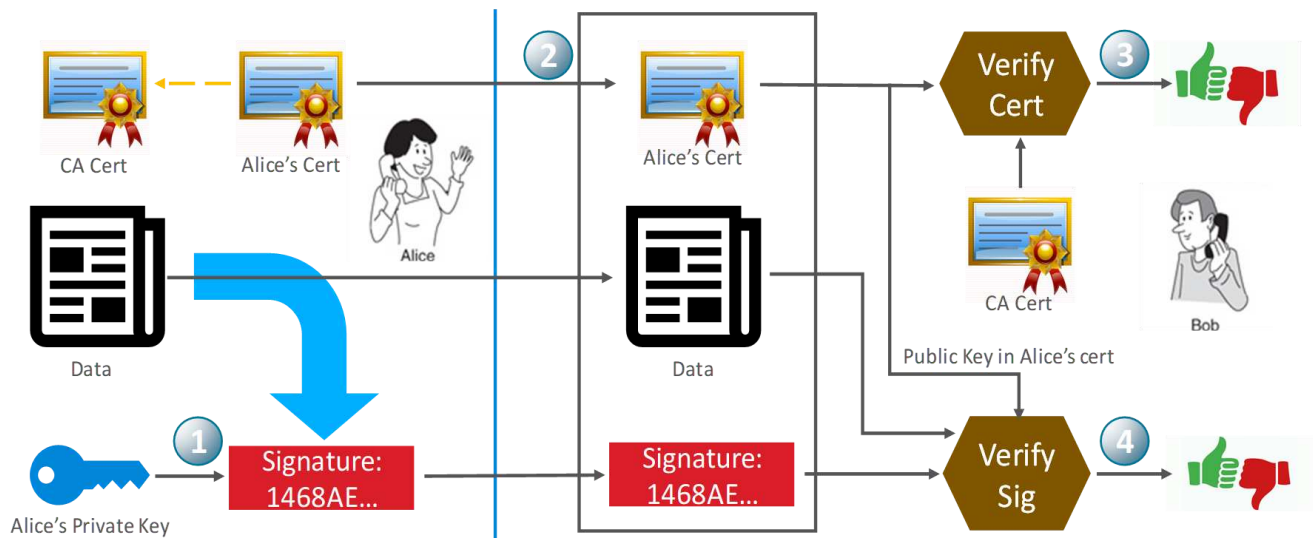
**Note:** The private key is never included as part of the certificate – it must be stored separately and kept private. The security of the scheme relies on protecting the private keys.

## Digital Certificates Verification

This section illustrates the process shown in [Digital Signature Example](#), but using digital certificates.

## Digital Certificates Verification Example

Alice wants to give data to Bob, signed with her private key. Alice has a digital certificate signed by a trusted third party (CA) in addition to her private key. Bob has a certificate from the trusted CA but nothing else is previously shared.



1. Alice uses her private key to sign the data.
2. Alice gives the data, signature, and her certificate to Bob.
3. Bob first verifies that Alice's certificate is valid, to prove Alice is the owner of the certificate's public key. This is done by verifying that Alice's certificate contains a valid signature created by the CA.
4. Bob then verifies the signature of the data using the public key in Alice's certificate.

**Note:** The hash process in [Digital Signature Example](#) is skipped in this example.

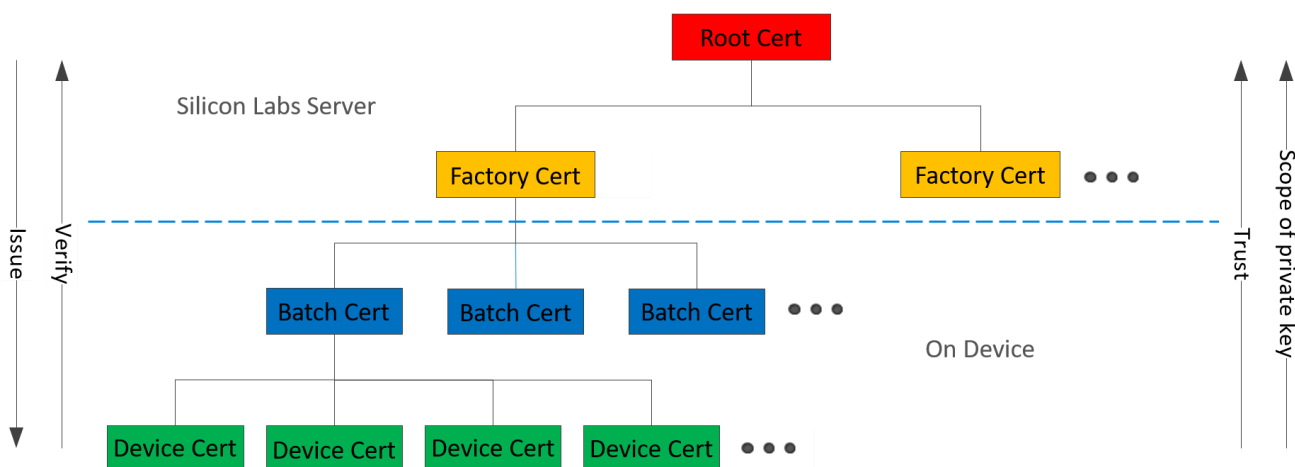
## Secure Identification on HSE-SVH Devices

# Secure Identification on HSE-SVH Devices

The goal of secure identification is to prove the ownership of a device's unique public key to an external service. It enables the external service to identify the device as legitimate and to authenticate device-generated data or messages.

## Chain of Trust

The chain of trust on HSE-SVH devices is illustrated in the following figure.



- Silicon Labs is a Certificate Authority (CA).
- The root certificate and factory certificate are stored in the Silicon Labs Server.
- The factory certificate is static per factory.
- The batch certificate and device certificate are stored on the device.
- The batch certificate is rolled per production batch.
- The device certificate is a unique cryptographic identity.
- All certificates are X.509 standard format.
  - TLS-compliant: Standard endpoint authentication methods are used in internet communications
  - Signature algorithm: ECDSA-prime256v1 with SHA256
- Each certificate in the chain is signed by the certificate above it ([Signing and Verification figure](#)).

**Note:** A certificate can be revoked if needed, for instance if security issues arise. The certificate revocation lists are stored in the Silicon Labs Server.

## Device Certificate

The device certificate example is described in the following figure.

Certificate:

Data: All data is hashed using the algorithm (SHA256) specified in the certificate

Version: 3 (0x2)

Serial Number:  
f2:59:94:21:76:1e:81:be:2b:6a:dd:09:1e:f2:fb:74:46:2c:20:b3

Signature Algorithm: ecdsa-with-SHA256

Issuer: CN = Batch 1001317, O = Silicon Labs Inc., C = US

Validity  
Not Before: Nov 19 14:30:15 2019 GMT  
Not After : Nov 19 14:30:15 2119 GMT

Subject: C = US, O = Silicon Labs Inc., CN = EUI:14B457FFFE0F7777 DMS:086AEC3CE650543EE73568DA S:SE0 ID:MCU

Subject Public Key Info:  
Public Key Algorithm: id-ecPublicKey  
Public-Key: (256 bit)  
pub:  
04:bd:7d:3b:3f:2b:de:9e:91:07:92:00:26:b5:25:  
de:5a:f7:27:ac:48:89:c3:0d:c3:e7:3f:96:19:3e:  
2a:07:14:0b:e5:b1:34:6b:53:9a:52:76:2e:63:7d:  
eb:9f:e7:47:15:33:9f:10:d2:08:0b:eb:3a:2e:66:  
83:34:4a:43:38

ASN1 OID: prime256v1  
NIST CURVE: P-256  
X509v3 extensions:  
X509v3 Basic Constraints: critical  
CA:FALSE  
X509v3 Key Usage: critical  
Digital Signature, Non Repudiation, Key Encipherment  
X509v3 Extended Key Usage: critical

TLS Web Client Authentication

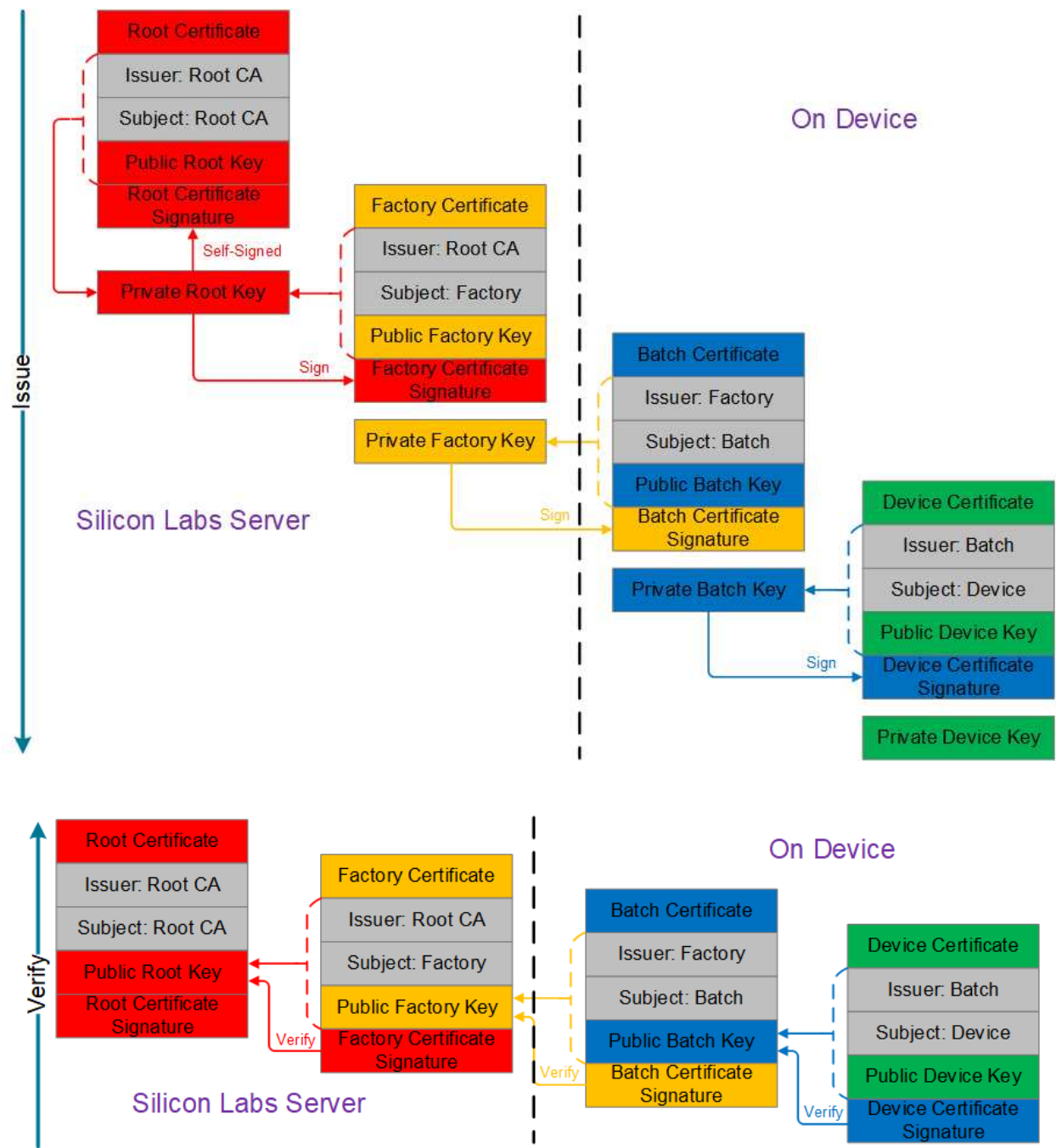
Signature Algorithm: ecdsa-with-SHA256  
30:46:02:21:00:9e:7a:65:a9:a4:be:e3:a3:00:77:d6:d0:68:  
1f:64:9a:ee:4f:9c:f3:1b:4b:58:f8:75:55:f8:48:f5:de:9a:  
73:02:21:00:c3:49:be:4d:54:07:22:95:f1:c3:84:72:f0:17:  
1f:92:1a:cf:6d:b9:ea:89:fa:af:1f:55:c8:0e:d2:ac:e0:a0

The hash is signed (ECDSA) using the Issuer's private key

- The device certificate is in X.509 DER format (~0.5 kB).
- The device certificate is stored in HSE one-time programmable memory (OTP). It cannot be modified once programmed.
- The batch number ( Issuer: CN = Batch field) identifies the factory and batch in which the device was produced.
- The validity period is 100 years from device manufacture date.
- The device 64-bit hard-coded unique ID (EUI) is encoded in the Subject: CN field, which blinds this certificate to the device.
- The device-specific public key is embedded in the device certificate and the corresponding private key is securely stored in the Secure Key Storage on the chip.
- The Issuer's private key is used to sign the hash of the certificate data to create a device certificate signature.

## Signing and Verification

Signing and verification for certificates on HSE-SVH devices are described in the following figures.



## Device Certificate Options

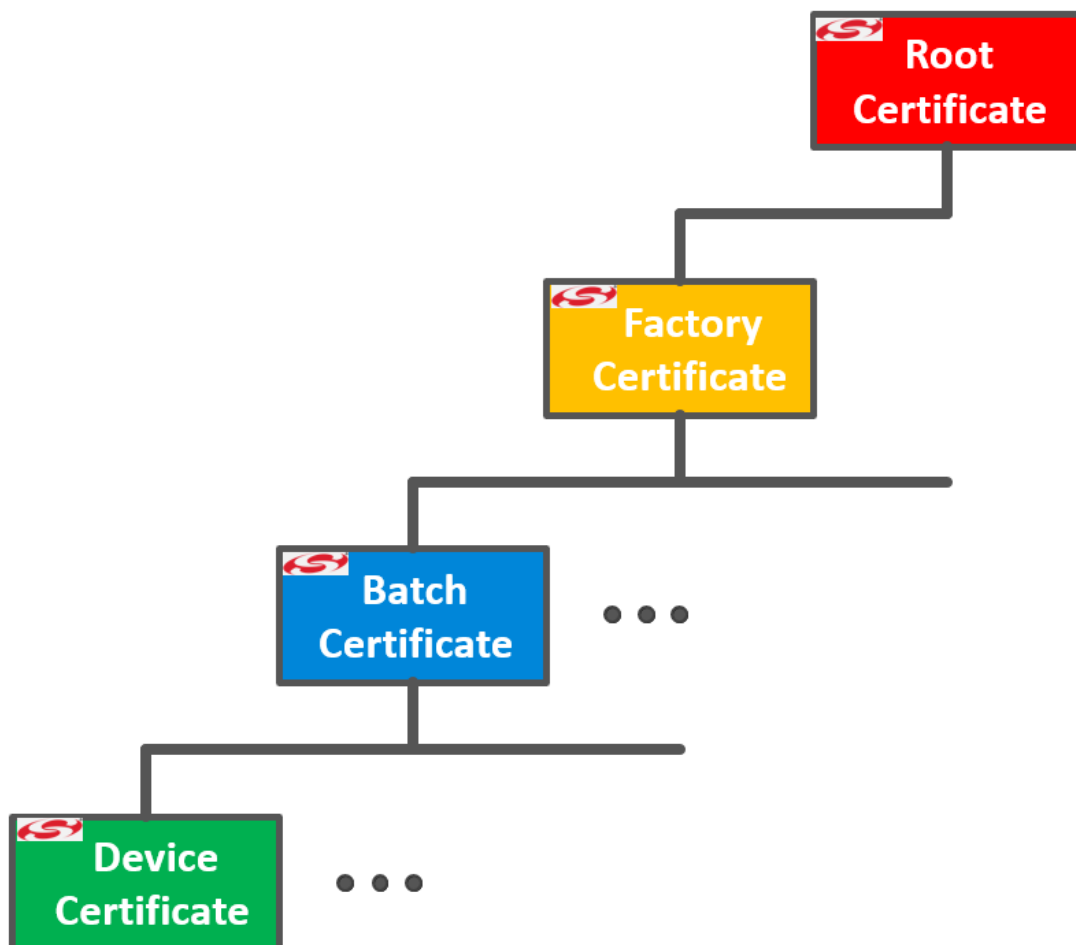
# Device Certificate Options

The HSE-SVH devices are each programmed with a device certificate during IC production. The device certificate is signed with a Public Device Key, using a Private Batch Key that can be validated against a Silicon Labs certificate chain [Verification for Certificates](#) and [Certificate Chain Verification](#). The device private key never leaves the Secure Key Storage on the chip. Customers can create their own device certificates during their production.

Three device certificate options (standard, modified, and external) are provided to meet different requirements. Silicon Labs provides [Custom Part Manufacturing Service \(CPMS\)](#) to program custom certificates on your chips at the Silicon Labs factories. For more information about CPMS, see [UG519: Custom Part Manufacturing Service User's Guide](#).

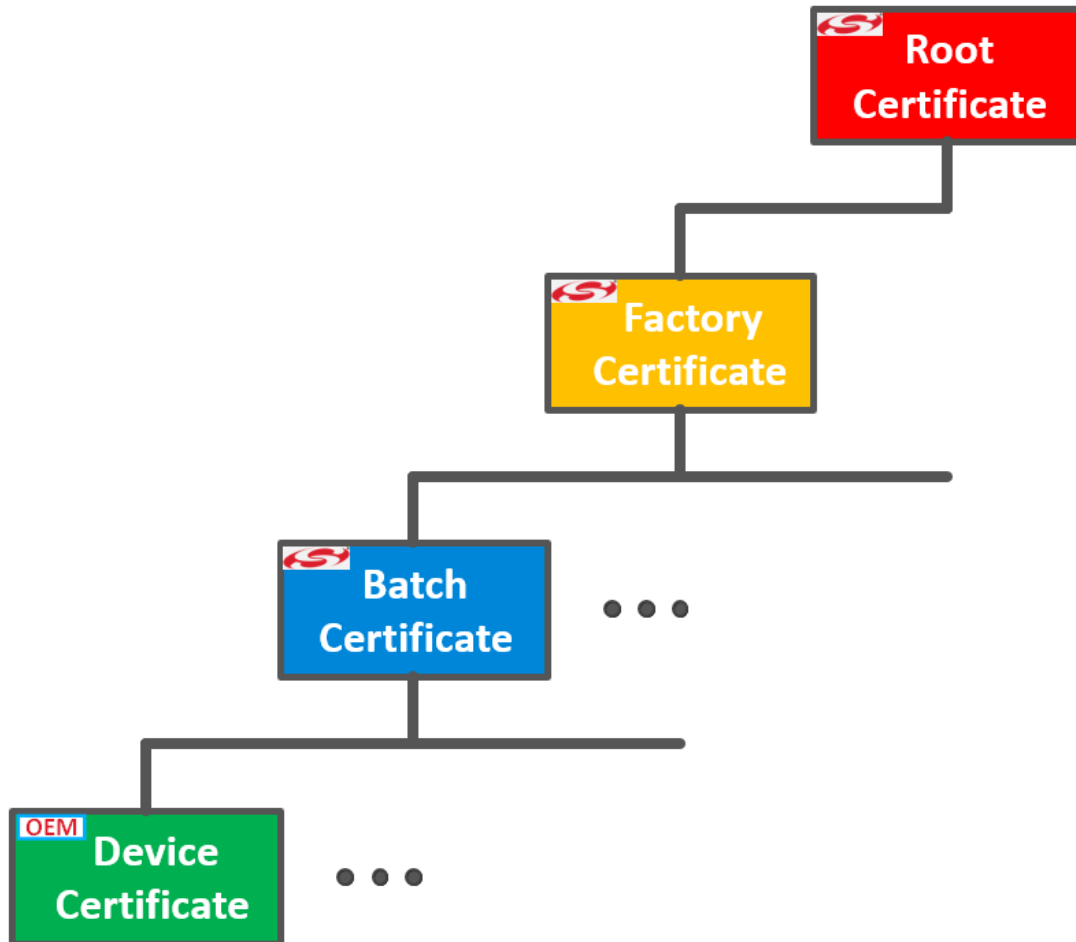
## Standard Device Certificate

- Comes standard with HSE-SVH devices.
- Cryptographically proves the device is an authentic Silicon Labs device.
- Does not protect against overproduction or counterfeit products that are built with authentic Silicon Labs devices.
- Signed to a Silicon Labs Certificate Authority (CA).
- The device can prove that it possesses the private key associated with the public key in its certificate by signing the response to a given challenge ([Remote Authentication Process](#) and [Certificate Chain Verification and Remote Authentication](#)).



## Modified Device Certificate

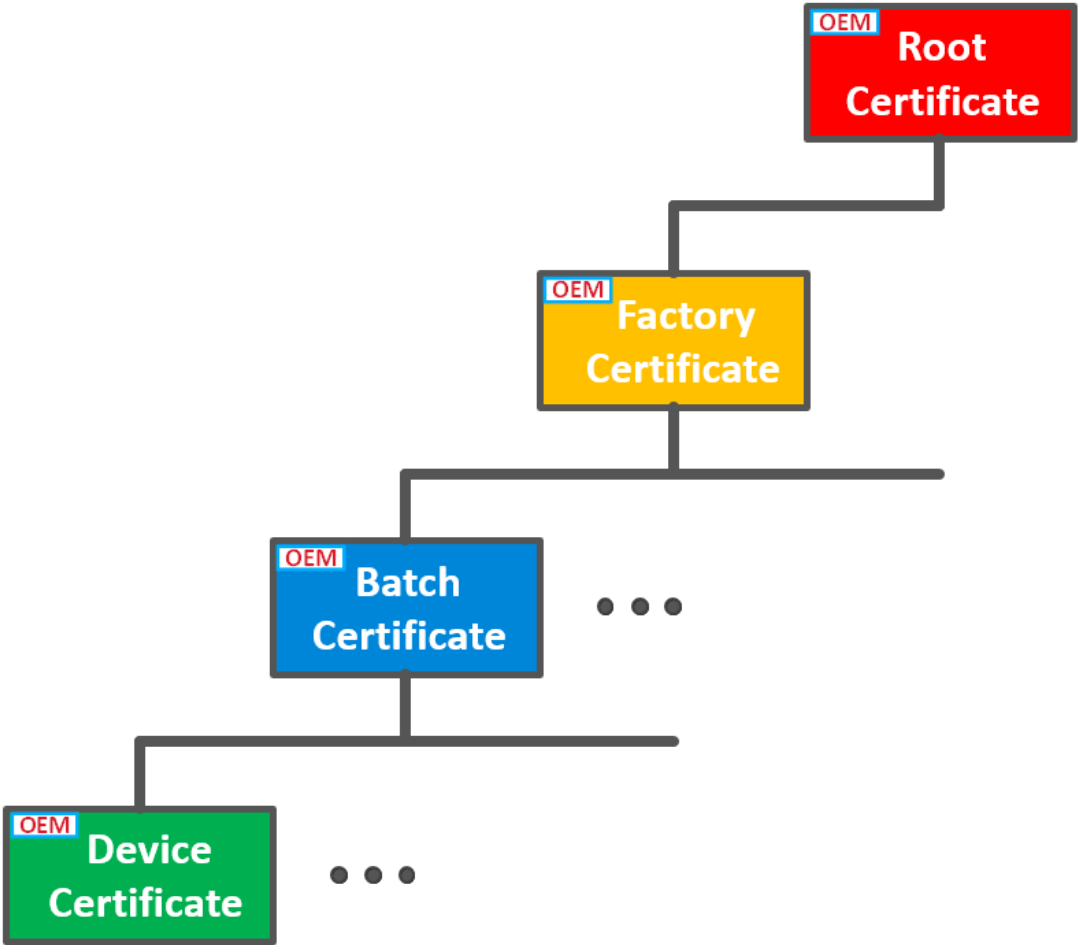
- Available as a customization service on HSE-SVH devices (OEM custom part number).
- Cryptographically proves the device is an authentic Silicon Labs device that was produced for a specific OEM.
- Protects against overproduction by Contract Manufacturer (CM).
- Device Certificate X.509 fields can be specified, with restrictions.
- Signed to a Silicon Labs Certificate Authority (CA).



## External Device Certificate

- Available as a customization service on HSE-SVH devices (OEM custom part number).
- Cryptographically proves the device is an authentic Silicon Labs device that was produced for a specific OEM.
- Protects against overproduction by Contract Manufacturer (CM).
- Factory Certificate is custom for each OEM.
- Device Certificate and Factory Certificate X.509 fields can be specified, with restrictions.
- Signed to a OEM Certificate Authority (CA).
- Root Certificate Authority is OEM-specified and is optional.
- Electronic delivery of all batch and device certificates signed under this OEM factory certificate is supported.





## Entity Attestation Token (EAT)

# Entity Attestation Token (EAT)

The device attestation service creates a token that contains a fixed set of device-specific data when requested from the caller. The device must contain an attestation key pair, which is unique per device, to sign the token. The HSE-SVH device uses the [Private Device Key](#) (aka attestation key) to sign the token, and the caller uses the Public Device Key to verify the token's authenticity.

An Entity Attestation Token (EAT) is a mini-report that is cryptographically signed. An EAT is encoded in either one of two standardized data formats: a Concise Binary Object Representation ([CBOR](#)) or in the text-based format JSON. A digital signature is then used to protect its content. The technical specification defining the content of the EAT, which are claims about the hardware and the software running on a device, is specified by the Internet Engineering Task Force ([IETF](#)).

An EAT is a collection of Key ID-Value pairs relating to device pedigree or any other information one wants the device to attest. Collected data can originate from the Root of Trust (RoT), any protected area, or non-protected areas.

The EAT must be signed following the structure of the CBOR Object Signing and Encryption ([COSE](#)) specification. For asymmetric key algorithms, the signature structure must be COSE-Sign1. A COSE-Sign1 is a CBOR encoded, self-secured data blob that contains headers, a payload, and a signature.

The primary need for EAT verification is to check correct formatting and verify signatures as for any token. In addition, though, the verifier can operate a policy where values of some of the claims in this profile can be compared to reference values, registered with the verifier for a given deployment, to confirm that the device is endorsed by the manufacturer supply chain.

The HSE can generate the [PSA attestation token or security configuration token](#) when requested from the caller with a [challenge](#) (Auth challenge claim below). The following tables describe EAT claims that are used in the [PSA attestation token](#) and security configuration token.

**Note:** The actual claims returned from the tokens are HSE firmware version dependent.

Claims of PSA Attestation Token

Key ID	Claim	Description	Value
-75000	Profile definition	Name of a document that describes the profile of the report.	PSA_IOT_PROFILE_1
-75001	Client ID	Represents the Partition ID of the caller.	See note below
-75002	Security lifecycle	Represents the current life cycle stage of the PSA RoT.	Device dependent
-75003	Implementation ID	Uniquely identifies the underlying immutable PSA RoT.	Device dependent (32 bytes)
-75004	Boot seed	Represents a random value created at system boot time.	Random bytes (32 bytes)
-75006	Software components	A list of Software components represents all the software loaded by PSA RoT.	See the software components table below.
-75008	Auth challenge	Input object from the caller. For example, this can be a cryptographic nonce or a hash of locally attested data. The length must be 32, 48, or 64 bytes.	Random bytes or hash (32/48/64 bytes)

Key ID	Claim	Description	Value
-75009	Instance ID	Unique identifier of the instance.	Device EUI-64 unique ID with type byte 0x06 (9 bytes)

Note:

- Key ID 75001: Client ID if present. Otherwise the value 1 for a token requested by a secure bus master and -1 for a non-secure master.
- Key ID 75002 (For the definitions of these lifecycle states, please refer to the ARM [Platform Security Model](#)):
  - UNKNOWN ( 0x0000 )
  - ASSEMBLY\_AND\_TEST ( 0x1000 )
  - PSA\_ROT\_PROVISIONING ( 0x2000 )
  - SECURED ( 0x3000 )
  - NON\_PSA\_ROT\_DEBUG ( 0x4000 )
  - RECOVERABLE\_PSA\_ROT\_DEBUG ( 0x5000 )
  - DECOMMISSIONED ( 0x6000 )
- Key ID 75003:
  - Word[0]: Die revision
  - Word[1]: HSE OTP version
  - Word[2]: Bit indicating it is an HSE-SVH device
  - Word[3]: Production version
  - Word[4:7]: Reserved (zeros)

Software Components

Key ID	Type	Description	Value
1	Measurement type	A short string represents the role of this software component.	See note below
2	Measurement value	Represents a hash of the invariant software component in memory at startup time.	See note below
4	Version	The issued software version is in the form of a text string.	See note below

Notes:

- Key ID 1:
  - HSE always exists — **PRoT**
  - If secure booted Gecko Bootloader exists at flash starting address — **BL**
  - If secure booted application exists at flash starting address — **ARoT**
- Key ID 2: SHA-256 hash (32 bytes) of the firmware (HSE, Gecko Bootloader, or application)
- Key ID 4: Version of the firmware (HSE, Gecko Bootloader, or application)

Claims of Security Configuration Token

Key ID	Claim	Description	Value
-75000	Profile definition	Name of a document that describes the profile of the report.	SILABS_1
-75008	Auth challenge	Input object from the caller. For example, this can be a cryptographic nonce or a hash of locally attested data. The length must be 32 bytes.	Random bytes or hash (32 bytes)
-75009	Instance ID	Unique identifier of the instance.	Device EUI-64 unique ID with type byte 0x06 (9 bytes)
-76000	SE status	Device HSE status.	Device dependent (36 bytes)

Key ID	Claim	Description	Value
-76001	OTP configuration	Device HSE OTP configuration if provisioned.	Device dependent (24 bytes)
-76002	Sign Key	Public Sign Key in HSE OTP if provisioned.	Device dependent (64 bytes)
-76003	Command Key	Public Command Key in HSE OTP if provisioned.	Device dependent (64 bytes)
-76004	Tamper settings	Current applied tamper settings.	Device dependent (16 bytes)

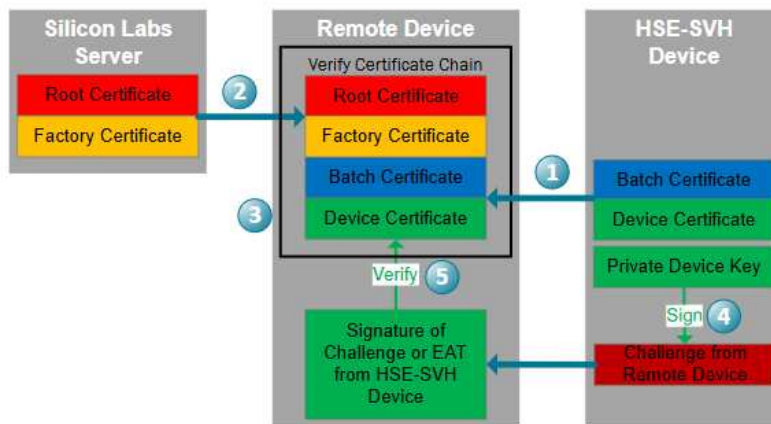
Notes:

- All custom Silicon Labs claims will have a base of 76000.
- Key ID 76000: Refer to section "Get Status" in [AN1303: Programming Series 2 Devices using the Debug Challenge Interface \(DCI\) and Serial Wire Debug \(SWD\)](#) for the description (HSE-SVH) of the value.
- Key ID 76001: Refer to section "Read User Configuration" in [AN1303: Programming Series 2 Devices using the Debug Challenge Interface \(DCI\) and Serial Wire Debug \(SWD\)](#) for the description (HSE-SVH) of the value.
- Key ID 76002 and 76003: Refer to [Key Reference](#) for Public Sign Key and Public Command Key.
- Key ID 76004: One nibble per tamper source. Refer to section "Anti-Tamper Configuration" in [AN1303: Programming Series 2 Devices using the Debug Challenge Interface \(DCI\) and Serial Wire Debug \(SWD\)](#) for the description of the value.

## Remote Authentication Process

# Remote Authentication Process

Remote authentication is used to manage attestation by requesting that the device sign a challenge or EAT based on its secure identity.



1. The remote device requests the device certificate and batch certificate from the HSE-SVH device.
2. The remote device looks up the factory certificate and root certificate from the Silicon Labs Server.
3. The remote device validates each certificate in the chain using the public key of each Issuer ([Verification for Certificates](#)).
4. The remote device then sends an attestation challenge (random number) to the HSE-SVH device. The HSE-SVH device uses the Private Device Key in the Secure Key Storage on the chip to sign the challenge or EAT and sends the signature of challenge or EAT to the remote device.
5. The remote device requires a small library to validate the signature of challenge or EAT using the Public Device Key in the device certificate.

Secure Engine Manager

# Secure Engine Manager

The Secure Engine Manager provides thread-safe APIs for the SE's mailbox interface. The following table lists the SE Manager APIs related to secure identity. The SE Manager API document can be found at <https://docs.silabs.com/gecko-platform/latest/service/api/group-sl-se-manager>.

For the SE's mailbox interface, see section *Secure Engine Subsystem* in [Series 2 Secure Debug](#).

SE Manager API for Security Identity

SE Manager API	Usage
sl_se_read_pubkey	Read stored Public Device Key in the HSE-SVH device.
sl_se_read_cert	Read stored certificates (DER format) in the HSE-SVH device.
sl_se_read_cert_size	Read the size of stored certificates in the HSE-SVH device.
sl_se_attestation_get_psa_iat_token	Get the PSA attestation token from the HSE with the given nonce.
sl_se_attestation_get_psa_iat_token_size	Get the size of a PSA attestation token with the given nonce.
sl_se_attestation_get_config_token	Get the security configuration token from the HSE with the given nonce.
sl_se_attestation_get_config_token_size	Get the size of a security configuration token with the given nonce.

Examples

# Examples

## Overview

The secure device authentication examples are described in the following table.

Secure Device Authentication Examples			
Example	Device (Radio Board)	HSE Firmware	Tool
Certificate chain verification	EFR32MG21B010F1024IM32 (BRD4181C)	Version 1.2.9	Simplicity Commander and OpenSSL
Certificate chain verification	EFR32MG21B010F1024IM32 (BRD4181C)	Version 1.2.9	Simplicity Commander
Certificate chain verification	EFR32MG21B010F1024IM32 (BRD4181C)	Version 1.2.9	Simplicity Studio 5
Certificate chain verification & Remote authentication	EFR32MG21B010F1024IM32 (BRD4181C)	Version 1.2.9	SE Manager and Mbed TLS
Entity Attestation Token (EAT)	EFR32MG21B010F1024IM32 (BRD4181C)	Version 1.2.9	SE Manager
Entity Attestation Token (EAT)	EFR32MG21B010F1024IM32 (BRD4181C)	Version 1.2.9	Simplicity Commander

**Note:** Unless specified in the example, these examples can apply to other HSE-SVH devices.

Users can download the device root certificate ( `Device-Root-CA-chain.pem` ) and factory certificate ( `Factory-chain.pem` ) from <https://www.silabs.com/certificate-authority>.

# Certificate Practice Statement

## Active Public Certificates:

- [Device Root Certificate](#)
- [Factory Certificate](#)
- [Zentri DMS Certificate](#)

## CRL Links:

- [Device Root Certificate Revocation List](#)
- [Factory Certificate Revocation List](#)
- [Zentri DMS Certificate Revocation List](#)

For more information contact: [certificateauthority@silabs.com](mailto:certificateauthority@silabs.com)

For Simplicity Studio v5.3.0.0 and higher, the device root certificate ( `device-root-prod.pem` ) and factory certificate ( `factory-prod.pem` ) can be found in the Window folder below.

C:\SiliconLabs\SimplicityStudio\v5\offline\common\certificates

## Using Simplicity Commander

1. This application note uses Simplicity Commander v1.11.2. The procedures and console output may be different on the other versions of Simplicity Commander. The latest version of Simplicity Commander can be downloaded from <https://www.silabs.com/developers/mcu-programming-options>.

```
commander --version
```

```
Simplicity Commander 1v11p2b998
```

```
JLink DLL version: 6.94d
```

```
Qt 5.12.1 Copyright (C) 2017 The Qt Company Ltd.
```

```
EMDLL Version: 0v17p18b581
```

```
mbd TLS version: 2.6.1
```

```
DONE
```

2. The Simplicity Commander's Command Line Interface (CLI) is invoked by `commander.exe` in the Simplicity Commander folder. The location for Simplicity Studio 5 in Windows is `C:\SiliconLabs\SimplicityStudio\v5\developer\adapter_packs\commander`. For ease of use, it is highly recommended to add the path of `commander.exe` to the system `PATH` in Windows.
3. If more than one Wireless Starter Kit (WSTK) is connected via USB, the target WSTK must be specified using the `--serialno \<J-Link serial number>` option.
4. If the WSTK is in debug mode OUT, the target device must be specified using the `--device \<device name>` option.

For more information about Simplicity Commander, see [UG162: Simplicity Commander Reference Guide](#).

## Using an External Tool

The [certificate chain verification](#) example uses the `OpenSSL` to validate the certificate chain. The Windows version of `OpenSSL` can be downloaded from <https://slproweb.com/products/Win32OpenSSL.html>. This application note uses `OpenSSL`



Version 1.1.1h (Win64).

```
openssl version
```

```
OpenSSL 1.1.1h 22 Sep 2020
```

The OpenSSL's Command Line Interface (CLI) is invoked by `openssl.exe` in the OpenSSL folder. The location in Windows (Win64) is `C:\Program Files\OpenSSL-Win64\bin`. For ease of use, it is highly recommended to add the path of `openssl.exe` to the system `PATH` in Windows.

## Using Platform Examples

Simplicity Studio 5 includes the [SE Manager platform examples](#) for secure identity and attestation. This application note uses platform example of GSDK v3.2.3. The console output may be different on the other versions of GSDK.

Refer to the corresponding `readme.html` file for details about each SE Manager platform example. This file also includes the procedures to create the project and run the example.

## Certificate Chain Verification

Certificate chain verification is the process of making sure a given certificate chain is well-formed, valid, properly signed, and trustworthy. The certificate signature is verified using the public key in the issuer certificate ([Verification for Certificates](#)).

## Simplicity Commander and OpenSSL

1. Run the `security readcert` command to save the batch certificate in PEM format.

```
commander security readcert batch -o batch.pem --serialno 440030580
```

```
Writing certificate to batch.pem...  
DONE
```

2. Run the `security readcert` command to save the device certificate in PEM format.

```
commander security readcert mcu -o device.pem --serialno 440030580
```

```
Writing certificate to device.pem...  
DONE
```

3. Get the root certificate ( `device-root-prod.pem` ) and factory certificate ( `factory-prod.pem` ) from the certificate folder in [Simplicity Studio](#).
4. Use OpenSSL to display the certificate information (e.g., `device.pem` ).

```
openssl x509 -in device.pem -text -noout
```

```

Certificate:
  Data:
    Version: 3 (0x2)
    Serial Number:
      66:f8:5a:e6:b4:ef:6e:49:d3:36:95:63:c9:c3:99:13:e4:71:93:f6
    Signature Algorithm: ecdsa-with-SHA256
    Issuer: CN = Batch 1001317, O = Silicon Labs Inc., C = US
    Validity
      Not Before: Nov 19 15:10:33 2019 GMT
      Not After : Nov 19 15:10:33 2119 GMT
    Subject: C = US, O = Silicon Labs Inc., CN = EUI:14B457FFFE0F77CE DMS:086AEC3C645836BFB04D312F S:SE0 ID:MCU
    Subject Public Key Info:
      Public Key Algorithm: id-ecPublicKey
      Public-Key: (256 bit)
      pub:
        04:5c:4b:c9:b0:b3:ff:fa:99:81:c5:99:be:ff:ae:
        77:74:1a:f4:30:f1:1e:0e:2d:df:96:4b:ff:d2:46:
        fa:fa:e7:23:4b:79:cb:0a:c7:71:13:fa:7c:39:5f:
        e2:18:9e:4e:06:43:88:a7:9c:65:53:f3:a3:a1:06:
        81:e6:06:f2:11
      ASN1 OID: prime256v1
      NIST CURVE: P-256
    X509v3 extensions:
      X509v3 Basic Constraints: critical
        CA:FALSE
      X509v3 Key Usage: critical
        Digital Signature, Non Repudiation, Key Encipherment
      X509v3 Extended Key Usage: critical
        TLS Web Client Authentication
    Signature Algorithm: ecdsa-with-SHA256
    30:44:02:20:57:12:a4:84:d8:37:b8:c0:44:8f:16:ac:c1:a3:
    be:a9:f1:16:38:9f:b9:a2:57:e6:12:49:bf:96:a9:a9:d2:b8:
    02:20:5f:ae:22:f5:00:05:49:b1:da:ee:4a:84:48:70:27:97:
    1c:40:2d:85:5f:f2:12:b3:8b:4a:d7:9a:ee:60:81:7c

```

5. Use OpenSSL to verify the certificate chain from steps 1 to 3.

```
openssl verify -show_chain -CAfile device-root-prod.pem -untrusted factory-prod.pem -untrusted batch.pem device.pem
```

```

device.pem: OKChain:
depth=0: C = US, O = Silicon Labs Inc., CN = EUI:14B457FFFE0F7777 DMS:086AEC3CE650543EE73568DA S:SE0 ID:MCU (untrusted)
depth=1: CN = Batch 1001317, O = Silicon Labs Inc., C = US (untrusted)
depth=2: CN = Factory, O = Silicon Labs Inc., C = US (untrusted)
depth=3: CN = Device Root CA, O = Silicon Labs Inc., C = US

```

## Simplicity Commander

Run the `security readcert` command to display the key information about the on-chip certificates (e.g., mcu).

```
commander security readcert mcu --serialno 440030580
```

```
Version      : 3
Subject      : C=US O=Silicon Labs Inc. CN=EUI:14B457FFFE0F77CE DMS:086AEC3C645836BFB04D312F S:SE0 ID:MCU
Issuer       : CN=Batch 1001317 O=Silicon Labs Inc. C=US
Valid From   : November 19 2019
Valid To     : November 19 2119
Signature algorithm: SHA256
Public Key Type : ECDSA
Public key    : 5c4bc9b0b3fffa9981c599beffae77741af430f11e0e2ddf964bffd246fafa7
              234b79cb0ac77113fa7c395fe2189e4e064388a79c6553f3a3a10681e606f211
DONE
```

Run the `security attestation` command to verify the on-chip batch and device certificates with root and factory certificates.

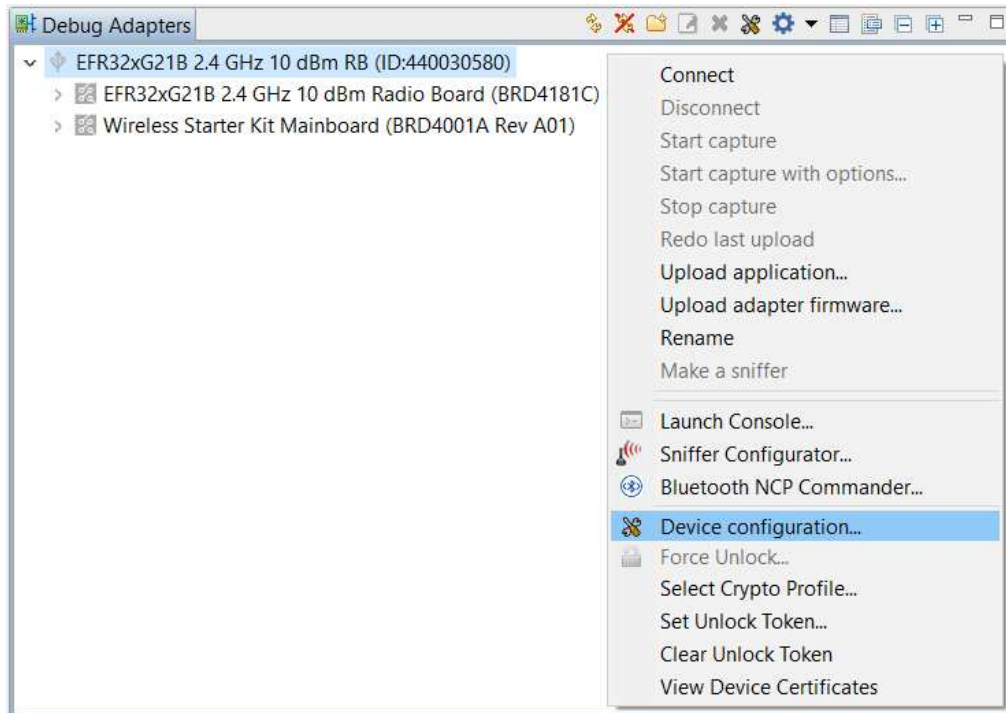
```
commander security attestation --serialno 440030580
```

```
Certificate chain successfully validated up to Silicon Labs device root certificate.
-75008 ARM PSA nonce      : 05a88aeef627dd663058e3d758fe9a827942da0793da72af81c79a4f60fa9824
-75000 ARM PSA Profile ID : SILABS_1
-75009 ARM PSA/IETF EAT UEID : 0614b457ffe0f77ce
-76000 SE status          : 000000001000000000000000000000003a90000002000010209ffffffff0000002500000000
-76001 OTP configuration  : 00000000100444400401041411224477242204420a060005
-76002 MCU sign key       : c4af4ac69aab9512db50f7a26ae5b4801183d85417e729a56da974f4e08a562c
                          de6019dea9411332dc1a743372d170b436238a34597c410ea177024de20fc819
-76003 MCU command key    : b1bc6f6fa56640ed522b2ee0f5b3cf7e5d48f0be8148f0dc08440f0a4e1dca4
                          7c04119ed6a1be31b7707e5f9d001a659a051003e95e1b936f05c37ea793ad63
-76004 Current applied tamper settings : 15044440040104141122447714220442
Successfully validated signature of attestation token.
DONE
```

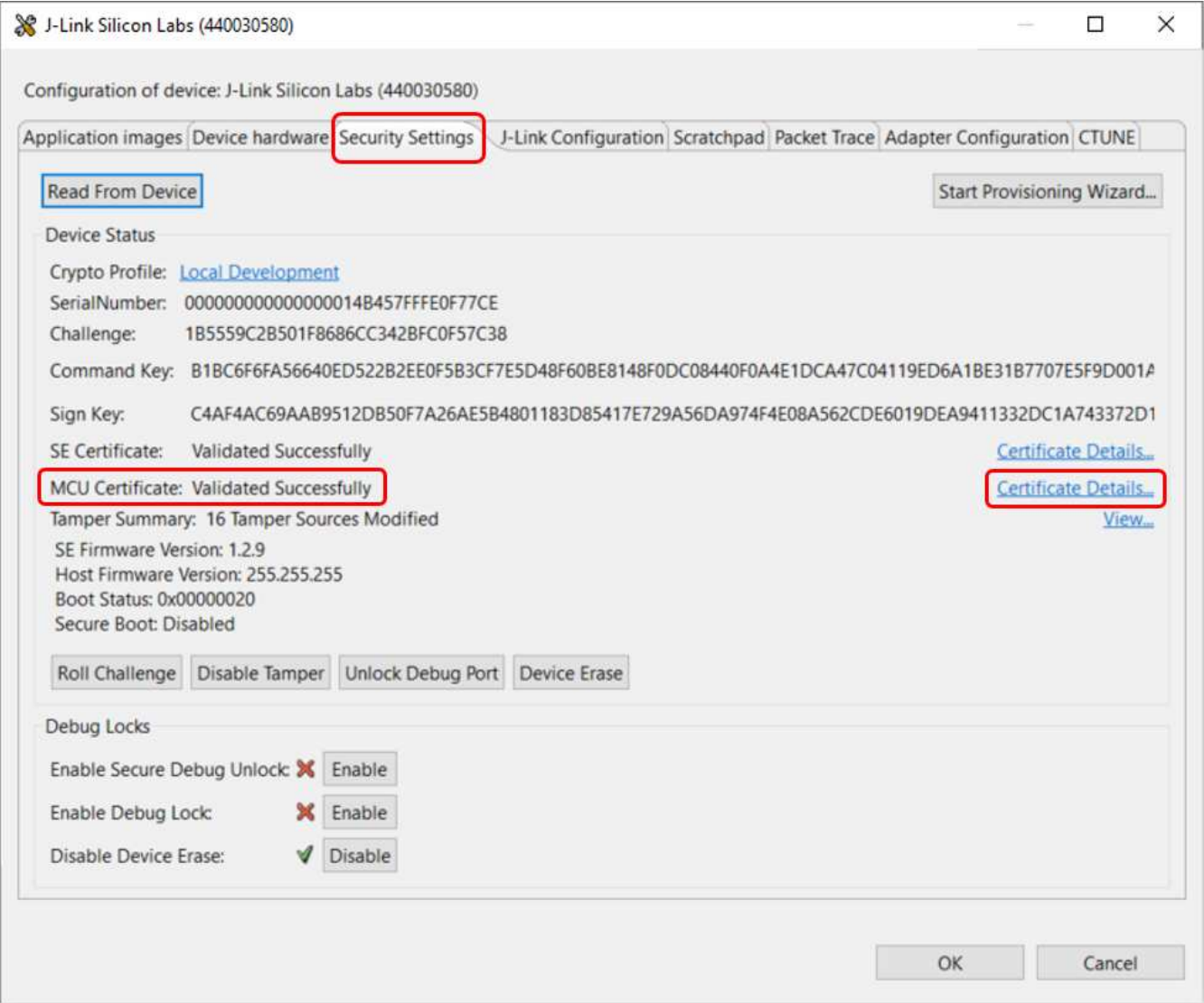
## Simplicity Studio

This application note uses Simplicity Studio v5.2.1.1. The procedures and pictures may be different on the other versions of Simplicity Studio 5.

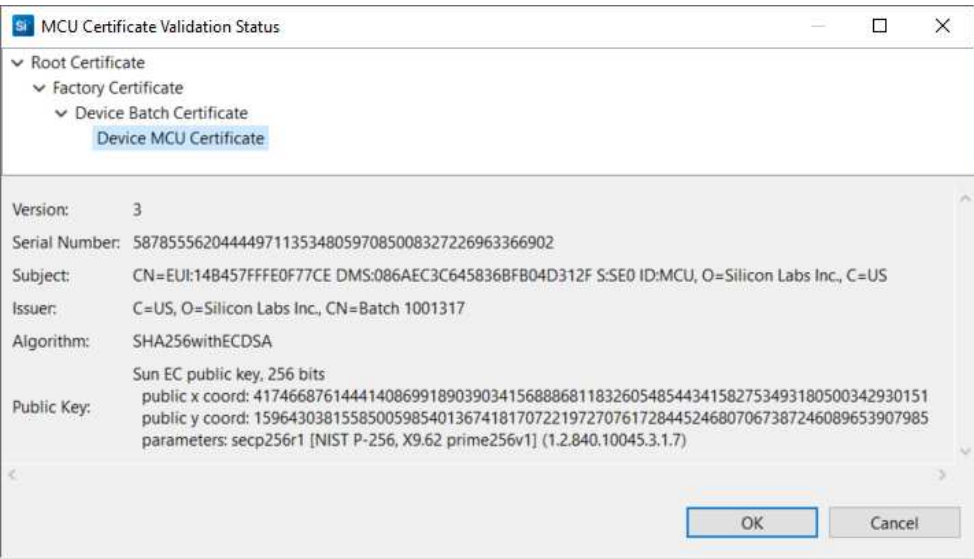
1. Right-click the selected debug adapter **RB (ID:J-Link serial number)** to display the context menu.



2. Click **Device configuration...** to open the **Configuration of device: J-Link Silicon Labs (serial number)** dialog box. Click the **Security Settings** tab to get the selected device configuration.
3. The **MCU Certificate:** will display **Validated Successfully** if it passed the certificate chain verification process.



4. Click **Certificate Details...** to browse the details of different certificates (e.g., Device MCU Certificate in the figure below).



## Certificate Chain Verification and Remote Authentication

The SE Manager Secure Identity platform example uses APIs in [SE Manager](#) and Mbed TLS to emulate the processes in [Remote Authentication Process](#).

Click the [View Project Documentation](#) link to open the `readme.html` file.

### Platform - SE Manager Secure Identity

This example project demonstrates the secure identity of Secure Vault High device.

[CREATE](#)[View Project Documentation](#)

The HSE-SVH device simulates the operations in the remote device to eliminate the communications between different parties in this example. The factory certificate and root certificate are hard-coded in the `app_mbedtls_x509.c` file.

The Private Device Key in the Secure Key Storage on the chip is used to sign the challenge from the remote device. Therefore this example can only run on a chip with the [Standard Device Certificate](#).

### Step 1 in the Remote Authentication Process

```
SE Manager Secure Identity Example - Core running at 38000 kHz.
. SE manager initialization... SL_STATUS_OK (cycles: 6 time: 0 us)

. Secure Vault High device:
+ Read size of on-chip certificates... SL_STATUS_OK (cycles: 5296 time: 139 us)
+ Read on-chip device certificate... SL_STATUS_OK (cycles: 5138 time: 135 us)
+ Parse the device certificate (DER format)... SL_STATUS_OK (cycles: 167043 time: 4395 us)
+ Get the public device key in device certificate... OK
+ Read on-chip batch certificate... SL_STATUS_OK (cycles: 5080 time: 133 us)
+ Parse the batch certificate (DER format)... SL_STATUS_OK (cycles: 173151 time: 4556 us)
```

### Steps 2 and 3 in the Remote Authentication Process (certificate chain printout is disabled)

```
. Remote device:
+ Parse the factory certificate (PEM format)... SL_STATUS_OK (cycles: 5373122 time: 141 ms)
+ Parse the root certificate (PEM format)... SL_STATUS_OK (cycles: 5448802 time: 143 ms)
+ Verify the certificate chain with root certificate... SL_STATUS_OK (cycles: 958730 time: 25229 us)
```

### Steps 2 and 3 in the Remote Authentication Process (certificate chain printout is enabled)

```
. Remote device:
+ Parse the factory certificate (PEM format)... SL_STATUS_OK (cycles: 5373935 time: 141 ms)
+ Parse the root certificate (PEM format)... SL_STATUS_OK (cycles: 5449622 time: 143 ms)
+ Verify requested for (Depth 3) ... OK
  cert. version   : 3
  serial number   : 12:E6:A2:A5:9C:AA:27:F9
  issuer name     : CN=Device Root CA, O=Silicon Labs Inc., C=USsubject name   : CN=Device Root CA, O=Silicon Labs Inc., C=US
  issued on      : 2018-10-10 17:32:00
  expires on     : 2118-09-16 17:32:00
  signed using    : ECDSA with SHA256
  EC key size     : 256 bits
  basic constraints : CA=true, max_pathlen=2
  key usage      : Digital Signature, Key Cert Sign, CRL Sign
+ Verify requested for (Depth 2) ... OK
  cert. version   : 3
  serial number   : 24:DC:7B:40:0C:32:9C:0A
  issuer name     : CN=Device Root CA, O=Silicon Labs Inc., C=USsubject name   : CN=Factory, O=Silicon Labs Inc., C=US
  issued on      : 2018-10-10 17:33:00
  expires on     : 2118-09-16 17:32:00
  signed using    : ECDSA with SHA256
  EC key size     : 256 bits
  basic constraints : CA=true, max_pathlen=1
  key usage      : Digital Signature, Key Cert Sign, CRL Sign
+ Verify requested for (Depth 1) ... OK
  cert. version   : 3
  serial number   : 23:09:DA:39:B4:78:05:AA
  issuer name     : CN=Factory, O=Silicon Labs Inc., C=USsubject name   : CN=Batch 1001317, O=Silicon Labs Inc., C=US
  issued on      : 2019-10-17 21:20:20
  expires on     : 2118-09-16 17:32:00
  signed using    : ECDSA with SHA256
  EC key size     : 256 bits
  basic constraints : CA=true, max_pathlen=0
  key usage      : Digital Signature, Key Cert Sign
+ Verify requested for (Depth 0) ... OK
  cert. version   : 3
  serial number   : 66:F8:5A:E6:B4:EF:6E:49:D3:36:95:63:C9:C3:99:13:E4:71:93:F6
  issuer name     : CN=Batch 1001317, O=Silicon Labs Inc., C=USsubject name   : C=US, O=Silicon Labs Inc., CN=EUI:14B457FFFE0F77CE
DMS:086AEC3C645836BFB04D312F S:SE0 ID:MCU
  issued on      : 2019-11-19 15:10:33
  expires on     : 2119-11-19 15:10:33
  signed using    : ECDSA with SHA256
  EC key size     : 256 bits
  basic constraints : CA=false
  key usage      : Digital Signature, Non Repudiation, Key Encipherment
  ext key usage   : TLS Web Client Authentication
+ Verify the certificate chain with root certificate... SL_STATUS_OK (cycles: 9703861 time: 255 ms)
```

**Note:** The longer processing time (255 ms) is due to the certificate chain printout.

## Steps 4 and 5 (signature of a challenge) in the Remote Authentication Process

```
. Remote authentication:
+ Create a 16 bytes challenge (random number) in remote device for signing... SL_STATUS_OK (cycles: 3700 time: 97 us)
+ Sign challenge with private device key in Secure Vault High device... SL_STATUS_OK (cycles: 221983 time: 5841 us)
+ Get public device key in Secure Vault High device... SL_STATUS_OK (cycles: 199788 time: 5257 us)
+ Verify signature with public device key in Secure Vault High device... SL_STATUS_OK (cycles: 229054 time: 6027 us)
+ Verify signature with public device key in remote device... SL_STATUS_OK (cycles: 230442 time: 6064 us)

. SE manager deinitialization... SL_STATUS_OK (cycles: 6 time: 0 us)
```

# Entity Attestation Token (EAT)

These examples demonstrate how to retrieve the [EAT tokens](#) from the HSE-SVH device.

## SE Manager - Attestation Platform Example

The SE Manager Attestation platform example uses APIs in [SE Manager](#) to retrieve the PSA attestation token and security configuration token from the HSE.

Click the [View Project Documentation](#) link to open the `readme.html` file.

### Platform - SE Manager Attestation

This example project demonstrates how to get attestation tokens using the SE Manager Attestation API and printing them in a human-readable format.

[View Project Documentation](#)

[CREATE](#)

Press `SPACE` to cycle the challenge size for the PSA attestation token. Press `ENTER` to make a selection and run the program.

```
SE Manager Attestation Example - Core running at 38000 kHz.
Initializing SE Manager...
SL_STATUS_OK (cycles: 10 time: 0 us)

Select nonce size for the IAT token (32, 48 or 64 bytes).
Press SPACE to cycle through the options.
Press ENTER to make a selection.
    Current nonce size: 32
    Selected nonce size: 32
Calling sl_se_attestation_get_psa_iat_token...

SL_STATUS_OK (cycles: 661072 time: 17396 us)
```

PSA Attestation Token ([Entity Attestation Token \(EAT\)](#) and [Entity Attestation Token \(EAT\)](#))



PSA IAT token

=====

-----

Raw token:

```
d28443a10126a058e4a83a000124ff58204ca14d0bc8601cad2e511de1964e93
9338b6fc20f8231aa178ca79519b0ffae73a000124f7715053415f494f545f50
524f46494c455f313a00012500490614b457ffe0f77ce3a000124f8013a0001
24f91920003a000124fa5820011c00010600000001000000f2030f0000000000
0000000000000000000000003a000124fb58204922b7bbd31c0c81c9b0485ccf
b5396ec24ffa877ece441e11c947b791218cf83a000124fd81a3016450526f54
046830303031303230390258206d39caedba129297062b820ba6d85b3e432c44
3c8a8a31d3c6232be6906d38dc584030f9d61523204793965fc9eb2be788db9d
2b02692d877673c86ebffb6769984515d2f1a287a92d2c134c1024f20f018d
be952a2ccae7ed2980a9f242d02c9c
```

COSE\_Sign1 structure:

```
d2 ;tag(18)
84 ;array(4)
43 ;byte_str(3)
  a10126
a0 ;map(0)
58 ;byte_str(228)
  a83a000124ff58204ca14d0bc8601cad2e511de1964e939338b6fc20f8231aa1
  78ca79519b0ffae73a000124f7715053415f494f545f50524f46494c455f313a
  00012500490614b457ffe0f77ce3a000124f8013a000124f91920003a000124
  fa5820011c00010600000001000000f2030f00000000000000000000000000
  0000003a000124fb58204922b7bbd31c0c81c9b0485ccfb5396ec24ffa877ece
  441e11c947b791218cf83a000124fd81a3016450526f54046830303031303230
  390258206d39caedba129297062b820ba6d85b3e432c443c8a8a31d3c6232be6
  906d38dc
58 ;byte_str(64)
  30f9d61523204793965fc9eb2be788db9d2b02692d877673c86ebffb676998
  4515d2f1a287a92d2c134c1024f20f018dbe952a2ccae7ed2980a9f242d02c9c
```

Token claims:

```
a8 ;map(8)
3a ;int(-75008)
58 ;byte_str(32)
  4ca14d0bc8601cad2e511de1964e939338b6fc20f8231aa178ca79519b0ffae7
3a ;int(-75000)
71 ;text_str(17)
  "PSA_IOT_PROFILE_1"
3a ;int(-75009)
49 ;byte_str(9)
  0614b457ffe0f77ce
3a ;int(-75001)
01 ;int(1)
3a ;int(-75002)
19 ;int(8192)
3a ;int(-75003)
58 ;byte_str(32)
  011c00010600000001000000f2030f00000000000000000000000000000000
3a ;int(-75004)
58 ;byte_str(32)
  4922b7bbd31c0c81c9b0485ccfb5396ec24ffa877ece441e11c947b791218cf8
3a ;int(-75006)
81 ;array(1)
a3 ;map(3)
  01 ;int(1)
  64 ;text_str(4)
  "PRoT"
  04 ;int(4)
  68 ;text_str(8)
  "00010209"
  02 ;int(2)
  58 ;byte_str(32)
```

[illegible]

## Security Configuration Token (Entity Attestation Token (EAT))

```
-----  
Calling sl_se_attestation_get_config_token...  
SL_STATUS_OK (cycles: 541281 time: 14244 us)
```

```
Config token
```

```
=====
```

```
-----  
Raw token:
```

```
d28443a10126a0590133a83a000124ff5820c3e3664dcc47711bf81734bc95f0  
87d81dd841d73fc805fc9237c7b3dfa25c503a000124f76853494c4142535f31  
3a00012500490614b457ffe0f77ce3a000128df582400000001000000000000  
00000000000000002000010209ffffff00000025000000003a000128e058  
1800000000100444400401041411224477242204420a0600053a000128e15840  
c4af4ac69aab9512db50f7a26ae5b4801183d85417e729a56da974f4e08a562c  
de6019dea9411332dc1a743372d170b436238a34597c410ea177024de20fc819  
3a000128e25840b1bc6f6fa56640ed522b2ee0f5b3cf7e5d48f60be8148f0dc0  
8440f0a4e1dca47c04119ed6a1be31b7707e5f9d001a659a051003e95e1b936f  
05c37ea793ad633a000128e350150444400401041411224477142204425840b7  
47d98be9cef8a91af0292a479a3fa499527018b97ac1188ddefb0fa6fcb9b3d1  
d4159240a8663c8803a2ef7cebd7f644fa3394cf1057d612e1b3977d9de92d
```

```
COSE_Sign1 structure:
```

```
d2 ;tag(18)  
84 ;array(4)  
43 ;byte_str(3)  
a10126  
a0 ;map(0)  
59 ;byte_str(307)  
a83a000124ff5820c3e3664dcc47711bf81734bc95f087d81dd841d73fc805fc  
9237c7b3dfa25c503a000124f76853494c4142535f313a00012500490614b457  
ffe0f77ce3a000128df58240000000100000000000000000000000000000020  
00010209ffffff00000025000000003a000128e05818000000001004444004  
01041411224477242204420a0600053a000128e15840c4af4ac69aab9512db50  
f7a26ae5b4801183d85417e729a56da974f4e08a562cde6019dea9411332dc1a  
743372d170b436238a34597c410ea177024de20fc8193a000128e25840b1bc6f  
6fa56640ed522b2ee0f5b3cf7e5d48f60be8148f0dc08440f0a4e1dca47c0411  
9ed6a1be31b7707e5f9d001a659a051003e95e1b936f05c37ea793ad633a0001  
28e35015044440040104141122447714220442  
58 ;byte_str(64)  
b747d98be9cef8a91af0292a479a3fa499527018b97ac1188ddefb0fa6fcb9b3  
d1d4159240a8663c8803a2ef7cebd7f644fa3394cf1057d612e1b3977d9de92d
```

```
-----  
Token claims:
```

```
a8 ;map(8)  
3a ;int(-75008)  
58 ;byte_str(32)  
c3e3664dcc47711bf81734bc95f087d81dd841d73fc805fc9237c7b3dfa25c50  
3a ;int(-75000)  
68 ;text_str(8)  
"SILABS_1"  
3a ;int(-75009)  
49 ;byte_str(9)  
0614b457ffe0f77ce  
3a ;int(-76000)  
58 ;byte_str(36)  
000000010000000000000000000000000000000000002000010209ffffff0000002500000000  
3a ;int(-76001)  
58 ;byte_str(24)  
00000000100444400401041411224477242204420a060005  
3a ;int(-76002)  
58 ;byte_str(64)  
c4af4ac69aab9512db50f7a26ae5b4801183d85417e729a56da974f4e08a562c  
de6019dea9411332dc1a743372d170b436238a34597c410ea177024de20fc819  
3a ;int(-76003)  
58 ;byte_str(64)
```

```
Exiting...
SL_STATUS_OK (cycles: 8 time: 0 us)
```

**Note:** The reserved tamper source in ID 76004 returns a value of 0 or 5.

## Simplicity Commander

Run the `security attestation` command to retrieve and validate the security configuration token ([Entity Attestation Token \(EAT\)](#)) from the HSE.

```
commander security attestation --serialno 440030580
```

Certificate chain successfully validated up to Silicon Labs device root certificate.

```
-75008 ARM PSA nonce      : 05a88aeef627dd663058e3d758fe9a827942da0793da72af81c79a4f60fa9824
-75000 ARM PSA Profile ID : SILABS_1
-75009 ARM PSA/IETF EAT UEID : 0614b457fffe0f77ce
-76000 SE status          : 000000001000000000000000000000003a9000000020000010209ffffff000000025000000000
-76001 OTP configuration  : 00000000100444400401041411224477242204420a060005
-76002 MCU sign key       : c4af4ac69aab9512db50f7a26ae5b4801183d85417e729a56da974f4e08a562c
                           de6019dea9411332dc1a743372d170b436238a34597c410ea177024de20fc819
-76003 MCU command key    : b1bc6f6fa56640ed522b2ee0f5b3cf7e5d48f60be8148f0dc08440f0a4e1dca4
                           7c04119ed6a1be31b7707e5f9d001a659a051003e95e1b936f05c37ea793ad63
-76004 Current applied tamper settings : 15044440040104141122447714220442
```

Successfully validated signature of attestation token.  
DONE

**Note:** The reserved tamper source in ID 76004 returns a value of 0 or 5.

## Secure Key Storage

# Secure Key Storage

**Note:** This section replaces *AN1271: Secure Key Storage*. Further updates to this application note will be provided here.

Secure Key Storage is a feature in High devices that allows for the protection of cryptographic keys by key wrapping. User keys are encrypted by the device's root key for non-volatile storage for later usage. This prevents the need for a key to be stored in plaintext format on the device, preventing attackers from gaining access to the keys through traditional flash-extraction or application attacks, and allowing for a potentially unlimited number of keys to be securely stored in any available storage.

Series 2 devices can use TrustZone to implement Secure Key Storage, so this feature is now also available on Mid devices.

This document describes the operation and usage of this feature, and provides comparisons with other key storage methods.

## Key Points

- Keys are encrypted or 'wrapped' with a root key
- root key is not stored on the device, instead it is generated on each reset
- Wrapped keys are confidential to the , and can be stored in non-volatile memory safely
- Wrapped keys can be cached into for usage at a later time
- TrustZone Secure Key Storage

Series 2 Device Security Features

# Series 2 Device Security Features

Protecting IoT devices against security threats is central to a quality product. Silicon Labs offers several security options to help developers build secure devices, secure application software, and secure paths of communication to manage those devices. Silicon Labs’ security offerings were significantly enhanced by the introduction of the Series 2 products that included a Secure Engine. The Secure Engine is a tamper-resistant component used to securely store sensitive data and keys and to execute cryptographic functions and secure services.

On Series 1 devices, the security features are implemented by the TRNG (if available) and CRYPTO peripherals.

On Series 2 devices, the security features are implemented by the Secure Engine and CRYPTOACC (if available). The Secure Engine may be hardware-based, or virtual (software-based). Throughout this document, the following abbreviations are used:

- HSE - Hardware Secure Engine
- VSE - Virtual Secure Engine
- SE - Secure Engine (either HSE or VSE)

Additional security features are provided by Secure Vault. Three levels of Secure Vault feature support are available, depending on the part and SE implementation, as reflected in the following table:

Level (1)	SE Support	Part (2)
Secure Vault High (SVH)	HSE only (HSE-SVH)	Refer to <a href="#">IoT Endpoint Security Fundamentals</a> for details on supporting devices.
Secure Vault Mid (SVM)	HSE (HSE-SVM)	"
"	VSE (VSE-SVM)	"
Secure Vault Base (SVB)	N/A	"

Notes:

1. The features of different Secure Vault levels can be found in <https://www.silabs.com/security>.
2. [IoT Endpoint Security Fundamentals](#).

Secure Vault Mid consists of two core security functions:

- Secure Boot: Process where the initial boot phase is executed from an immutable memory (such as ROM) and where code is authenticated before being authorized for execution.
- Secure Debug access control: The ability to lock access to the debug ports for operational security, and to securely unlock them when access is required by an authorized entity.

Secure Vault High offers additional security options:

- Secure Key Storage: Protects cryptographic keys by "wrapping" or encrypting the keys using a root key known only to the HSE-SVH.
- Anti-Tamper protection: A configurable module to protect the device against tamper attacks.
- Device authentication: Functionality that uses a secure device identity certificate along with digital signatures to verify the source or target of device communications.

A Secure Engine Manager and other tools allow users to configure and control their devices both in-house during testing and manufacturing, and after the device is in the field.

## User Assistance

In support of these products Silicon Labs offers whitepapers, webinars, and documentation. The following table summarizes the key security documents:

Document	Summary	Applicability
<a href="#">Series 2 Secure Debug</a>	How to lock and unlock Series 2 debug access, including background information about the SE	Secure Vault Mid and High
<a href="#">Series 2 Secure Boot with RTSL</a>	Describes the secure boot process on Series 2 devices using SE	Secure Vault Mid and High
AN1222: Production Programming of Series 2 Devices	How to program, provision, and configure security information using SE during device production	Secure Vault Mid and High
<a href="#">Anti-Tamper Protection Configuration and Use</a>	How to program, provision, and configure the anti-tamper module	Secure Vault High
<a href="#">Authenticating Silicon Labs Devices using Device Certificates</a>	How to authenticate a device using secure device certificates and signatures, at any time during the life of the product	Secure Vault High
Secure Key Storage (this document)	How to securely 'wrap' keys so they can be stored in non-volatile storage.	Secure Vault High

## Key Reference

Public/Private keypairs along with other keys are used throughout Silicon Labs security implementations. Because terminology can sometimes be confusing, the following table lists the key names, their applicability, and the documentation where they are used.

Key Name	Customer Programmed	Purpose
Public Sign key (Sign Key Public)	Yes	Secure Boot binary authentication and/or OTA upgrade payload authentication
Public Command key (Command Key Public)	Yes	Secure Debug Unlock or Disable Tamper command authentication
OTA Decryption key (GBL Decryption key) aka AES-128 Key	Yes	Decrypting GBL payloads used for firmware upgrades
Attestation key aka Private Device Key	No	Device authentication for secure identity

## SE Firmware

Silicon Labs strongly recommends installing the latest SE firmware on Series 2 devices to support the required security features. Refer to [AN1222](#) for the procedure to upgrade the SE firmware and [IoT Endpoint Security Fundamentals](#) for the latest SE Firmware shipped with Series 2 devices and modules.



## Introduction

# Introduction

The HSE isolates cryptographic functions and data from the host Cortex-M33 core. It is used to accelerate cryptographic operations as well as to provide a method to securely store keys. This application note will cover the Secure Key Storage feature of the HSE-SVH devices.

The HSE contains one-time programmable memory (OTP) key storage slots for three specific keys:

1. The Public Sign Key, used for Secure Boot and Secure Upgrades
2. The Public Command Key, used for Secure Debug unlock and tamper disable
3. The Symmetric OTA Decryption Key, used for Over-The-Air updates

These keys are one-time programmable, and, after programming, are persistent for the lifetime of the device.

HSE-SVH devices also contain four volatile storage slots for any other user keys. These slots are not persistent through a reset. In the case where a key needs persistent storage, the key must be stored outside of the HSE in non-volatile storage. After a device reset, the key can be loaded into the HSE volatile key storage for usage by index, or used in-place (passed to the HSE on every requested operation). Without any secure key storage mechanism, the user key stored in non-volatile storage is opened to storage-extraction attacks (such as gaining access to and downloading device flash), as well as application-level attacks (i.e., taking control of the user application or privileges in a manner that allows access to the keys).

With Secure Key Storage, a user can only access a key from the HSE in a 'wrapped' format. In this format, the key is encrypted by a device-unique root key, only available to the HSE. This allows a user to store a key confidentially in non-volatile storage to provide key persistence. Using Secure Key Storage, the plaintext key is never stored in non-volatile memory, preventing storage-extraction attacks from obtaining the key. After a device reset, the wrapped key can be loaded into the HSE for usage without ever exposing the plaintext key to the application, which also prevents application-level attacks from exposing the key.

SVM devices can only support Secure Key Storage through the use of [TrustZone](#). GSDK v4.2.2 is the first version to support TrustZone software development on Series 2 devices.

Silicon Labs provides [Custom Part Manufacturing Service \(CPMS\)](#) to inject custom secret keys on the chips during manufacturing. For more information about CPMS, see the [Custom Part Manufacturing Service User's Guide](#).

## HSE Secure Key Storage

# HSE Secure Key Storage

The following sections demonstrate three methods for key storage: ARM® TrustZone®, plaintext, and Secure Key Storage.

**Note:** In the following examples, AES key usage is demonstrated. However, any other key types supported by the device can also be used for key storage.

## Key Generation and Usage

In HSE-SVH devices, cryptographic functions are performed by the HSE. In order to perform these functions, the HSE must have access to any user keys needed. Keys can be generated and used by the HSE in multiple ways:

1. External storage, in-place usage:
  1. A user generates a plaintext key and stores it in device memory.
  2. The user provides a key descriptor to the HSE that points to this key for a specific cryptographic operation.
  3. The HSE performs the cryptographic operation using this key, but does not store it in any HSE volatile storage slot.
2. External storage with HSE import:
  1. A user generates a plaintext key and stores it in device memory.
  2. The user provides a key descriptor to the HSE that points to this key, as well as a slot number to store the key.
  3. The HSE imports this key into a volatile key storage slot or can optionally save it in wrapped form in device memory.
  4. The user requests that the HSE performs a cryptographic function by providing the index of the storage slot or a pointer to the wrapped key in device memory.
3. Internal HSE key generation:
  1. The user commands the HSE to generate a new key within one of the HSE's volatile key slots or can optionally save it in wrapped form in device memory.
  2. The user requests that the HSE performs a cryptographic function by providing the index of the storage slot or a pointer to the wrapped key in device memory.

### Notes:

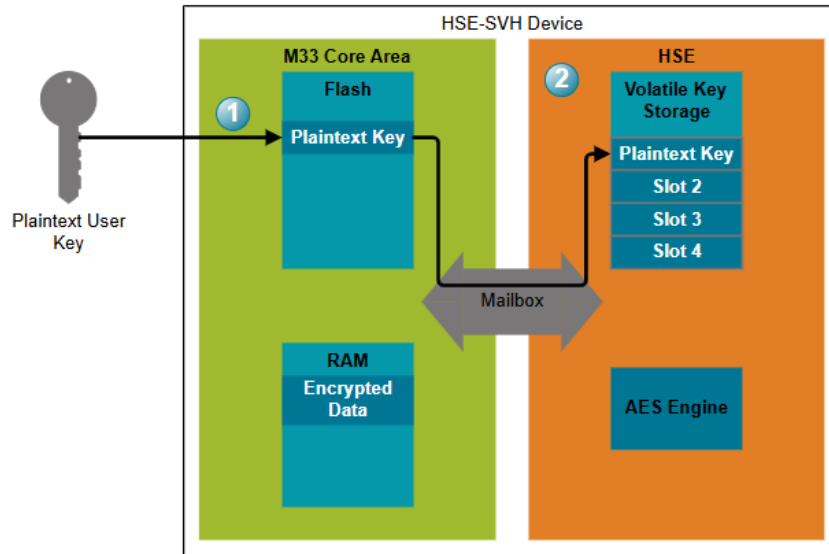
- In each case, to provide persistent storage for the key, the key must be stored in non-volatile memory.
- [Plain Key Storage](#) and [Secure Key Storage](#) provide details on key generation and usage with HSE-SVH device.

## Plaintext Key Storage

### Plaintext Key Import

The simplest manner to store a key is to save it in plaintext form. The steps to store and use a key stored in plaintext form are as follows:

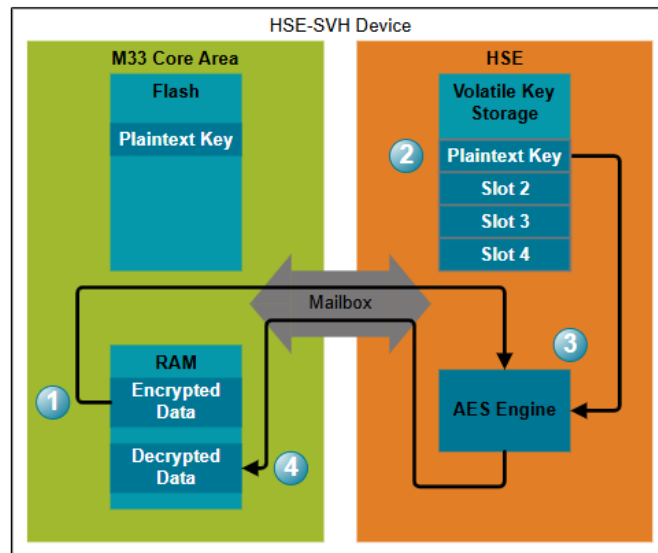
1. A user key is generated and imported into device memory. For persistent storage, this must be non-volatile storage, such as device flash.
2. After a device reset, the HSE volatile key storage will be empty. The plaintext key is imported ([method 2](#)) into a slot for usage. Alternatively, the key could be used in place ([method 1](#)) from non-volatile storage on a per-operation basis.



## Plaintext Key Usage

In order to use the key for a cryptographic operation, the following procedure is used.

1. The user passes data to be processed (in this specific example, AES encrypted data) to the HSE.
2. The user requests that a cryptographic operation be performed on this data using one of the keys stored in the HSE volatile key storage slots ([method 2](#)). Alternatively, the key can be passed to the HSE directly for a singular cryptographic operation ([method 1](#)).
3. The HSE performs the cryptographic operation.
4. The output of the cryptographic operation is passed back to the user for processing.



This method exposes the keys to two major vulnerabilities:

1. Access to device storage gives access to the keys. In this case, an attack that gains access to the flash contents will expose the user key.
2. Since the application has access to the keys, compromising the application or device privileges can compromise the keys. Such an attack might not directly access device memory, but take control of the application in a way that causes the application to expose the key to an attacker.

## Secure Key Storage

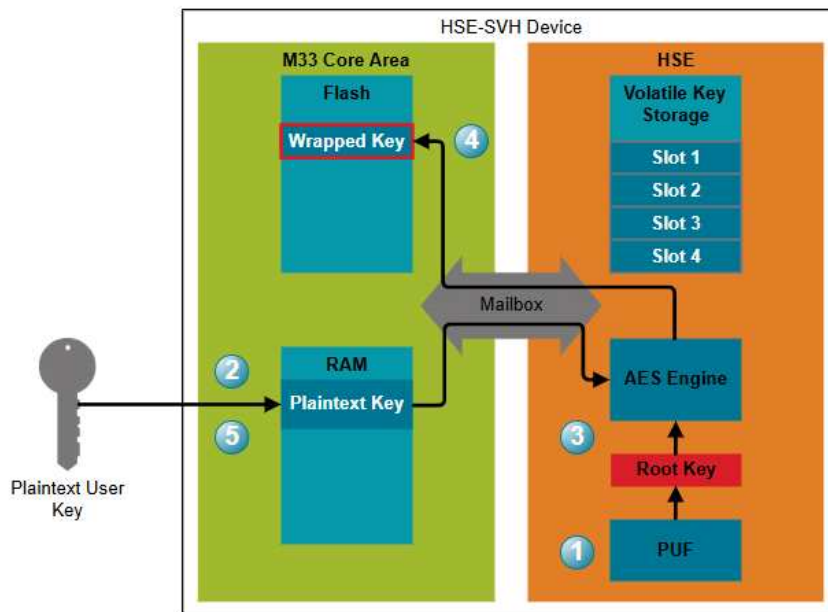
With Secure Key Storage, the user key, using the HSE, can be accessed in an encrypted, or 'wrapped' form. Only the HSE has access to the HSE root key used to decrypt, or 'unwrap', the wrapped key. This HSE root key is not stored on the device during power-down, but rather reconstructed after each reset. Key wrapping allows a user to securely store a key in non-volatile memory, limiting the number of keys that can be stored only by the amount of storage the user has available.

**Note:** The reconstructed root key after each reset is IDENTICAL and UNIQUE on each HSE-SVH device.

### Wrap an External Key

To wrap an externally-generated key:

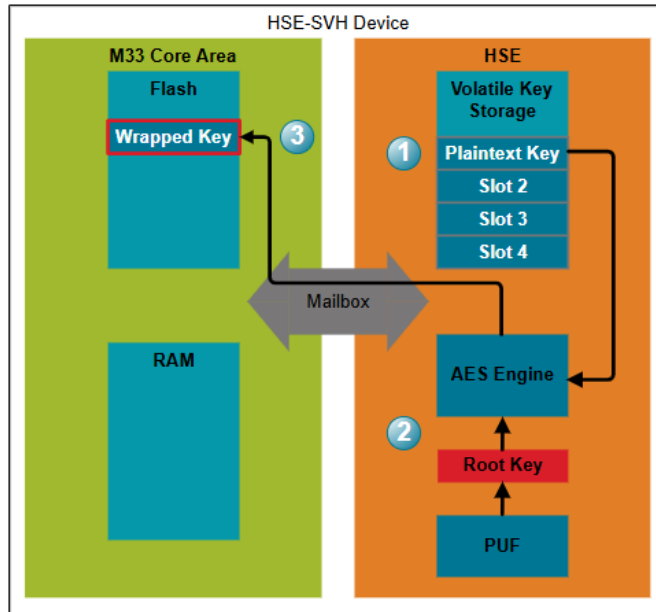
1. After power-on, the device's unique root key is reconstructed with output from the Physically Unclonable Function (PUF).
2. A user key is generated and imported into device memory. In this example, the key is imported into RAM for easy deletion, and the added security that, if device power is removed, the key will be lost.
3. The user key is passed to the HSE, where it is encrypted with the HSE's root key.
4. The wrapped key is passed back to the user application for storage in non-volatile memory (in this case, device flash).
5. The plaintext key can now be deleted from the device. From this point forward, only the HSE will have access to the plaintext key.



### Generate an Internal Wrapped Key

Instead of importing an external key, the HSE can generate a new key directly into one of its volatile key storage slots. This key can then be exported in wrapped form for secure persistent storage.

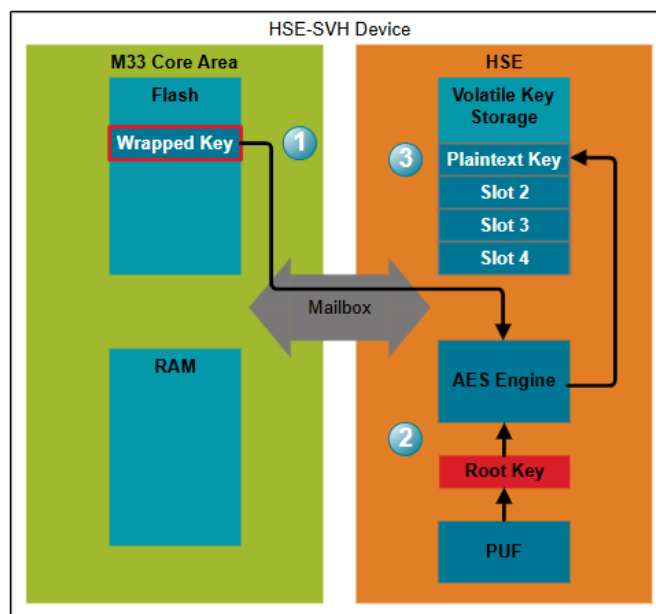
1. The user requests that the HSE generates a new key into one of its storage slots using the True Random Number Generator (TRNG).
2. The key is encrypted with the HSE's root key.
3. The wrapped key is passed back to the user application for non-volatile storage (flash, in this case).



## Wrapped Key Import

In order to import a wrapped key into the HSE for usage:

1. The wrapped key is passed to the HSE.
2. The wrapped key is decrypted ("unwrapped") with the HSE's root key.
3. The plaintext key is stored in a volatile key storage slot.

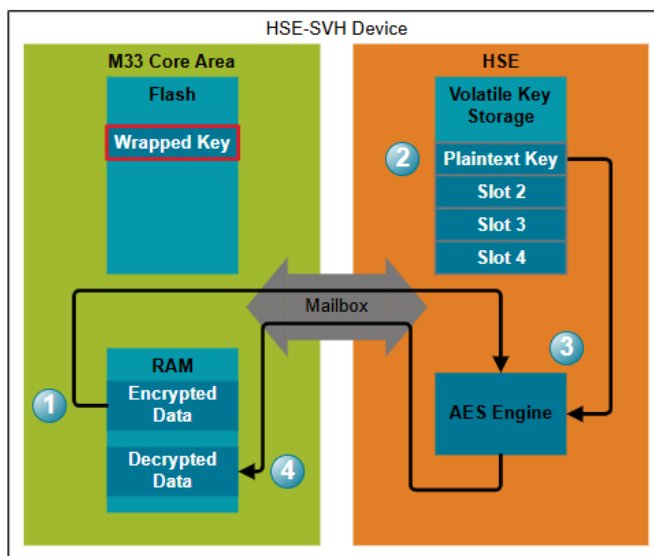


## Wrapped Key Usage

In order to use the key for a cryptographic operation, the same steps are followed as when using a plaintext key that has been imported into the HSE:

1. The user passes data to be processed (in this specific example, AES encrypted data) to the HSE.

2. The user requests that a cryptographic operation be performed on this data using one of the keys stored in the HSE volatile key storage slots. Alternatively, the wrapped key can be passed to the HSE directly for a singular cryptographic operation. In this case, the key will be unwrapped before being used, but will not be stored for future operations.
3. The HSE performs the cryptographic operation.
4. The output of the cryptographic operation is passed back to the user for processing.



## Secure Key Storage Advantages

Secure Key Storage confers the following benefits over other key storage methods:

1. Access to device memory does not expose user keys.
2. Compromising the user application does not expose user keys, since the user application itself does not have access to the plaintext keys.
3. The number of user keys that can be securely stored is only limited by the amount of storage available to the user, including external storage.

## Operation Details

### Root Key Generation

Secure Key Storage depends on the HSE to encrypt / decrypt (wrap / unwrap) user keys with its own symmetric root key. The symmetric key used for this wrapping and unwrapping must be highly secure as it can expose all other key material in the system. The HSE key Management system uses a Physically Unclonable Function (PUF) to generate a persistent device-unique seed on power up to dynamically reconstruct this critical root key. The key is only visible to the AES encryption engine, and it is not retained when the device loses power.

### Access a Wrapped Key

By default, a key in an HSE storage slot can be exported to the application as a plaintext key. To prevent this, the user can use the key descriptor to set a user key to [non-exportable](#). This option prevents any request to export the wrapped key in plaintext from HSE, so the user application can only access the key encrypted by the HSE's root key. The HSE also tags the key with information to identify the wrapped key. Since only the HSE can access the root key to unwrap the user key, the plaintext key is non-accessible to the user application.

**Note:** Wrapped keys are slightly larger than the equivalent plaintext key, as some additional metadata is required to identify the wrapped key to the HSE.

## Wrapped Key Storage and Usage

Once a key has been wrapped, it can be safely stored anywhere - device flash, RAM, external storage, etc. The number of keys that can be securely stored is only limited by the available storage space. A wrapped key can later be imported into a HSE volatile storage slot for usage, or used in-place. Once the key is wrapped and stored, the plaintext key available to the application can be deleted. From here, only the HSE will have the ability to unwrap and use the key.

With access to the wrapped key, the HSE can use this key in one of two ways:

1. A user can request that a cryptographic operation be performed using the key stored in memory. In this case, the HSE will import the key, unwrap it, and then perform the cryptographic operation. The key will not be stored within the HSE.
2. A user can import the wrapped key into a HSE volatile storage slot. In this case, the key is unwrapped by the HSE and stored in plaintext in a volatile slot. The user can then later request that a cryptographic function be performed by the HSE by referencing the volatile slot index. This provides a performance increase over using wrapped keys in place, as the HSE does not need to import and unwrap the key on each requested operation.

## Password Protection

When defining a key descriptor for a new key, or when importing an existing key into HSE, the user can choose to require a password to allow use of the key. The password field in the key descriptor structure is eight bytes in length. If unspecified, the key will use the default password of all zeros.

After importing a key with a password, failing to provide the correct password when performing a cryptographic operation will result in HSE returning an invalid credentials error, and no operation will be performed.

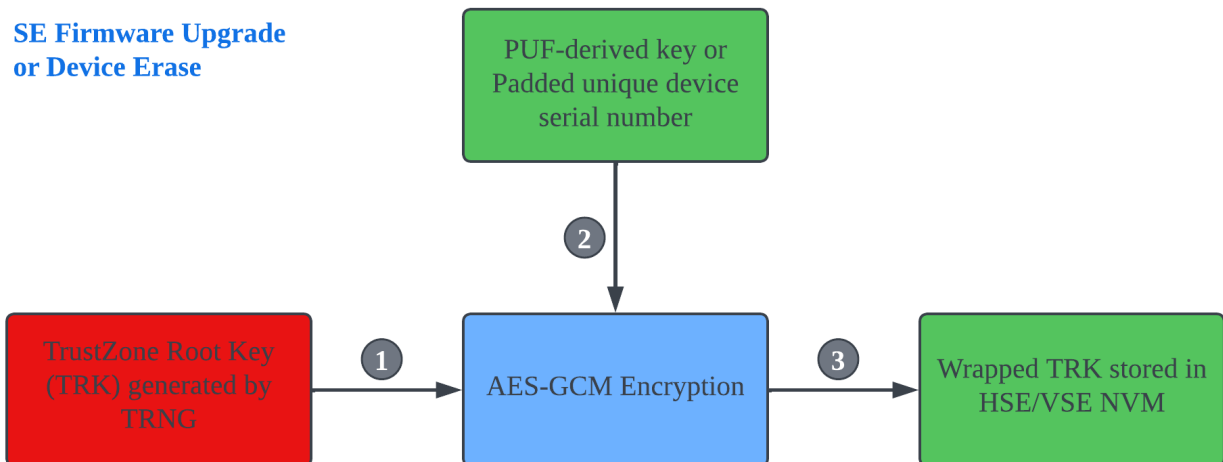
## TrustZone Secure Key Storage

# TrustZone Secure Key Storage

In Series 2 devices, key management can be handled by a feature called TrustZone. TrustZone divides the device memory map into a Secure Processing Environment (SPE) and a Non-secure Processing Environment (NSPE). User code is executed from the NSPE, which cannot access any part of the SPE. The SPE is used to store cryptographic keys securely and to control other Secure operations.

The following sections describe using TrustZone on Series 2 devices for Secure Key Storage. Refer to [Series 2 TrustZone](#) for details about TrustZone implementation on Series 2 devices.

## TrustZone Root Key Generation (HSE and VSE)

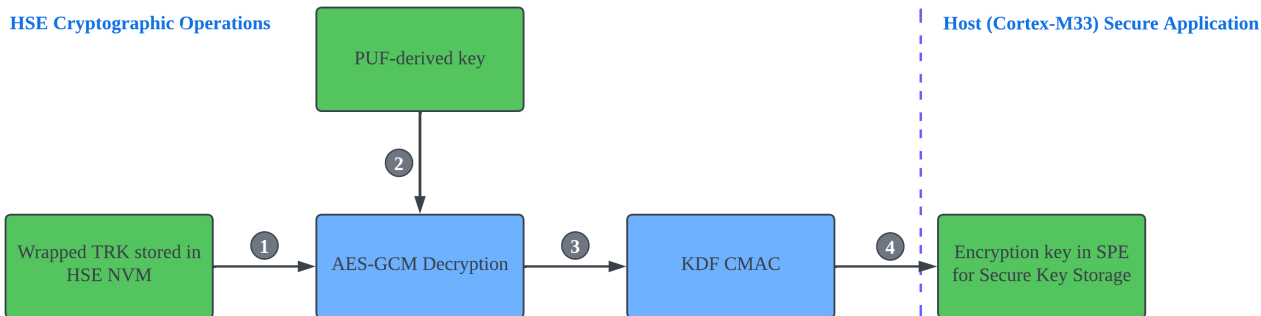


1. The TrustZone Root Key (TRK) is generated by the True Random Number Generator (TRNG) in Series 2 devices.
2. The PUF-derived key (HSE and xG27 VSE devices) or padded unique device serial number (xG22 VSE devices) is used to wrap (AES-GCM) the TRK.
3. The wrapped TRK is stored in the SE Non-volatile memory (NVM), and the TRK in RAM is deleted.
  - The wrapped TRK already existed if the shipped Series 2 device with SE firmware version supports this key.
  - The wrapped TRK will be generated when upgrading from a SE firmware version that did not support this key to the one that does.
  - The wrapped TRK will be renewed after performing a Device Erase.

**Note:** The Physically Unclonable Function (PUF) is not retained when the device loses power, so the TRK wrapped by the PUF-derived key is not vulnerable to a storage-extraction attack.

## TrustZone Root Key Usage (HSE)



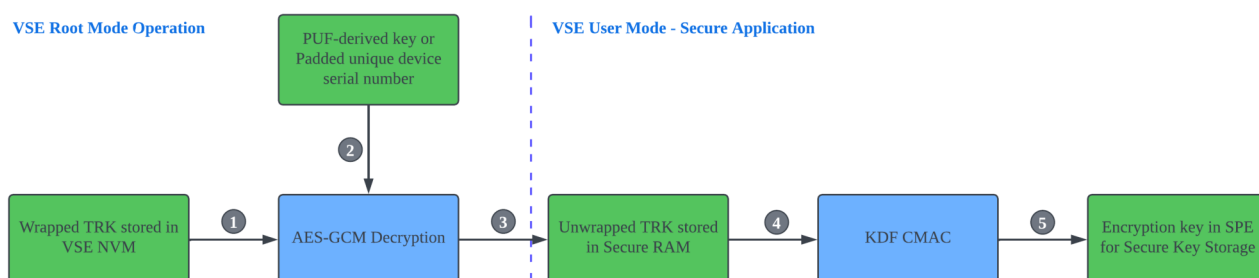


1. The Secure application in the host uses a non-exportable built-in key to access the wrapped TRK in HSE NVM for cryptographic operations.
2. The PUF-derived key is used to decrypt (AES-GCM) the wrapped TRK in HSE NVM.
3. The unwrapped TRK in the HSE is the master key of a Key Derivation Function (KDF).
4. The encryption key in SPE for Secure Key Storage is derived from the KDF CMAC.

**Notes:**

- All cryptographic operations are performed by the HSE (security co-processor).
- Only the HSE can access the unwrapped TRK for KDF, so this key will not expose the Secure application in the host.

## TrustZone Root Key Usage (VSE)



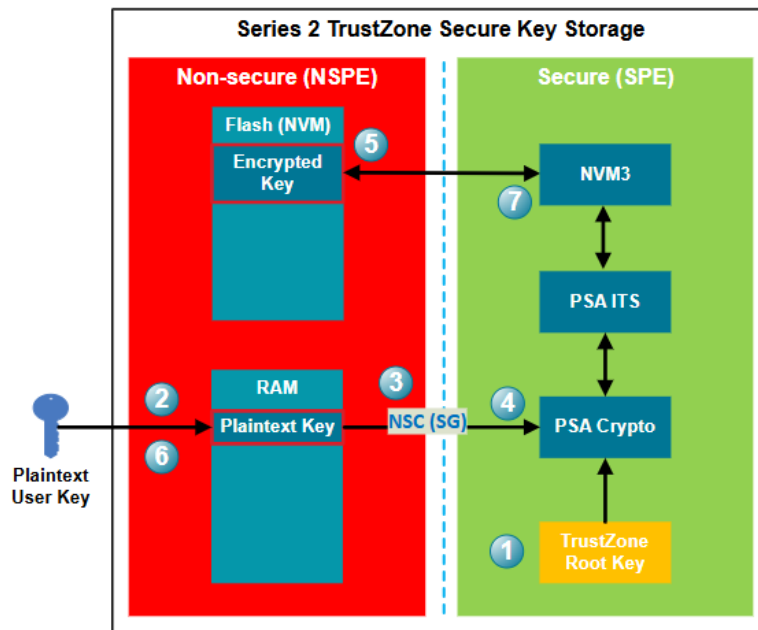
1. The wrapped TRK in VSE NVM is accessed by the VSE Root mode firmware.
2. The PUF-derived key (xG27) or padded unique device serial number (xG22) is used to decrypt (AES-GCM) the wrapped TRK in VSE NVM.
3. Unwrapped TRK is transferred to the shared RAM when switching from VSE Root mode to User mode. The VSE user mode Secure application stores this key to the Secure RAM in SPE and deletes this key in the shared RAM.
4. The unwrapped TRK in the Secure RAM is the master key of a Key Derivation Function (KDF).
5. The encryption key in SPE for Secure Key Storage is derived from the KDF CMAC.

**Note:** On VSE devices, all cryptographic operations are performed by the Cryptographic Accelerator (CRYPTOACC) peripheral.

For more information about the HSE and VSE, refer to the section *Secure Engine Subsystem* in [Series 2 Secure Debug](#).

## TrustZone Secure Key Storage (HSE and VSE)

The TRK allows a user to securely store a key in the Non-secure flash, limiting the number of keys that can be saved only by the amount of Non-secure storage. The following figure describes using the TRK to encrypt a plaintext key and store it



in Non-secure NVM.

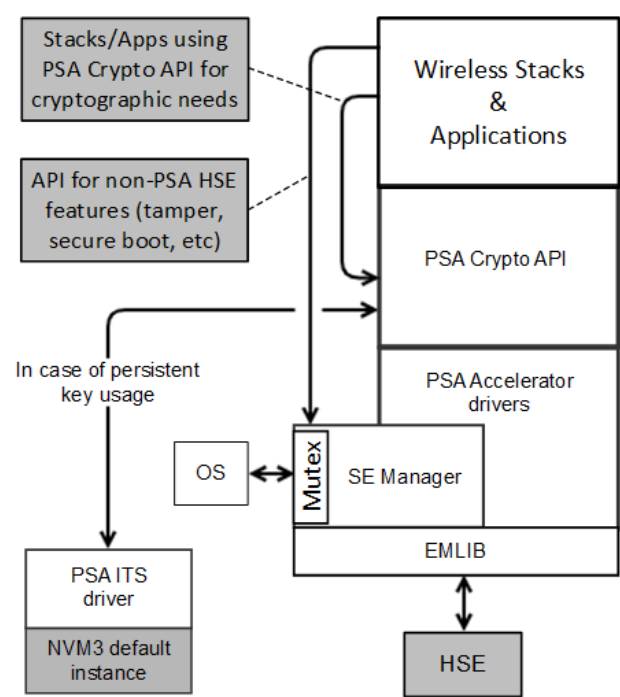
1. After power-on, the device's TRK (wrapped in HSE NVM and unwrapped in VSE Secure RAM) is available for the SPE.
2. A user key is generated and imported into the device's Non-secure memory. In this example, the key is imported into Non-secure RAM for easy deletion, and the key is lost if device power is removed.
3. Call the PSA Crypto API ( `psa_import_key()` or `psa_generate_key()` ) through the Secure Gateway (SG) in Non-secure Callable (NSC) memory to generate a key for crypto operations.
4. The plaintext key is passed in the PSA Crypto API to the SPE, where it is encrypted (AES-GCM) with the encryption key derived (KDF CMAC) from the TRK.
5. The encrypted key is stored to the NVM region in the NSPE through the PSA Internal Trusted Storage (ITS) and [NVM3](#) drivers.
6. The plaintext key can now be deleted from the Non-secure RAM.
7. Only the PSA Crypto API in the SPE can retrieve the encrypted key from NVM in the NSPE and decrypt it for crypto operations in the SPE.

**Note:** Ignore steps 2 and 6 if the plaintext key is randomly generated by the PSA Crypto.

# Secure Key Storage Implementations

Users can use Secure Engine Manager (SE Manager) or PSA Crypto in the following figure to access the secure key storage on HSE-SVH devices. SE Manager APIs for secure key storage and crypto are usually not considered external APIs. PSA Crypto API abstracts the entropy sources, crypto primitives, and even advanced security features like secure key storage from the calling functions.

Silicon Labs recommends using PSA Crypto API for secure key storage and cryptography whenever possible. It makes the solution more portable and hardware agnostic. In some cases, however, setting up tamper and initializing the secure boot can only be implemented by the SE Manager APIs.



Component	Functionality
EMLIB (em_se.c)	Abstracts the mailbox interface: how to construct, send and receive low-level HSE mailbox commands.
SE Manager	On top of EMLIB, it abstracts the HSE command set: translates function calls into mailbox messages. The SE Manager also provides thread synchronization.
PSA Accelerator Drivers	A translation layer to map the PSA Crypto HSE interface and crypto acceleration calls to SE Manager calls.
PSA Crypto API	Platform independent cryptographic hardware acceleration support by implementing standardized APIs.
PSA ITS Driver	The key management functionality in PSA Crypto needs access to non-volatile memory for persistent storage of plaintext or wrapped keys. NVM3 gets wrapped by this translation layer, mapping the PSA ITS (Internal Trusted Storage) interface to NVM3 calls.

For the SE's mailbox interface, see section *Secure Engine Subsystem* in [Series 2 Secure Debug](#).  
For more information about NVM3, see <https://docs.silabs.com/gecko-platform/latest/driver/api/group-nvm3>.

For more information about PSA Crypto, see [AN1311: Integrating Crypto Functionality Using PSA Crypto Compared to Mbed TLS](#).

### SE Manager API

The following table lists the SE Manager APIs related to Secure Key Storage operations. The SE Manager API document can be found at <https://docs.silabs.com/gecko-platform/latest/service/api/group-sl-se-manager>.

SE Manager API	Usage
sl_se_generate_key	Generate a new key and store it either in a volatile HSE storage slot or as a wrapped key.
sl_se_import_key	Import a plaintext key and store it either in a volatile HSE storage slot or as a wrapped key.
sl_se_export_key	Export a volatile or wrapped key back to plaintext if allowed. It will fail for a key that has been flagged as SL_SE_KEY_FLAG_NON_EXPORTABLE.
sl_se_transfer_key	Transfer a volatile or wrapped key to another storage option (volatile HSE storage slot or a wrapped key) if allowed.
sl_se_delete_key	Delete a key from a volatile HSE storage slot.

### PSA Crypto API

The following table lists the PSA Crypto APIs related to Secure Key Storage operations. The PSA Crypto API document can be found at <https://docs.silabs.com/mbed-tls/latest/>.

For more information about PSA Crypto APIs on Secure Key Storage, see [AN1311: Integrating Crypto Functionality Using PSA Crypto Compared to Mbed TLS](#).

PSA Crypto API	Usage
psa_generate_key	Generate a new plaintext or wrapped key and store it either in volatile or non-volatile memory.
psa_import_key	Import a plaintext key and save it in plaintext or wrapped form. It can store either in volatile or non-volatile memory.
psa_export_key	Export a key back to plaintext if allowed. The policy on the key must have the usage flag PSA_KEY_USAGE_EXPORT set.
psa_copy_key	Copy key material from one location to another, which may have a different lifetime (e.g., volatile to non-volatile).
psa_destroy_key	Destroy a key from both volatile memory and, if applicable, non-volatile storage.

### SE Manager API Versus PSA Crypto API

The following table compares the SE Manager APIs with PSA Crypto APIs on Secure Key Storage.

Item	SE Manager API	PSA Crypto API
Availability	Only on HSE devices	Platform independent
API	Silicon Labs proprietary	Standardized by ARM®
Key Storage	Volatile (RAM) memory only	Volatile (RAM) or non-volatile (flash) memory
Wrapped Key Cache	Can use a volatile HSE storage slot	Not yet implemented
Password Protection	Can define in a key descriptor	Not yet defined in PSA Crypto
Custom ECC Curve	Can define in a key descriptor	Not yet defined in PSA Crypto

### PSA Crypto Key Types with TrustZone Secure Key Storage

The following tables describes the storage differences between key storage with and without TrustZone on SVM and SVH devices.

Table: TrustZone Secure Key Storage (SKS) on SVM Devices

Key Type	Storage without TrustZone SKS	Storage with TrustZone SKS
Volatile Plaintext	RAM	Secure RAM (2)
Persistent Plaintext	NVM	Encrypted in NS NVM (2)
Volatile Wrapped	Not supported	Not supported
Persistent Wrapped	Not supported	Not supported

Table: TrustZone Secure Key Storage (SKS) on SVH Devices

Key Type	Storage without TrustZone SKS	Storage with TrustZone SKS
Volatile Plaintext	Plaintext key in RAM	Plaintext key in Secure RAM
Persistent Plaintext	Plaintext key in NVM	Encrypted plaintext key in NS NVM
Volatile Wrapped	Wrapped key in RAM (1)	Wrapped key in Secure RAM
Persistent Wrapped	Wrapped key in NVM (1)	Encrypted wrapped key in NS NVM

Notes:

- The NVM or NS NVM is at the last part of the main flash.
- It is possible to replace the wrapped key solution on the SVH device (1) with TrustZone Secure Key Storage on the SVM device (2), but this is a less secure approach.

Examples

# Examples

Simplicity Studio 5 includes the [SE Manager and PSA Crypto platform examples](#) for Secure Key Storage. Refer to the corresponding `readme` file for details about each SE Manager and PSA Crypto platform example. This file also includes the procedures to create the project and run the example.

Table: Platform Examples for Secure Key Storage

Category	SE Manager Platform Example	PSA Crypto Platform Example
Key Handling	SE Manager Symmetric Key Handling	PSA Crypto Symmetric Key
	SE Manager Asymmetric Key Handling	PSA Crypto Asymmetric Key
Symmetric Key Usage	SE Manager Block Cipher	PSA Crypto AEAD
		PSA Crypto Cipher
		PSA Crypto KDF
		PSA Crypto MAC
Asymmetric Key Usage	SE Manager Digital Signature (ECDSA and EdDSA)	PSA Crypto DSA
	SE Manager Key Agreement (ECDH)	PSA Crypto ECDH
X.509 Certificate	-	PSA Crypto X.509
TrustZone Secure Key Storage	-	tz_psa_crypto_ecdh_ws

## Protocol-Specific Information

# Protocol-Specific Security References

The pages in this section offer protocol-specific information. For general content applicable to any protocol that supports the feature, see the [main development section](#).

## Bluetooth

[Bluetooth Low Energy Application Security Design Considerations in SDK v3.x and Higher \(PDF\)](#): Provides details on designing Bluetooth Low Energy applications with security and privacy in mind.

[Certificate-Based Bluetooth Authentication and Pairing \(PDF\)](#): Describes the theoretical background of certificate-based authentication and pairing, and demonstrates the usage of the related sample applications that can be found in Silicon Labs' Bluetooth SDK.

## Bluetooth Mesh

[Bluetooth Mesh Certificate-Based Provisioning \(PDF\)](#): Describes how certificates are used to establish the authenticity of devices wishing to join a mesh network.

## OpenThread

[Using Silicon Labs Secure Vault Features with OpenThread \(PDF\)](#): Describes how Secure Vault features are leveraged in OpenThread applications. Focuses on specific PSA features and emphasizes how these are integrated into the OpenThread stack.

## Zigbee

[Zigbee Security](#): Introduces some basic security concepts, including network layer security, trust centers, and application support layer security features. It then discusses the types of standard security protocols available in EmberZNet PRO. Coding requirements for implementing security are reviewed in summary. Finally, information on implementing Zigbee Smart Energy security is provided.

## Overview

# Silicon Labs IoT Security Production Guide

Securing an IoT device is a highly complicated and costly process. You must generate public and private keys for secure boot and secure debug, sign code with a private key, store all the private keys in an HSM, place the public keys for secure boot and secure debug in one-time-programmable (OTP) memory, flip OTP bits for secure boot and secure debug, and flash the encrypted code and identity certificates within the hardware.

CPMS streamlines the programming part of this process for you. Even the most advanced security features, certificates, and identities can be programmed in a secure, fast, and cost-efficient way at the Silicon Labs factories. This section provides details on CPMS, in addition to Public Key Infrastructure (PKI) Recommendations.

- [Custom Part Manufacturing Service](#): Explains the process for ordering custom Series 2 parts through the CPMS, including details on security settings and use cases for configuring a device for an untrusted manufacturing environment and importing custom wrapped keys.
- [PKI Recommendations](#): Outlines the recommended establishment, management, and security of PKI for business partners and customers of Silicon Labs.



## Custom Part Manufacturing Service

# Custom Part Manufacturing Service

This section explains the process for ordering custom Series 2 parts through the Custom Part Manufacturing Service (CPMS). Instructions for customizing device identity security certificates and wrapping custom keys are also included. For more information on Silicon Labs' security offerings on Series 2 devices, refer to [IoT Endpoint Security Fundamentals](#).

## What is CPMS?

Custom Part Manufacturing Service (CPMS) allows you to customize Silicon Labs hardware – wireless SoCs, modules, MCUs – at the factory. The CPMS self-service web portal guides you through the customization process and its various customizable features and settings. You can place orders for customized test and production units to our factories securely via the CPMS portal.

Unlike traditional flash programming, CPMS is a secure provisioning service that enables you to customize your chips with several highly advanced features such as secure boot, secure debug, encrypted OTA, public, private and secret keys, secure identity certificates, and more.

The custom features, identities, and certificates are injected on the hardware securely, quickly, and cost-efficiently at the world's safest place, the Silicon Labs factories.

## Why Choose CPMS?

Securing an IoT device is a highly complicated and costly process. You must:

- Generate public and private keys for secure boot and secure debug
- Sign code with a private key
- Store all the private keys in a Hardware Security Module (HSM)
- Place the public keys for secure boot and secure debug in one-time-programmable (OTP) memory
- Flip OTP bits for secure boot and secure debug
- Flash the encrypted code and identity certificates within the hardware

CPMS streamlines the programming part of this process for you. Even the most advanced security features, certificates, and identities can be programmed in a secure, fast, and cost-efficient way at the Silicon Labs factories.

## SE Firmware Version

# SE Firmware Version

Selecting the latest SE version available is recommended to stay up to date on bug fixes and security patches. It is also recommended to continue updating the SE Firmware as new versions are released. Instructions for implementing field upgrades of the SE Firmware can be found in [AN1222: Production Programming of Series 2 Devices](#).

## Debug Lock Settings

# Debug Lock Settings


Four debug lock settings are available for selection in CPMS for this required field. These settings are standard debug lock, secure debug lock, permanent lock, and unlocked. A public Command Key will be required if the Secure Debug setting is selected. CPMS will provision the public command key to the device, if required, and enable any debug settings specified here. More information on each debug lock setting can be found in [Series 2 Secure Debug](#).

## Secure Boot with RTSL Settings

# Secure Boot with RTSL Settings

Secure Boot with RTSL is a security setting available to Silicon Labs' Series 2 devices that is used to validate the integrity and authenticity of each piece of firmware before the firmware is allowed to run on these devices. Setting up a full root of trust includes enabling secure boot settings in the device's OTP as well as in the user-generated bootloader. The setting stored in device OTP will enforce the first stage bootloader in the Secure Engine to perform a signature check on the second stage bootloader, which is generated and signed by a user. The next link in the chain is established by enabling secure boot in the second stage bootloader, which will enforce a check on the application image signature.

In CPMS, you can configure the OTP settings for secure boot quickly and easily. These OTP settings are irreversible, so it is recommended to read about each setting in detail before making selections. The OTP settings for Secure Boot with RTSL are the enable bit, Certificates required bit, Anti-Rollback enable bit, and flash page locking settings. Each of these settings is covered in detail in [Series 2 Secure Boot with RTSL](#). Once these settings are selected, CPMS will prompt you to provide a secure boot key, also known as a public sign key. This key will be used to sign the firmware to be verified during the secure boot process. For more information on the public sign key, see [Secure Boot Key](#).


**Configure Secure Boot and Tamper Response Settings**

These configurations can only be made at one time and are **irreversible** once they are made. Read more about [secure boot with RTSL](#) and [production programming](#)

- ☒ **Enable Secure Boot with RTSL**  
 If set, authenticates the first code image in flash memory, which is typically the second stage bootloader, before allowing that code to run. Enabling secure boot will ensure that the device will only boot code that has been properly signed by you.
- ☐ **Require Verify Certificate before secure boot**  
 The Verify intermediate certificate before secure boot option provisions the Public Sign Key to enable certificate-based Secure Boot. Enabling this reduces the need to access the OTP signing key allowing more stringent access restrictions. It also provides the ability to roll the intermediate key in the event it is compromised.
- ☐ **Enable Anti Rollback**  
 We recommend enabling anti-rollback. If set, the first stage bootloader will compare the version of the first image in flash memory, which is typically the second stage bootloader, to the version of the image that has been staged for upgrade. If the staged image has a version that is **greater than** the current image, the upgrade will succeed. Otherwise the upgrade operation will be ignored.
- ☐ **Enable Flash Page Locking**  
 This feature write/erase locks flash pages starting at 0 that have been validated by the first stage bootloader signature check. This will prevent flash modification of the locked pages by any means other than through the hardware secure engine (write/erase attempts from the CPU or from the debug port will be ignored).  
**"Full"** - locks from page 0 up to and including the page containing the signature. This may lock flash bytes that are

**TASKS**  
 Please correct the following tasks.

☐ Please flash at least a bootloader (because Debug Lock is not Unlocked)

☐ Provide a public command key (because Secure Debug Lock is selected)

☐ Provide a public sign key (because secure boot is enabled)

Complete all **tasks** before submitting order.

Review Order

As mentioned previously, to establish the full root of trust for secure boot, secure boot must also be enabled in the bootloader that is uploaded as part of the firmware programming to CPMS. If the OTP setting for secure boot with RTSL was enabled, and the secure boot setting was not enabled in the bootloader, no signature check would be performed on the application image; only a signature check would be performed on the bootloader image. It is recommended to establish a full root of trust to narrow the attack surface of the device. Refer to [Generating the Bootloader](#) for instructions on creating a bootloader project with secure boot enabled.

## Enabling Secure Boot with RTSL on a VSE Device

A few extra steps are required to establish a full root of trust when enabling secure boot on a Virtual SE (VSE) device in CPMS. As outlined previously, the secure boot settings in OTP will need to be enabled and a public signing key should be uploaded to CPMS as the first steps to enable secure boot. This public signing key will be provisioned in the device's OTP memory by CPMS and will be used by the device to verify the signature on the second stage bootloader. For the second stage bootloader to verify the signature on the application on a VSE part, the bootloader will use the public signing key

stored in the top page of main flash. Refer to [Series 2 Secure Boot with RTSL](#) for more information on enabling secure boot with RTSL on a VSE device.

Follow the steps outlined in this section to place the public signing key in flash using a token file.

1. When creating a public and private signing key pair for secure boot, use the `--tokenfile` flag to write the public signing key to the token file. Refer to [Section 6.18.3 of UG162](#) for more information on this command.

```
commander util genkey --type ecc-p256 --privkey  
sign_privkey.pem --pubkey sign_pubkey.pem --tokenfile  
sign_pubkey.txt
```

2. Flash the token file to the device by running the `flash` command. Refer to [Section 6.1.6 of UG162](#) for more information on this command.

```
commander flash --tokenfile sign_pubkey.txt
```

3. Once the token file is flashed to the device, the signed bootloader and application images should be flashed to the device using the same command listed in step 2. To generate and sign a bootloader and application image, follow the steps listed in [Section 3.1.2](#) and [3.1.3](#).
4. Dump the flash contents (which contains the token file, signed bootloader, and signed application firmware) to a hex file using the `readmem` command. Refer to [section 6.3.2 of UG162](#) for more information.

```
commander readmem --region @mainflash --outfile all.hex
```

5. Upload the file **all.hex** to CPMS using the **App and Bootloader** selection in the Flash Programming section. As mentioned, this hex image should contain the token file, signed bootloader, and signed application. This will establish a full root of trust on the VSE device, ensuring that only authentic firmware can run on the device.

## Flash Programming

Flash Programming involves the addition of customer specific code to a standard product. Customer code in **INTEL HEX** format is required.

### Firmware

#### Fill Character

0x FF



We will fill unused or unspecified addresses of the flash with the byte you provide here.

#### Upload new Intel HEX files

##### Select Firmware Type

☐ App only ☐ Bootloader only ☒ App and Bootloader

##### Standard File Input

 CLICK HERE OR DRAG & DROP TO UPLOAD A FILE



## Tamper Response

# Tamper Response

In CPMS, the default tamper configuration will be automatically displayed for HSE SVH (Secure Vault High) parts, customization is available for each of these configurations for Secure Vault High parts only. The anti-tamper protection feature is only available on SVH devices. Each tamper configuration can be set to one of five different tamper response levels, ranging from ignore the tamper event to erase OTP of the affected device. CPMS will require a public command key to be uploaded when tamper is configured on a HSE-SVH device. For more information on anti-tamper protection, refer to [Anti-Tamper Protection Configuration and Use](#).

## Standard Security Keys

# Standard Security Keys

The following subsections will address the different security keys that are accepted by CPMS. Note that only public keys should be uploaded to CPMS. Silicon Labs strongly recommends that each key be generated using robust methods, that private keys are not shared with unauthorized parties, and that keys be stored in a well-managed and protected hardware security module (HSM).

## Secure Boot Key

The secure boot key, also known as the public signing key, is used for authenticating the signature on a bootloader or application image. CPMS will accept this key in .pem or .der format. This key should be generated as a public/private key pair, and only the public key should be provided to CPMS. For more information on this key pair, refer to [Series 2 Secure Boot with RTSL](#).

## Command Key

The command key is used for disabling tamper responses and performing a secure debug unlock. CPMS will accept this key in .pem or .der format. This key should be generated as a public/private key pair and only the public key should be provided to CPMS. For more information on this key pair and how to use it, refer to [Series 2 Secure Debug](#) and [Anti-Tamper Protection Configuration and Use](#).

## OTA Decryption Key

The OTA Decryption Key, also known as the GBL Decryption Key, is used for decrypting GBL payloads used for firmware upgrades. This key will only be required if you enable “require encrypted firmware upgrade files” in the bootloader. Refer to [Silicon Labs Gecko Bootloader User's Guide](#) for more information. An example of creating a bootloader with encrypted upgrades required is shown in [Generating the Bootloader](#).

On an HSE device, a 16-byte decryption key can be provided to CPMS to be provisioned to the device. On a VSE part, this key can only be provided to CPMS in a token file, like the public sign key used for secure boot on a VSE part.

## OTA Decryption Key for VSE Devices

A few additional steps are required to setup a VSE device to use an OTA Decryption Key in CPMS. Refer to [Section 7.2 of AN1222](#) for more information.

1. Generate the key using the `util genkey` command.

```
commander util genkey --type aes-ccm --outfile aes_key.txt
```

2. Once the key is generated, it needs to be written to a place accessible to the bootloader. This key can be placed in either the app properties struct of the GBL, or in the top page of main flash. Only one of these methods need to be used. To write the OTA Decryption key into the Application properties struct of the bootloader project, use the following command.

```
commander convert bootloader.hex --aeskey aes_key.txt --outfile bootloader-keys.hex
```

Note when using this method:

- The `--aeskey` option for the convert command requires Simplicity Commander v1.12.3 or above.
- The GBL Decryption Key can only be added to the GBL with Application Properties Struct v1.2 or higher (GSDK v4.1.0 or higher).
- This procedure must be implemented before signing the GBL image for Secure Boot.

To write the OTA Decryption Key to the top page of flash on a VSE device, use the following command.

```
commander flash --tokenfile aes_key.txt
```



Additional Custom Keys

# Additional Custom Keys

## Key Wrapping

Secure Vault High devices support Key Wrapping, which is a feature where keys are encrypted using a Physically Unclonable Function (PUF) key. A PUF key is secret, random, and unique to each individual device. PUF keys do not live in flash and are not vulnerable to flash extraction attacks.

CPMS allows customers to provide their own keys, which will be wrapped by the secure element and stored on the device. This means that the firmware image does not need to contain the key at any point in production.

To use this feature, you need to provide CPMS with four fields:

- 1. **Key Auth:** An 8-byte password that must be provided by software whenever the key is used. This password can be disabled by setting the Key Auth to 0x0000000000000000.
- 2. **Key Value:** The value of the key to be wrapped (max 200 bytes).
- 3. **Key Metadata:** 4 bytes of key metadata, including information such as the type of key, allowed uses, length, etc. More information on how to generate this value for an existing key can be found in [Importing Custom Wrapped Keys](#).
- 4. **Key Address:** The address in user flash to which the key should be programmed.

Additional Custom Keys

User Key 1 

Key Auth

0x

Auth data for key (must be 8 bytes)

Key Value

0x

Value of the key to be wrapped (max 200 bytes)

Key Metadata

0x

4 bytes of metadata

Key Address

0x

Address in user flash to which the key should be programmed

 ADD CUSTOM KEY

Custom Certificates

# Custom Certificates

CPMS allows you to customize the device identity certificate chain. The certificates use the X.509 format and must conform to [RFC-3280](#). For an example of a Silicon Labs device certificate, refer to [Authenticating Silicon Labs Devices using Device Certificates](#). Currently, CPMS supports customization of four fields in the device certificate:

- 1. **Common name:** User-defined, 30-character name that will terminate with the 64-bit EUI of the device (example is "EUI:xxxxxxxxxxxxxxxx" and will terminate with " S:SE0 ID:MCU" or " S:FL0 ID:MCU" depending on if the device is a Secure Vault High device or not.)
- 2. **Organization:** User-defined, 64-character company name
- 3. **Country:** Must be a legitimate country code letter pair (e.g., US)
- 4. **Organizational Unit:** User-defined field of up to 64 characters

If there are other certificate customizations you would like to implement, specify them in the **Special Instructions** section in the CPMS.

☒ Custom Identity

Custom Identity allows customers to extend the default Silicon Labs certificate identity cert chain to provide your own. This is an advanced feature which requires additional charges. Please contact a Silicon Labs sales representative for details.  
[Read more about secure identity](#)

Scope of Customization  
☒ Device certificate only    ☐ The certificate chain

Special Instructions

## Configure Device for Untrusted Environment Example

# Configuring a Device for an Untrusted Manufacturing Environment Example

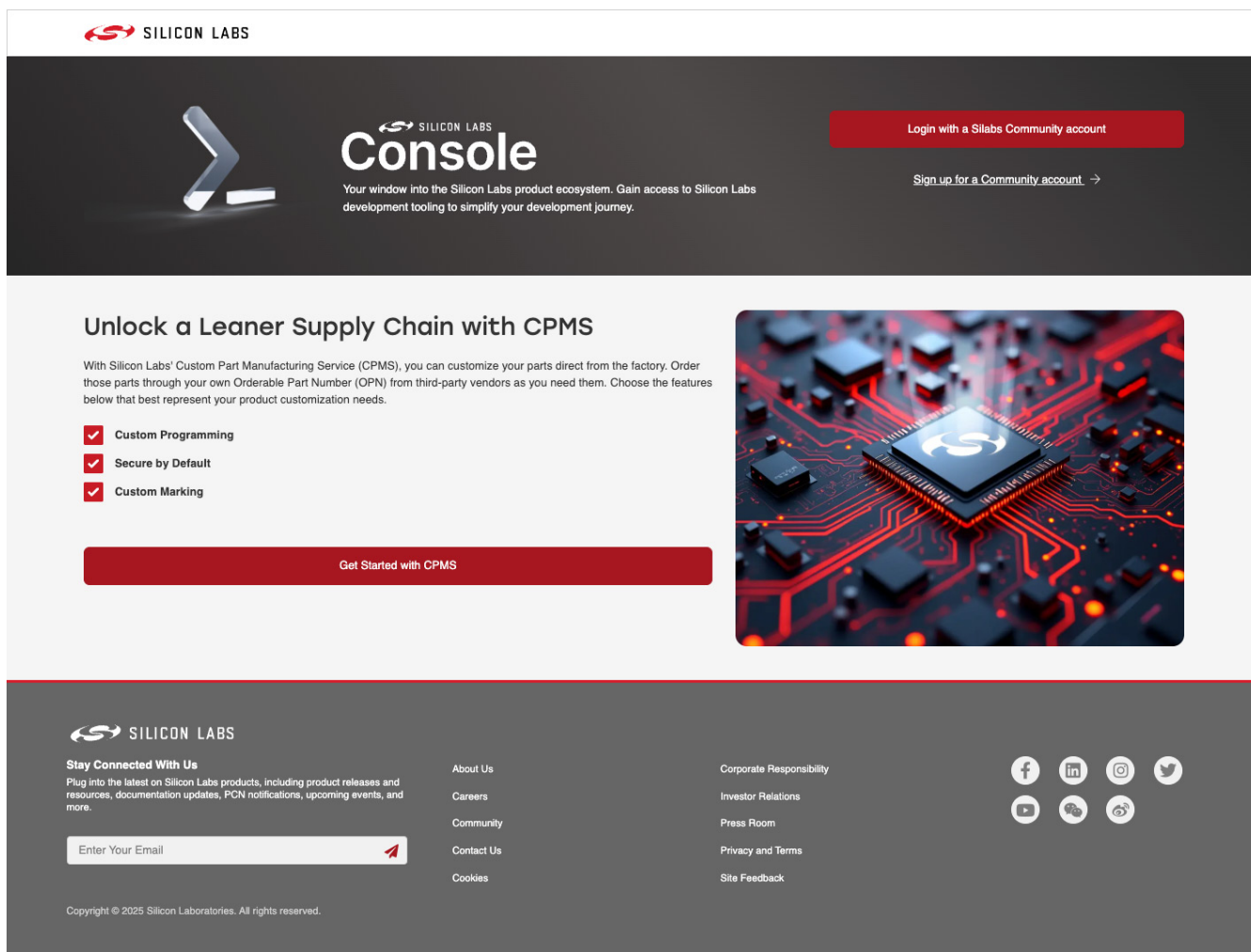
This example will show how to order a custom part that is secure from the moment it leaves Silicon Labs. It has secure boot, secure debug lock, and encrypted upgrades enabled so that an untrusted contract manufacturer cannot access the debug port or upload un- signed and/or unencrypted applications.

This example uses an EFR32MG21B, which is a Secure Vault High part. Secure Vault Base or Mid parts do not have the same customization options, so some sections of this example will not be applicable to such devices.


## CPMS

This section provides detailed information on starting a new custom part in CPMS and configuring the debug lock and Secure Boot.

1. In a browser, open CPMS at <https://console.silabs.com/cpms>.



2. Log in using your [www.silabs.com](https://www.silabs.com) account credentials.



### Custom Part Manufacturing Service (CPMS)


Start your customization by selecting a part and giving that part a name to be used in CPMS.

Give your part a name.

Find a Base Part to customize.

Create New Customization

a. Name: Enter *Example-1*. This name will be used within CPMS to help differentiate between custom devices.



### Custom Part Manufacturing Service (CPMS)


Start your customization by selecting a part and giving that part a name to be used in CPMS.

Give your part a name.

Find a Base Part to customize.

EFR32MG21B010F1024IM32-B	ZigBee and Thread	1024kB	Wireless
EFR32MG21B010F1024IM32-D	ZigBee and Thread	1024kB	Wireless
EFR32MG21B010F512IM32-B	ZigBee and Thread	512kB	Wireless
EFR32MG21B010F512IM32-D	ZigBee and Thread	512kB	Wireless
EFR32MG21B010F768IM32-B	ZigBee and Thread	768kB	Wireless
EFR32MG21B010F768IM32-D	ZigBee and Thread	768kB	Wireless


b. Part: Select any Secure Vault Mid or High part. In this example, select the part **EFR32MG21B010F1024IM32-B**. As you begin to type your part, the list will filter the part list based on your entry.




### Custom Part Manufacturing Service (CPMS)

Start your customization by selecting a part and giving that part a name to be used in CPMS.

Give your part a name.


Example-1 


Find a Base Part to customize.

EFR32MG21B010F512IM32-D 

Create New Customization

- c. Once you have successfully entered a name for your order and selected the part to be customized, the **Create New Customization** button is enabled. You can begin to customize this part for your sample order.
4. Click **Create New Customization**. This takes you to the part customization page. Change the following configurations (configurations not listed can be left as the default):
- a. Debug Lock: Select **Secure**.

 **Security Options**

**SE Version v1.2.16 (latest)** 

We recommend using the latest SE version to ensure all patches are in place. We further recommend that you implement the ability to apply SE updates in your manufacturing line and over the air in the event new vulnerabilities are patched.

**Debug Lock**

☐ Standard ☒ **Secure** ☐ Permanent ☐ Unlocked

The debug access port connected to the Series 2 device's Cortex-M33 processor can be closed by issuing commands to the Secure Element, either from a debugger over DCI or through the mailbox interface. Three properties govern the behavior of the debug lock. Locking the part reduces the general attack surface and prevents information leakage post Silicon Labs manufacturing.

[Read more about secure debug](#)

**TASKS**

Please correct the following tasks.

☐ Please flash at least a bootloader (because Debug Lock is not Unlocked)

☐ Provide a public command key (because Secure Debug Lock is selected)

Complete all tasks before submitting order.

Review Order

- b. Configure Secure Boot, Flash Lock, and Tamper Settings: On. Turn off **Require Verify Certificate before secure boot**, since this example will not use certificates.

Configure Secure Boot and Tamper Response Settings

These configurations can only be made at one time and are **irreversible** once they are made. Read more about [secure boot with RTSL](#) and [production programming](#)

☒ **Enable Secure Boot with RTSL**  
If set, authenticates the first code image in flash memory, which is typically the second stage bootloader, before allowing that code to run. Enabling secure boot will ensure that the device will only boot code that has been properly signed by you.

☐ **Require Verify Certificate before secure boot**  
The Verify intermediate certificate before secure boot option provisions the Public Sign Key to enable certificate-based Secure Boot. Enabling this reduces the need to access the OTP signing key allowing more stringent access restrictions. It also provides the ability to roll the intermediate key in the event it is compromised.

☐ **Enable Anti Rollback**  
We recommend enabling anti-rollback. If set, the first stage bootloader will compare the version of the first image in flash memory, which is typically the second stage bootloader, to the version of the image that has been staged for upgrade. If the staged image has a version that is **greater than** the current image, the upgrade will succeed. Otherwise the upgrade operation will be ignored.

☐ **Enable Flash Page Locking**  
This feature write/erase locks flash pages starting at 0 that have been validated by the first stage bootloader signature check. This will prevent flash modification of the locked pages by any means other than through the hardware secure engine (write/erase attempts from the CPU or from the debug port will be ignored).  
"Full" - locks from page 0 up to and including the page containing the signature. This may lock flash bytes that are

TASKS

Please correct the following tasks.

☐ Please flash at least a bootloader (because Debug Lock is not Unlocked)

☐ Provide a public command key (because Secure Debug Lock is selected)

☐ Provide a public sign key (because secure boot is enabled)

Complete all tasks before submitting order.

Review Order

c. Before you enter the keys and images, you need to generate them. This is covered in the following sections.

## Generating the Application

Follow the instructions below to generate and configure an application.

1. Open Simplicity Studio.

2. In the Launcher view, click **EXAMPLE PROJECTS & DEMOS**.

3. Search for *blink*, and select the **Platform - Blink Bare-metal** project.

4. Click **Finish**.

5. There should now be a `blink_baremetal` project open in the Simplicity IDE view. Open `blink_baremetal.slc`.


blink\_baremetal

OVERVIEW

SOFTWARE COMPONENTS

CONFIGURATION TOOLS

Target and SDK Selection



**EFR32MG21B010F1024IM32**  
EFR32xG21B 2.4 GHz 10 dBm Radio Board (BRD4181C)  
Wireless Starter Kit Mainboard (BRD4001A Rev A01)

Change Target/SDK

Project Details

blink\_baremetal

This example project shows how to blink an LED in a bare-metal configuration.

Category

Example|Platform

Preferred SDK

Gecko SDK Suite: Amazon, Bluetooth 3.2.2, Bluetooth Mesh 2.1.2, EmberZNet 6.10.2.0, Flex 3.2.2.0, HomeKit 1.0.2.0, MCU 6.1.2.0, Micrium OS Kernel, OpenThread 1.2.2.0 (GitHub-48b129e74), Platform 3.2.1.0, Wi-SUN 1.1.1.0, Z-Wave SDK 7.16.2.0

Import Mode

Link sdk and copy project sources

Force Generation

Project Generators

Simplicity IDE Project

A Simplicity IDE project supporting builds for MCUs using C/C++ and assembly files.

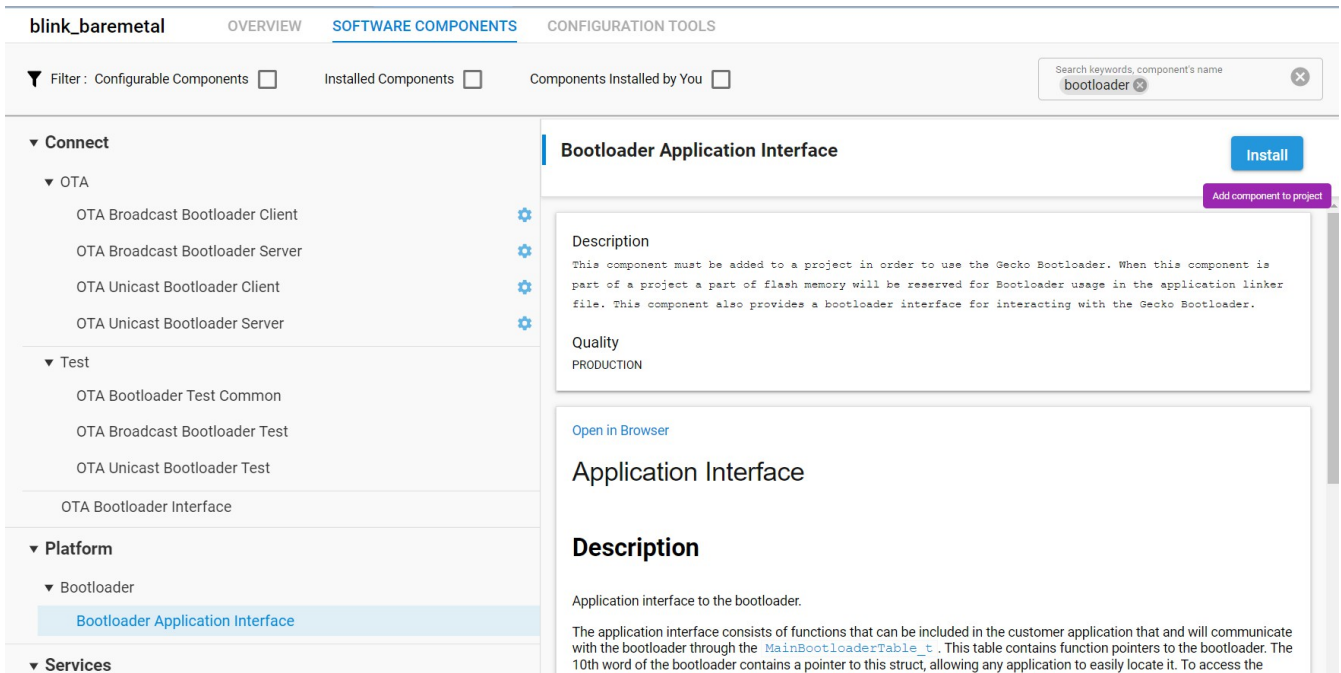
Edit

6. Click on the **SOFTWARE COMPONENTS** tab.

7. In the Search bar, search for *bootloader*.

8. Click **Platform > Bootloader > Bootloader Application Interface**, and click **Install**.





**blink\_baremetal** OVERVIEW SOFTWARE COMPONENTS CONFIGURATION TOOLS

Filter: Configurable Components ☐ Installed Components ☐ Components Installed by You ☐

Search keywords, component's name:

- ▼ Connect
  - ▼ OTA
    - OTA Broadcast Bootloader Client
    - OTA Broadcast Bootloader Server
    - OTA Unicast Bootloader Client
    - OTA Unicast Bootloader Server
  - ▼ Test
    - OTA Bootloader Test Common
    - OTA Broadcast Bootloader Test
    - OTA Unicast Bootloader Test
  - OTA Bootloader Interface
- ▼ Platform
  - ▼ Bootloader
    - Bootloader Application Interface**
- ▼ Services

**Bootloader Application Interface** Install Add component to project

**Description**

This component must be added to a project in order to use the Gecko Bootloader. When this component is part of a project a part of flash memory will be reserved for Bootloader usage in the application linker file. This component also provides a bootloader interface for interacting with the Gecko Bootloader.

**Quality**  
PRODUCTION

[Open in Browser](#)

**Application Interface**

**Description**

Application interface to the bootloader.

The application interface consists of functions that can be included in the customer application that will communicate with the bootloader through the `MainBootloaderTable_t`. This table contains function pointers to the bootloader. The 10th word of the bootloader contains a pointer to this struct, allowing any application to easily locate it. To access the

9. The application image will need an `application_properties.c` file as shown below to enable secure boot. The `.cert` pointer is set to NULL to disable the application certificate option. The `signatureType` and `signatureLocation` fields are filled by Simplicity Commander when signing the application image using the `convert` command.

```

1  #include <stdint.h>
2  #include "application_properties.h"
3
4  // Application version number (uint32_t) for anti-rollback
5  #define APP_PROPERTIES_VERSION (0UL)
6  // Application properties for secure boot
7  const ApplicationProperties_t sl_app_properties = {
8      .magic = APPLICATION_PROPERTIES_MAGIC,
9      .structVersion = APPLICATION_PROPERTIES_VERSION,
10     .signatureType = APPLICATION_SIGNATURE_NONE,
11     .signatureLocation = 0,
12     .app = {
13         .type = APPLICATION_TYPE_MCU,
14         .version = APP_PROPERTIES_VERSION,
15         .capabilities = 0UL,
16         .productId = { 0U },
17     },
18     .cert = NULL,
19     .longTokenSectionAddress = NULL,
20 };
21
22

```

10. Now that the configuration is set, "Build" the project. This will generate binaries for the project.



## Generating the Bootloader

Follow the steps below to generate and configure a bootloader.

1. Now go back to the Launcher and search for *bootloader*.
2. Locate and **Create** the **Internal Storage Bootloader (single image on 1MB device)** example.
3. Open **bootloader-storage-internal-single.slc**.
4. Click the **Software Components** tab, search for **Bootloader Core** and then click **Configure**.
5. Click **Require encrypted firmware upgrade files** and **Enable Secure Boot**.

**Bootloader Core**Pin Tool</> View SourceX

### Bootloader Core Configuration

Require signed firmware upgrade files <input type="checkbox"/>	Require encrypted firmware upgrade files <input checked="" type="checkbox"/>	Use symmetric key stored in Secure Element storage <input type="checkbox"/>	Use symmetric key stored in Application Properties Struct <input type="checkbox"/>
Allow use of public key from manufacturing token storage <input checked="" type="checkbox"/>	Prevent bootloader write/erase <input type="checkbox"/>	Upgrade SE without using the staging area <input type="checkbox"/>	Base address of bootloader upgrade image <div>0x8000</div>
Bootloader Version Main Customer <div>0</div>			

**Enable secure boot**☒  
Prevent write/erase of verified application  
☐

6. Build the project (if the build button is greyed out, you may need to click on the project in the Project Explorer).

## Generating the Sign Key, the Command Key, and the OTA Decryption Key

Enabling secure boot and secure debug requires importing public keys. Ideally, these keys would be generated and managed by an HSM. This example will use Commander.

1. Create a sign key pair for secure boot.

```
commander util genkey --type ecc-p256 --privkey cpms-sign-priv.pem -- pubkey cpms-sign-pub.pem
```

```
Generating ECC P256 key pair...
Writing private key file in PEM format to cpms-sign-priv.pem
Writing public key file in PEM format to cpms-sign-pub.pem
DONE
```

2. Create a command key pair for secure debug:

```
commander util genkey --type ecc-p256 --privkey cpms-cmd-priv.pem -- pubkey cpms-cmd-pub.pem
```



```
Generating ECC P256 key pair...
Writing private key file in PEM format to cpms-cmd-priv.pem
Writing public key file in PEM format to cpms-cmd-pub.pem
DONE
```

3. Create an OTA decryption/encryption key for GBL upgrades:

```
commander util genkey --type aes-ccm --outfile cpms-gbl.txt
```

```
Using Windows' Cryptographic random number generator
DONE
```

## Signing and Merging the Application and Bootloader Images

We now need to prepare our application and bootloader for CPMS. First, we need to sign the images. Then, since CPMS requires the firmware image to be in one file, we need to merge the signed hex files. We will do this using the Simplicity Commander command line interface.

1. Open a terminal and navigate to your Simplicity Studio workspace.
2. Sign the bootloader:

```
commander convert internal-storage-bootloader-single.hex --secureboot --keyfile cpms-sign-priv.pem --outfile cpms-btl-signed.hex
```

```
Parsing file internal-storage-bootloader-single.hex...
Found Application Properties at 0x000024a8
Writing Application Properties signature pointer to point to 0x000025e0
Setting signature type in Application Properties: 0x00000001
Image SHA256: ca36debc860cdb720aabe9fdd37dc730172fe34571aedc452b52f9ef5a824264
R = 3E8E58AF660F769FE25E9262E6899188B61716723352367F0EC96DF6C7133B20
S = 5C36A7B3124F320C9B9B56B80D2F1A1D8B3593BC008E11B50015E3BEE4638537
Writing to cpms-btl-signed.hex
DONE
```

This will create the **cpms-btl-signed.hex** signed image file in your workspace.

3. Sign the application:

```
commander convert blink_baremetal.hex --secureboot --keyfile cpms-sign-priv.pem --outfile cpms-app-signed.hex
```

```
Parsing file blink_baremetal.hex...
Found Application Properties at 0x000061bc
Writing Application Properties signature pointer to point to 0x000064d8
Setting signature type in Application Properties: 0x00000001
Image SHA256: 030b8cdb43e7666b1a015ada8a658a96169be086177548b692a385edb5840295
R = 0C64B8EC9FEFD081EFEBF08E0744A13CA606BD654C1A6B108AF2F5C06AECD5A1
S = CA9DE6279F50C86CD317365FD98380D097D90764A9EDEF06623FE9126763844
Writing to cpms-app-signed.hex
DONE
```

This will create the **cpms-app-signed.hex** signed image file in your workspace.

4. Merge the signed hex files:

```
commander convert cpms-app-signed.hex cpms-btl-signed.hex -o cpms-merged.hex
```

```
Parsing file cpms-app-signed.hex...
Parsing file cpms-btl-signed.hex...
Writing to cpms-merged.hex...
DONE
```

This will create **cpms-merged.hex** in your workspace.

## Programming the Keys and Flash Memory

This section describes how to upload the public sign key and the merged signed hex file.

- 1. In CPMS, return to the **Standard Security Keys** section.
- 2. Click the blue upload button in the **Public Sign Key** field. In the file explorer pop-up, find and select the **cpms-sign-pub.pem** file.

Standard Security Keys

Public Sign Key

0x 049FCB906DD4F282715C049C82FE83A0B0C0E0438216E67A63D4495E399278797

This key is used for binary authentication and/or OTA upgrade payload authentication. If you enabled secure boot, you must provide the public part of the key you used to sign your bootloader or application image here. (eg. 0x04123456789...ABCEDF, total 65 bytes. You can also upload a .pem, .der, or .pub file)

Public Command Key

0x

This key is used for Secure Debug Unlock or Disable Tamper command authentication. If you chose secure debug lock, you must provide the public part of your command key here. (eg. 0x04123456789...ABCEDF total 65 bytes. You can also upload a .pem, .der, or .pub file)

TASKS

Please correct the following tasks.

☐ Please flash at least a bootloader (because Debug Lock is not Unlocked)

☐ Provide a public command key (because Secure Debug Lock is selected)

Complete all tasks before submitting order.

Review Order

- 3. Click the blue upload button in the **Public Command Key** field. In the file explorer pop-up, find and select the **cpms-cmd-pub.pem** file.

Standard Security Keys

Public Sign Key

0x 049FCB906DD4F282715C049C82FE83A0B0C0E0438216E67A63D4495E399278797

This key is used for binary authentication and/or OTA upgrade payload authentication. If you enabled secure boot, you must provide the public part of the key you used to sign your bootloader or application image here. (eg. 0x04123456789...ABCEDF, total 65 bytes. You can also upload a .pem, .der, or .pub file)

Public Command Key

0x 8CFE1457BA79E55E92EBE595E57832321C627E078DD7602C7D493000FB943C0

This key is used for Secure Debug Unlock or Disable Tamper command authentication. If you chose secure debug lock, you must provide the public part of your command key here. (eg. 0x04123456789...ABCEDF total 65 bytes. You can also upload a .pem, .der, or .pub file)

TASKS

Please correct the following tasks.

☐ Please flash at least a bootloader (because Debug Lock is not Unlocked)

Complete all tasks before submitting order.

Review Order

- 4. For the OTA Decryption Key, copy the key value (in hex) from **cpms-gbl.txt** into the **OTA Decryption Key** field.

OTA Decryption Key

0x D374A93C78C6A115D8F51D287C633165

This key is used for decrypting GBL payloads used for firmware upgrades. (eg. 0x0123456789...ABCEDF total 16 bytes.)

- 5. Scroll down to the **Flash Programming** section.

Flash Programming

Flash Programming involves the addition of customer specific code to a standard product. Customer code in **INTEL HEX** format is required.

Firmware

Fill Character

0x FF

We will fill unused or unspecified addresses of the flash with the byte you provide here.

Upload new Intel HEX files

Select Firmware Type

☐ App only

☐ Bootloader only

☒ App and Bootloader

Standard File Input

CLICK HERE OR DRAG & DROP TO UPLOAD A FILE

6. Click **CLICK HERE OR DRAG DROP TO UPLOAD A FILE**.
7. Navigate to your workspace. On Windows this will be in **C:/Users/<username>/SimplicityStudio/v5\_workspace**.
8. Select **cpms-merged.hex** and click **Open**. CPMS only accepts Intel Hex files for firmware images.
9. You should now be able to see the binary added to your customization in CPMS.

Flash Programming

Flash Programming involves the addition of customer specific code to a standard product. Customer code in **INTEL HEX** format is required.

Firmware

Fill Character

0x FF

We will fill unused or unspecified addresses of the flash with the byte you provide here.

Upload new Intel HEX files

Select Firmware Type

☐ App only

☐ Bootloader only

☒ App and Bootloader

Standard File Input

CLICK HERE OR DRAG & DROP TO UPLOAD A FILE


AN1363.hex has been uploaded

Uploaded Files

AN1363.hex


App and Bootloader


10. Scroll to the top of the page, and click **Review Order**.





Title: **Example-1**


Base Part: **EFR32MG21B010F1024IM32-B**





 Select Part


 **Customize**


 Review

 Payment

 Processing


 Shipping

 Complete




### Customize Your Part

Use the form below to configure your custom part. Your input is autosaved and you can leave this page and come back at any time to complete it. Incomplete orders are retained for 30 days from last access.



### Security Options

**SE Version v1.2.16 (latest)** 

We recommend using the latest SE version to ensure all patches are in place. We further recommend that you implement the ability to apply SE updates in your manufacturing line and over the air in the event new vulnerabilities are patched.

**Debug Lock**


☐ Standard

☒ **Secure**

☐ Permanent

☐ Unlocked

The debug access port connected to the Series 2 device's Cortex-M33 processor can be closed by issuing commands to the Secure Element, either from a debugger over DCI or through the mailbox interface. Three properties govern the behavior of the debug lock. Locking the part reduces the general attack surface and prevents information leakage post Silicon Labs manufacturing.



Congratulations! Your OPN programming data is valid and ready to be reviewed for sample programming. You can leave this page and return at any time to complete your order. Incomplete orders are retained 30 days from last access. [Show less](#)

Review Order

11. You can now review the pricing for the custom part and the security configurations you've entered.

## Import Custom Wrapped Keys Example

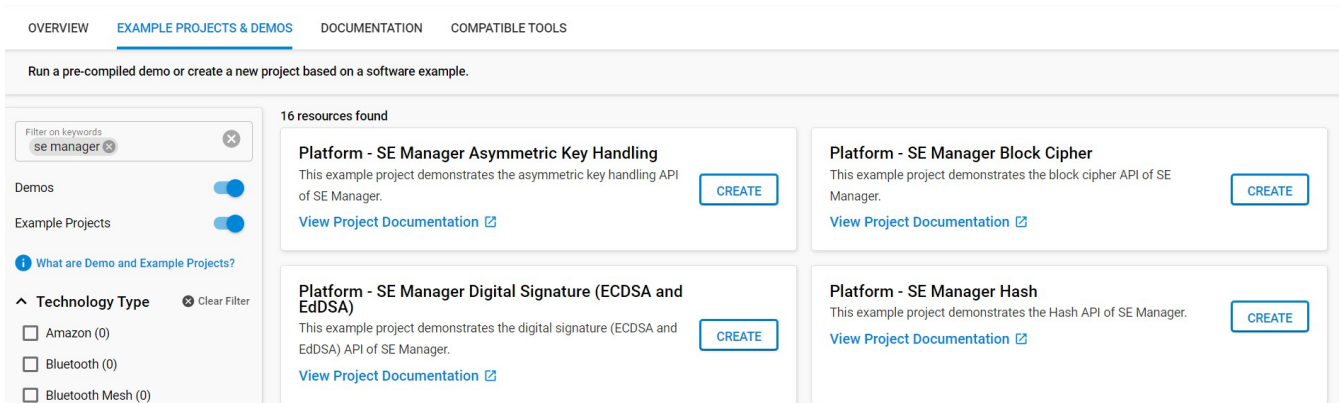
# Importing Custom Wrapped Keys Example

To import custom wrapped keys into CPMS, you need four fields: value, address, auth, and metadata. The following examples will show how to get the metadata value for an asymmetric and a symmetric key.

## Example #1: Importing Custom Wrapped Asymmetric Keys

1. In Simplicity Studio, in the Launcher view, click **EXAMPLE PROJECTS & DEMOS**.
2. Search for *SE Manager*.
3. Create a project from the **Platform - SE Manager Digital Signature (ECDSA and EdDSA)** example.

EFR32xG21B 2.4 GHz 10 dBm RB, WSTK Mainboard (ID: 000440169815)



The screenshot shows the Simplicity Studio Launcher interface. At the top, there are tabs: OVERVIEW, EXAMPLE PROJECTS & DEMOS (selected), DOCUMENTATION, and COMPATIBLE TOOLS. Below the tabs, a message says "Run a pre-compiled demo or create a new project based on a software example." On the left, there is a sidebar with a search bar containing "se manager" and a filter section for "Technology Type" with options: Amazon (0), Bluetooth (0), and Bluetooth Mesh (0). The main area displays "16 resources found" and lists four project examples, each with a "CREATE" button and a "View Project Documentation" link:

- Platform - SE Manager Asymmetric Key Handling**: This example project demonstrates the asymmetric key handling API of SE Manager.
- Platform - SE Manager Block Cipher**: This example project demonstrates the block cipher API of SE Manager.
- Platform - SE Manager Digital Signature (ECDSA and EdDSA)**: This example project demonstrates the digital signature (ECDSA and EdDSA) API of SE Manager.
- Platform - SE Manager Hash**: This example project demonstrates the Hash API of SE Manager.

4. CPMS will automatically wrap your key and write it into flash. To emulate that for testing, use the Memory System Controller to write the key into flash. To enable the MSC, first open `se_manager_signature.slcp`.
5. Open the **SOFTWARE COMPONENTS** tab.
6. Search for *msc*.
7. Click the MSC Peripheral and click **Install**.

blink\_baremetal OVERVIEW SOFTWARE COMPONENTS CONFIGURATION TOOLS

Filter: Configurable Components ☐ Installed Components ☐ Components Installed by You ☐

Search keywords, component's name  
bootloader

▼ Connect

▼ OTA

- OTA Broadcast Bootloader Client
- OTA Broadcast Bootloader Server
- OTA Unicast Bootloader Client
- OTA Unicast Bootloader Server

▼ Test

- OTA Bootloader Test Common
- OTA Broadcast Bootloader Test
- OTA Unicast Bootloader Test

OTA Bootloader Interface

▼ Platform

▼ Bootloader

Bootloader Application Interface

▼ Services

### Bootloader Application Interface

[Install](#)

[Add component to project](#)

**Description**

This component must be added to a project in order to use the Gecko Bootloader. When this component is part of a project a part of flash memory will be reserved for Bootloader usage in the application linker file. This component also provides a bootloader interface for interacting with the Gecko Bootloader.

**Quality**

PRODUCTION

[Open in Browser](#)

### Application Interface

### Description

Application interface to the bootloader.

The application interface consists of functions that can be included in the customer application that and will communicate with the bootloader through the `MainBootloaderTable_t`. This table contains function pointers to the bootloader. The 10th word of the bootloader contains a pointer to this struct, allowing any application to easily locate it. To access the

8. Modify the "create\_wrap\_asymmetric\_key" function of `app_se_manager_signature.c` to use our "CPMS key". Instead of generating a key, we will import our ecc key. In `app_se_manager_signature.c` line 255, replace the lines:

```
print_error_cycle(sl_se_generate_key(&cmd_ctx, &asymmetric_key_desc), &cmd_ctx);
```

with the following:

```
// YOUR KEY VALUE GOES HERE:
static uint8_t user_key[64] =
{
    0x79, 0x7D, 0x86, 0xE3, 0x5B, 0xAA, 0x03, 0xA5,
    0xEE, 0x09, 0xAB, 0x5E, 0x7E, 0xB1, 0x2D, 0xC3,
    0x92, 0xFC, 0xCE, 0xDC, 0xD0, 0x2A, 0xB0, 0xF7,
    0x56, 0x5E, 0x73, 0x30, 0x86, 0x1D, 0xAE, 0xD5,
    0xDD, 0x8A, 0x84, 0xA2, 0x87, 0x0F, 0xCC, 0x2B,
    0x70, 0x66, 0xAE, 0xE0, 0x88, 0x44, 0x2C, 0xCC,
    0x0C, 0x53, 0xCE, 0x9D, 0x26, 0xBB, 0xB3, 0x04,
    0xA8, 0xB7, 0xB9, 0xE5, 0x20, 0x43, 0x62, 0xAE
};

sl_se_key_descriptor_t plaintext_desc = {
    .type = key_type,
    .flags = SL_SE_KEY_FLAG_ASYMMETRIC_BUFFER_HAS_PRIVATE_KEY
        | SL_SE_KEY_FLAG_ASYMMETRIC_SIGNING_ONLY,
    .storage.method = SL_SE_KEY_STORAGE_EXTERNAL_PLAINTEXT,
    .storage.location.buffer.pointer = user_key,
    .storage.location.buffer.size = 64
};

if (sl_se_import_key(&cmd_ctx, &plaintext_desc, &asymmetric_key_desc) != SL_STATUS_OK) return SL_STATUS_FAIL;
```

This code will import your key into the Secure Engine, wrap it, then store the wrapped key to the `asymmetric_key_buf` that `asymmetric_key_desc.storage.location.buffer.pointer` is pointing to.

```

247 // The size of the wrapped key buffer must have space for the overhead of the
248 // key wrapping
249 if (sl_se_validate_key(&asymmetric_key_desc) != SL_STATUS_OK
250     || sl_se_get_storage_size(&asymmetric_key_desc, &req_size) != SL_STATUS_OK
251     || asymmetric_key_desc.storage.location.buffer.size < req_size) {
252     return SL_STATUS_FAIL;
253 }
254
255 // YOUR KEY VALUE GOES HERE:
256 static uint8_t user_key[64] =
257 {
258     0x79, 0x7D, 0x86, 0xE3, 0x5B, 0xAA, 0x03, 0xA5,
259     0xEE, 0x09, 0xAB, 0x5E, 0x7E, 0xB1, 0x2D, 0xC3,
260     0x92, 0xFC, 0xCE, 0xDC, 0xD0, 0x2A, 0xB0, 0xF7,
261     0x56, 0x5E, 0x73, 0x30, 0x86, 0x1D, 0xAE, 0xD5,
262     0xDD, 0x8A, 0x84, 0xA2, 0x87, 0x0F, 0xCC, 0x2B,
263     0x70, 0x66, 0xAE, 0xE0, 0x88, 0x44, 0x2C, 0xCC,
264     0x0C, 0x53, 0xCE, 0x9D, 0x26, 0xBB, 0xB3, 0x04,
265     0xA8, 0xB7, 0xB9, 0xE5, 0x20, 0x43, 0x62, 0xAE
266 };
267
268 sl_se_key_descriptor_t plaintext_desc = {
269     .type = key_type,
270     .flags = SL_SE_KEY_FLAG_ASYMMETRIC_BUFFER_HAS_PRIVATE_KEY
271             | SL_SE_KEY_FLAG_ASYMMETRIC_SIGNING_ONLY,
272     .storage.method = SL_SE_KEY_STORAGE_EXTERNAL_PLAINTEXT,
273     .storage.location.buffer.pointer = user_key,
274     .storage.location.buffer.size = 64
275 };
276
277 if (sl_se_import_key(&cmd_ctx, &plaintext_desc, &asymmetric_key_desc) != SL_STATUS_OK)
278     return SL_STATUS_FAIL;
279 }
280
281 //*****
282 * Generate a non-exportable asymmetric key into a volatile SE key slot.

```

9. Next, write the wrapped key blob into flash. Add the following lines to `create_wrap_asymmetric_key`:

```

// YOUR KEY ADDRESS GOES HERE:
unsigned int wrapped_key_address = 0x00080000;

printf("\nWriting key into flash at 0x%08x...\n", wrapped_key_address);

// Clear out the old wrapped key MSC_ErasePage((uint32_t*)wrapped_key_address);

// Flash the new wrapped key MSC_WriteWord((uint32_t*)wrapped_key_address, asymmetric_key_buf, sizeof(asymmetric_key_buf));

// Update the key descriptor to point to the key in flash asymmetric_key_desc.storage.location.buffer.pointer =
(uint8_t*)wrapped_key_address;

```

10. Next, print out the keyspec needed for CPMS. Add the following lines to `create_wrap_asymmetric_key`:



```

275 };
276
277 if (sl_se_import_key(&cmd_ctx, &plaintext_desc, &asymmetric_key_desc) != SL_STATUS_OK)
278     return SL_STATUS_FAIL;
279
280 // YOUR KEY ADDRESS GOES HERE:
281 unsigned int wrapped_key_address = 0x00080000;
282
283 printf("\nWriting key into flash at 0x%08x...\n", wrapped_key_address);
284
285 // Clear out the old wrapped key
286 MSC_ErasePage((uint32_t*)wrapped_key_address);
287
288 // Flash the new wrapped key
289 MSC_WriteWord((uint32_t*)wrapped_key_address, asymmetric_key_buf, sizeof(asymmetric_key_buf));
290
291 // Update the key descriptor to point to the key in flash
292 asymmetric_key_desc.storage.location.buffer.pointer = (uint8_t*)wrapped_key_address;
293 unsigned int keyspec;
294
295 if (sli_se_key_to_keyspec(&asymmetric_key_desc, &keyspec) != SL_STATUS_OK)
296     return SL_STATUS_FAIL;
297
298 printf("\nKeyspec: 0x%08x\n", keyspec);
299
300 return SL_STATUS_OK;
301 }
302
303 //*****

```

```

unsigned int keyspec;

if (sli_se_key_to_keyspec(&asymmetric_key_desc, &keyspec) != SL_STATUS_OK) return SL_STATUS_FAIL;

printf("\nKeyspec: 0x%08x\n", keyspec);

return SL_STATUS_OK;

```

11. Keys imported using CPMS use a different bus master than the CPU, so the key descriptor needs to be updated. In `create_wrap_symmetric_key`, edit the `symmetric_key_desc.flags` field to remove `SL_SE_FLAG_ASYMMETRIC_BUFFER_HAS_PUBLIC_KEY` and add `SL_SE_KEY_FLAG_ALLOW_ANY_ACCESS` (line 229):

```

asymmetric_key_desc.flags = SL_SE_KEY_FLAG_ASYMMETRIC_BUFFER_HAS_PRIVATE_KEY
| SL_SE_KEY_FLAG_ASYMMETRIC_SIGNING_ONLY
| SL_SE_KEY_FLAG_NON_EXPORTABLE
| SL_SE_KEY_FLAG_ALLOW_ANY_ACCESS;

```

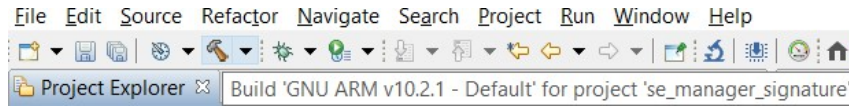
```

228 asymmetric_key_desc.type = key_type;
229 asymmetric_key_desc.flags = SL_SE_KEY_FLAG_ASYMMETRIC_BUFFER_HAS_PRIVATE_KEY
230 | SL_SE_KEY_FLAG_ASYMMETRIC_SIGNING_ONLY
231 | SL_SE_KEY_FLAG_NON_EXPORTABLE
232 | SL_SE_KEY_FLAG_ALLOW_ANY_ACCESS;
233 asymmetric_key_desc.storage.method = SL_SE_KEY_STORAGE_EXTERNAL_WRAPPED;
234 // Set pointer to a RAM buffer to support key generation
235 asymmetric_key_desc.storage.location.buffer.pointer = asymmetric_key_buf;
236 asymmetric_key_desc.storage.location.buffer.size = sizeof(asymmetric_key_buf);
237

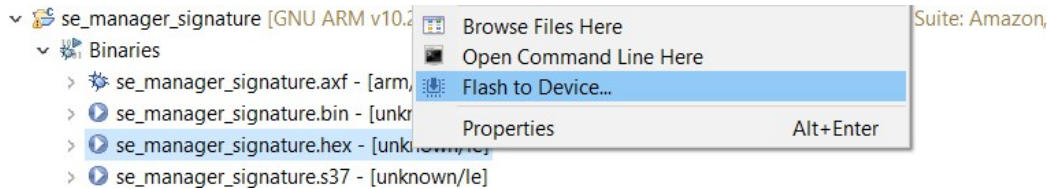
```

12. Build the project.

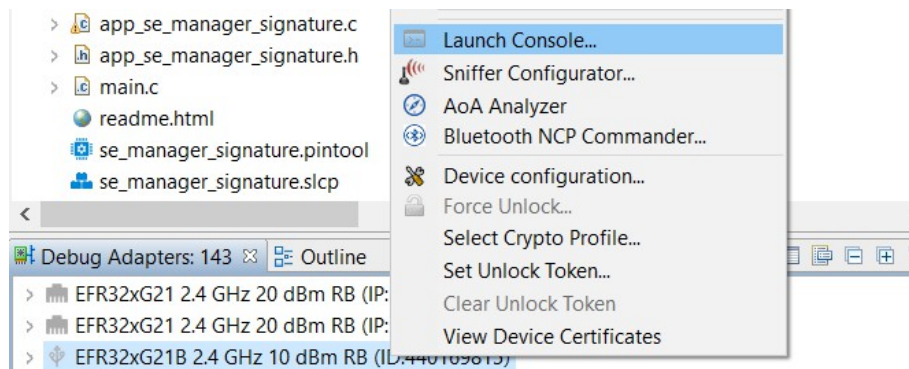




13. Flash to the target device.

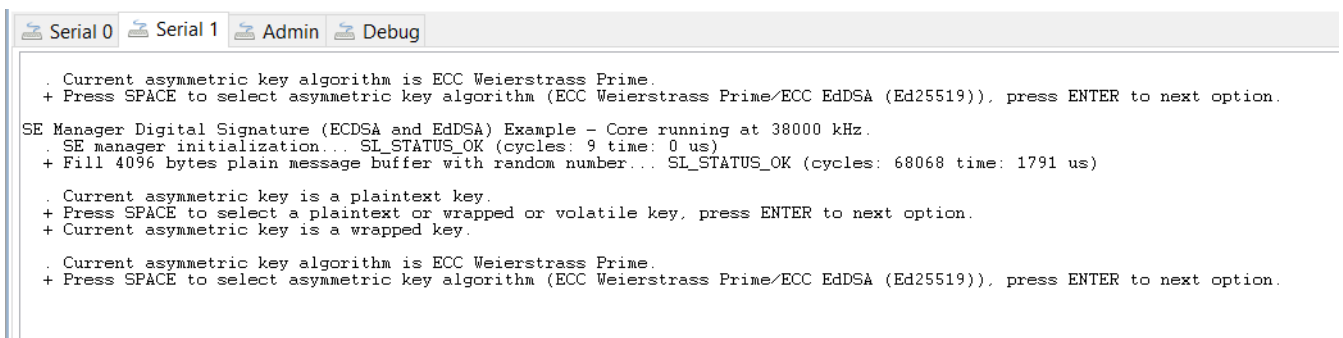


14. In the **Debug Adapters** window, right click on the adapter for your device and click **Launch Console**.



15. Click the **Serial 1** tab, and then send **Enter** to start the console.

16. Reset the device. The program will first ask which type of key you want to use: plaintext, wrapped, or volatile. Type a space, and then press **Enter** to select the second option, **wrapped**.



17. Press **Enter** four more times to see the keyspec printed to the console. When entering a custom wrapped key into CPMS, this value is the **Key Metadata** value.

```

. Digital signature
+ ECC Weierstrass Prime - ECC P192
+ Generate a non-exportable wrapped asymmetric key...
Writing key into flash at 0x00080000...
Keyspec: 0x8900c417
+ Sign 256 bytes message with SHA1 and wrapped private key... SL_STATUS_OK (cycles: 131246 time: 3453 us)
+ Export public key from private key... SL_STATUS_OK (cycles: 118968 time: 3130 us)
+ Verify signature with SHA1 and wrapped public key... SL_STATUS_OK (cycles: 121817 time: 3205 us)

. Current asymmetric key is a wrapped key.
+ Press SPACE to select a plaintext or wrapped or volatile key, press ENTER to next option.

```

18. Now that the key is wrapped and stored in flash, you want to see that the program can use it without having the plaintext key anywhere in the application. Go back to `app_se_manager_signature.c` and comment out lines 255 to 278 and lines 283 to 289.

```

252     return SL_STATUS_FAIL;
253 }
254
255 // // YOUR KEY VALUE GOES HERE:
256 // static uint8_t user_key[64] =
257 // {
258 //     0x79, 0x7D, 0x86, 0xE3, 0x5B, 0xAA, 0x03, 0xA5,
259 //     0xEE, 0x09, 0xAB, 0x5E, 0x7E, 0xB1, 0x2D, 0xC3,
260 //     0x92, 0xFC, 0xCE, 0xDC, 0xD0, 0x2A, 0xB0, 0xF7,
261 //     0x56, 0x5E, 0x73, 0x30, 0x86, 0x1D, 0xAE, 0xD5,
262 //     0xDD, 0x8A, 0x84, 0xA2, 0x87, 0x0F, 0xCC, 0x2B,
263 //     0x70, 0x66, 0xAE, 0xE0, 0x88, 0x44, 0x2C, 0xCC,
264 //     0x0C, 0x53, 0xCE, 0x9D, 0x26, 0xBB, 0xB3, 0x04,
265 //     0xA8, 0xB7, 0xB9, 0xE5, 0x20, 0x43, 0x62, 0xAE
266 // };
267 //
268 // sl_se_key_descriptor_t plaintext_desc = {
269 //     .type = key_type,
270 //     .flags = SL_SE_KEY_FLAG_ASYMMETRIC_BUFFER_HAS_PRIVATE_KEY
271 //         | SL_SE_KEY_FLAG_ASYMMETRIC_SIGNING_ONLY,
272 //     .storage.method = SL_SE_KEY_STORAGE_EXTERNAL_PLAINTEXT,
273 //     .storage.location.buffer.pointer = user_key,
274 //     .storage.location.buffer.size = 64
275 // };
276 //
277 // if (sl_se_import_key(&cmd.ctx, &plaintext_desc, &asymmetric_key_desc) != SL_STATUS_OK)
278 //     return SL_STATUS_FAIL;
279 //
280 // // YOUR KEY ADDRESS GOES HERE:
281 // unsigned int wrapped_key_address = 0x00080000;
282 //
283 // printf("\nWriting key into flash at 0x%08x...\n", wrapped_key_address);
284 //
285 // // Clear out the old wrapped key
286 // MSC_ErasePage((uint32_t*)wrapped_key_address);
287 //
288 // // Flash the new wrapped key
289 // MSC_WriteWord((uint32_t*)wrapped_key_address, asymmetric_key_buf, sizeof(asymmetric_key_buf));
290 //
291 // Update the key descriptor to point to the key in flash
292 asymmetric_key_desc.storage.location.buffer.pointer = (uint8_t*)wrapped_key_address;
293 unsigned int keyspec;
294

```

19. Now the application simply sets up the key descriptor to point to where we wrote the wrapped key in flash, without knowing the value of the key.
20. Repeat steps 12 to 17 to verify that the wrapped key can still be used. Note that if the flash is erased (by a commander device unlock command, for instance), this application will fail. It needs the wrapped key to be stored in flash by a previous process.

```
SE Manager Digital Signature (ECDSA and EdDSA) Example - Core running at 38000 kHz.
. SE manager initialization... SL_STATUS_OK (cycles: 9 time: 0 us)
+ Fill 4096 bytes plain message buffer with random number... SL_STATUS_OK (cycles: 68674 time: 1807 us)

. Current asymmetric key is a plaintext key.
+ Press SPACE to select a plaintext or wrapped or volatile key, press ENTER to next option.
+ Current asymmetric key is a wrapped key.

. Current asymmetric key algorithm is ECC Weierstrass Prime.
+ Press SPACE to select asymmetric key algorithm (ECC Weierstrass Prime/ECC EdDSA (Ed25519)), press ENTER to next option.

. Current ECC Weierstrass Prime key is ECC P192.
+ Press SPACE to select ECC Weierstrass Prime key (ECC P192/ECC P256/ECC P384/ECC P521/ECC Custom (secp256k1 in this example)), press ENTER to next option.

. Current Hash algorithm for signature is SHA1.
+ Press SPACE to select Hash algorithm (SHA1/224/256/384/512) for signature, press ENTER to next option.

. Current data length is 256 bytes.
+ Press SPACE to select data length (256 or 1024 or 4096), press ENTER to run.

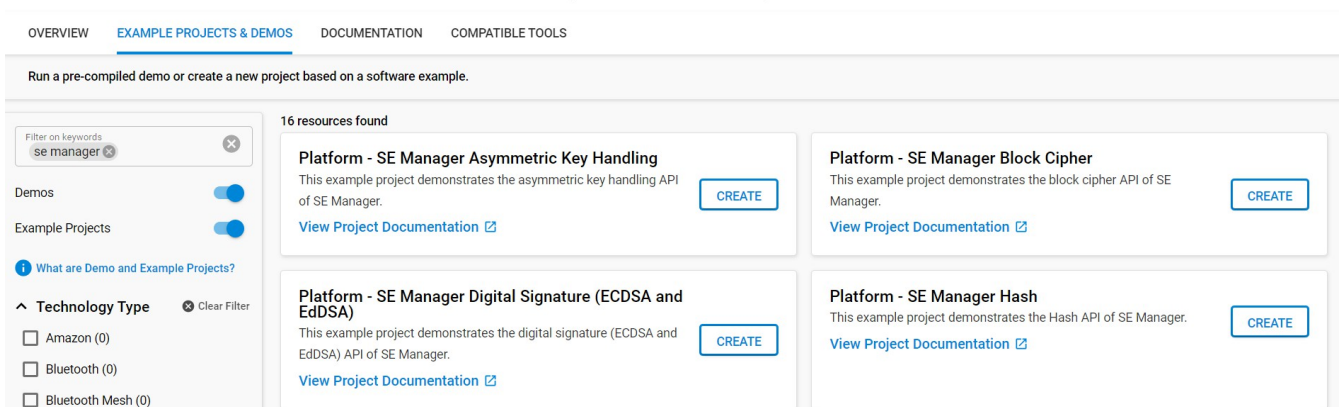
. Digital signature
+ ECC Weierstrass Prime - ECC P192
+ Generate a non-exportable wrapped asymmetric key...
Keyspec: 0x8900c417
+ Sign 256 bytes message with SHA1 and wrapped private key... SL_STATUS_OK (cycles: 125919 time: 3313 us)
+ Export public key from private key... SL_STATUS_OK (cycles: 115894 time: 3049 us)
+ Verify signature with SHA1 and wrapped public key... SL_STATUS_OK (cycles: 122195 time: 3215 us)

. Current asymmetric key is a wrapped key.
+ Press SPACE to select a plaintext or wrapped or volatile key, press ENTER to next option.
```

## Example #2: Importing Custom Wrapped Symmetric Keys

1. In Simplicity Studio, in the Launcher view, click **EXAMPLE PROJECTS & DEMOS**.
2. Search for *SE Manager*.
3. Create a project from the **Platform - SE Manager Block Cipher** example:

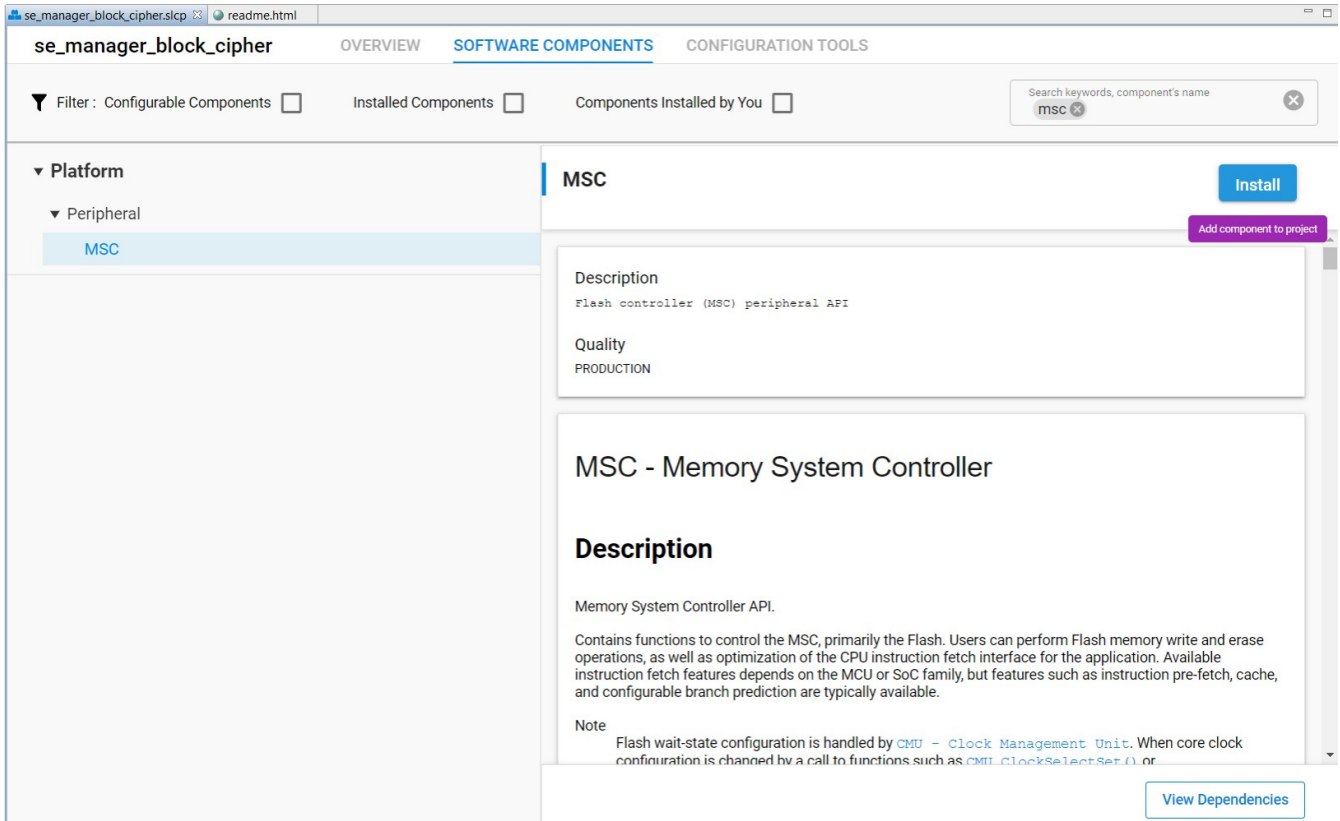
### EFR32xG21B 2.4 GHz 10 dBm RB, WSTK Mainboard (ID: 000440169815)



The screenshot shows the Simplicity Studio Launcher interface. The top navigation bar includes 'OVERVIEW', 'EXAMPLE PROJECTS & DEMOS' (selected), 'DOCUMENTATION', and 'COMPATIBLE TOOLS'. Below the navigation bar, a message states: 'Run a pre-compiled demo or create a new project based on a software example.' On the left, a search filter is set to 'se manager'. The main area displays '16 resources found' and lists four project examples, each with a 'CREATE' button and a link to 'View Project Documentation':

- Platform - SE Manager Asymmetric Key Handling**: This example project demonstrates the asymmetric key handling API of SE Manager.
- Platform - SE Manager Block Cipher**: This example project demonstrates the block cipher API of SE Manager.
- Platform - SE Manager Digital Signature (ECDSA and EdDSA)**: This example project demonstrates the digital signature (ECDSA and EdDSA) API of SE Manager.
- Platform - SE Manager Hash**: This example project demonstrates the Hash API of SE Manager.

4. CPMS will automatically wrap your key and write it into flash. To emulate that for testing, use the Memory System Controller to write the key into flash. To enable the MSC, first open **se\_manager\_block\_cipher.slcp**.
5. Open the **SOFTWARE COMPONENTS** tab.
6. Search for *msc*.
7. Click the MSC Peripheral and click Install.



The screenshot shows the Silicon Labs IDE interface. The top navigation bar includes 'se\_manager\_block\_cipher', 'OVERVIEW', 'SOFTWARE COMPONENTS' (active), and 'CONFIGURATION TOOLS'. Below the navigation bar, there are filter buttons: 'Filter: Configurable Components', 'Installed Components', and 'Components Installed by You'. A search bar on the right contains the text 'msc'. The left sidebar shows a tree view with 'Platform' expanded, then 'Peripheral', and finally 'MSC' selected. The main panel displays the details for the 'MSC' component. It includes an 'Install' button, a description of the 'Flash controller (MSC) peripheral API', a 'Quality' of 'PRODUCTION', and a detailed 'Description' of the 'MSC - Memory System Controller' API. A note at the bottom states: 'Flash wait-state configuration is handled by CMU - Clock Management Unit. When core clock configuration is changed by a call to functions such as CMU.ClockSelectSet() or...'. A 'View Dependencies' button is located at the bottom right of the component details panel.

8. Modify the "create\_wrap\_symmetric\_key" function of app\_se\_manager\_block\_cipher.c to use the "CPMS key". Instead of generating a key, import the aes key. In **app\_se\_manager\_block\_cipher.c** line 259, replace the lines:

```
print_error_cycle(sl_se_generate_key(&cmd_ctx, &symmetric_key_desc), &cmd_ctx);
```

with the following:

```
// YOUR KEY VALUE GOES HERE:
static uint8_t user_key[16] =
{
    0x70, 0xF4, 0x82, 0x4E, 0x49, 0xBD, 0x97, 0xAB,
    0x65, 0x65, 0x32, 0x22, 0xA0, 0x70, 0xB5, 0x16
};

sl_se_key_descriptor_t plaintext_desc = {
    .type = SL_SE_KEY_TYPE_AES_128,
    .flags = 0,
    .storage.method = SL_SE_KEY_STORAGE_EXTERNAL_PLAINTEXT,
    .storage.location.buffer.pointer = user_key,
    .storage.location.buffer.size = 16
};

if (sl_se_import_key(&cmd_ctx, &plaintext_desc, &symmetric_key_desc) != SL_STATUS_OK) return SL_STATUS_FAIL;
```

This code will import your key into the Secure Engine, wrap it, and then store the wrapped key to the **symmetric\_key\_buf** that **symmetric\_key\_desc.storage.location.buffer.pointer** is pointing to.

```

242 sl_status_t create_wrap_symmetric_key(sl_se_key_type_t key_type)
243 {
244     uint32_t req_size;
245
246     symmetric_key_desc.type = key_type;
247     symmetric_key_desc.flags = SL_SE_KEY_FLAG_NON_EXPORTABLE;
248     symmetric_key_desc.storage.method = SL_SE_KEY_STORAGE_EXTERNAL_WRAPPED;
249     symmetric_key_desc.storage.location.buffer.pointer = symmetric_key_buf;
250     symmetric_key_desc.storage.location.buffer.size = sizeof(symmetric_key_buf);
251
252     if ((sl_se_validate_key(&symmetric_key_desc) != SL_STATUS_OK)
253         || (sl_se_get_storage_size(&symmetric_key_desc,
254                                   &req_size) != SL_STATUS_OK)
255         || (sizeof(symmetric_key_buf) < req_size)) {
256         return SL_STATUS_FAIL;
257     }
258
259     // YOUR KEY VALUE GOES HERE:
260     static uint8_t user_key[16] =
261     {
262         0x70, 0xF4, 0x82, 0x4E, 0x49, 0xBD, 0x97, 0xAB,
263         0x65, 0x65, 0x32, 0x22, 0xA0, 0x70, 0xB5, 0x16
264     };
265
266     sl_se_key_descriptor_t plaintext_desc = {
267         .type = SL_SE_KEY_TYPE_AES_128,
268         .flags = 0,
269         .storage.method = SL_SE_KEY_STORAGE_EXTERNAL_PLAINTEXT,
270         .storage.location.buffer.pointer = user_key,
271         .storage.location.buffer.size = 16
272     };
273
274     if (sl_se_import_key(&cmd_ctx, &plaintext_desc, &symmetric_key_desc) != SL_STATUS_OK)
275         return SL_STATUS_FAIL;
276 }

```

9. Next, write the wrapped key blob into flash. Add the following lines to `create_wrap_symmetric_key`:

```

// YOUR KEY ADDRESS GOES HERE:
unsigned int wrapped_key_address = 0x00080000;

printf("Writing key into flash at 0x%08x...\n", wrapped_key_address);

// Clear out the old wrapped key MSC_ErasePage((uint32_t*)wrapped_key_address);

// Flash the new wrapped key
MSC_WriteWord((uint32_t*)wrapped_key_address, symmetric_key_buf, sizeof(symmetric_key_buf));

// Update the key descriptor to point to the key in flash symmetric_key_desc.storage.location.buffer.pointer =
(uint8_t*)wrapped_key_address;

```

10. Next, we'll print out the keyspec that we need for CPMS. Add the following lines to `create_wrap_symmetric_key`:

```

unsigned int keyspec;

if (sl_se_key_to_keyspec(&symmetric_key_desc, &keyspec) != SL_STATUS_OK) return SL_STATUS_FAIL;

printf("\nKeyspec: 0x%08x\n", keyspec);

return SL_STATUS_OK;

```



```

274 if (sl_se_import_key(&cmd_ctx, &plaintext_desc, &symmetric_key_desc) != SL_STATUS_OK)
275     return SL_STATUS_FAIL;
276
277 // YOUR KEY ADDRESS GOES HERE:
278 unsigned int wrapped_key_address = 0x00080000;
279
280 printf("Writing key into flash at 0x%08x...\n", wrapped_key_address);
281
282 // Clear out the old wrapped key
283 MSC_ErasePage((uint32_t*)wrapped_key_address);
284
285 // Flash the new wrapped key
286 MSC_WriteWord((uint32_t*)wrapped_key_address, symmetric_key_buf, sizeof(symmetric_key_buf));
287
288 // Update the key descriptor to point to the key in flash
289 symmetric_key_desc.storage.location.buffer.pointer = (uint8_t*)wrapped_key_address;
290
291 unsigned int keyspec;
292
293 if (sli_se_key_to_keyspec(&symmetric_key_desc, &keyspec) != SL_STATUS_OK)
294     return SL_STATUS_FAIL;
295
296 printf("\nKeyspec: 0x%08x\n", keyspec);
297
298 return SL_STATUS_OK;
299 }

```

11. Keys imported using CPMS use a different bus master than the CPU, so the key descriptor needs to be updated. In `create_wrap_symmetric_key`, edit the `symmetric_key_desc.flags` field to include `SL_SE_KEY_FLAG_ALLOW_ANY_ACCESS` (line 247):

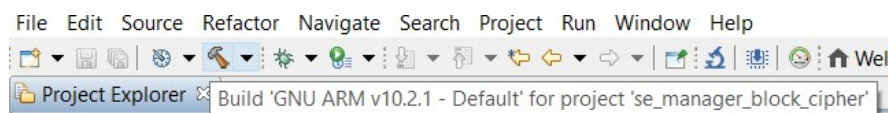
```
symmetric_key_desc.flags = SL_SE_KEY_FLAG_NON_EXPORTABLE | SL_SE_KEY_FLAG_ALLOW_ANY_ACCESS;
```

```

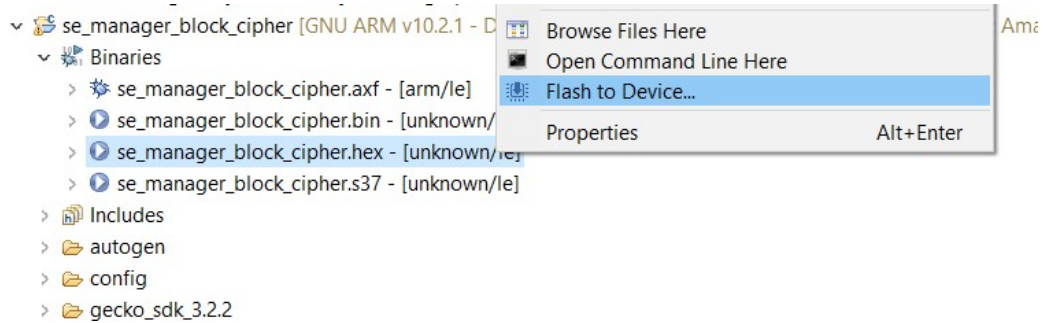
245
246 symmetric_key_desc.type = key_type;
247 symmetric_key_desc.flags = SL_SE_KEY_FLAG_NON_EXPORTABLE | SL_SE_KEY_FLAG_ALLOW_ANY_ACCESS;
248 symmetric_key_desc.storage.method = SL_SE_KEY_STORAGE_EXTERNAL_WRAPPED;
249 symmetric_key_desc.storage.location.buffer.pointer = symmetric_key_buf;
250 symmetric_key_desc.storage.location.buffer.size = sizeof(symmetric_key_buf);
251

```

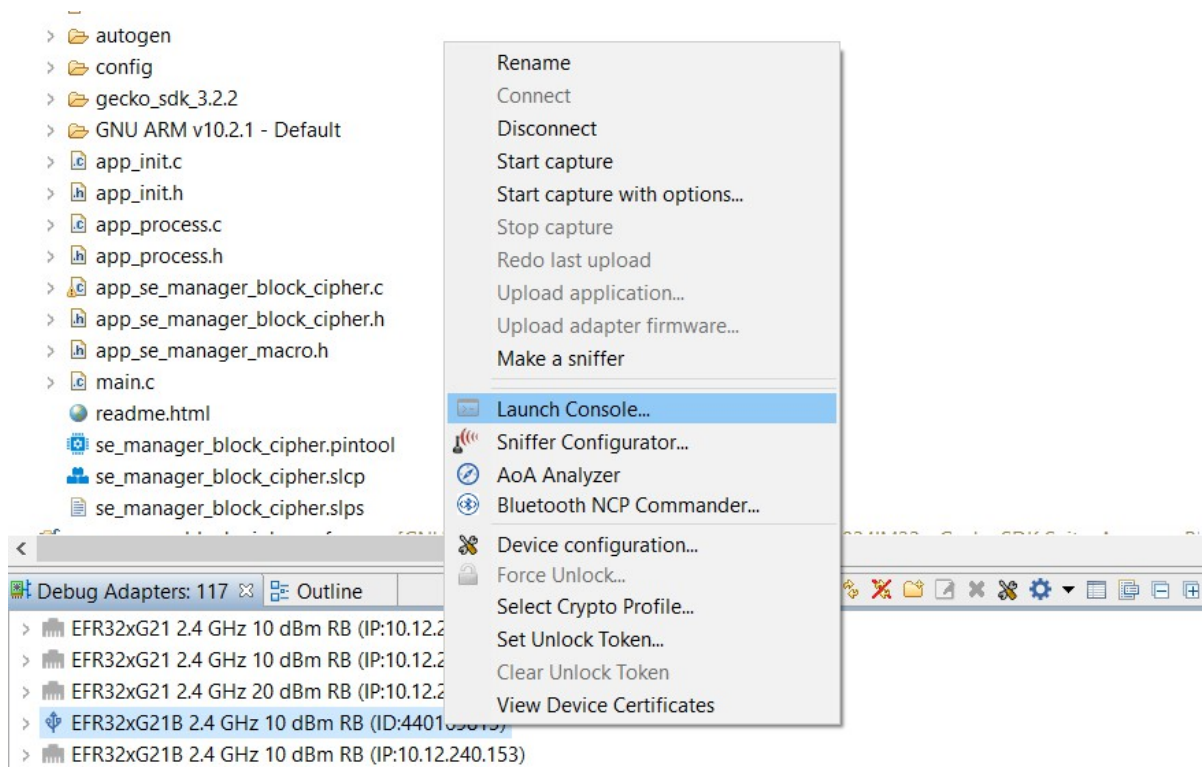
12. Build the project.



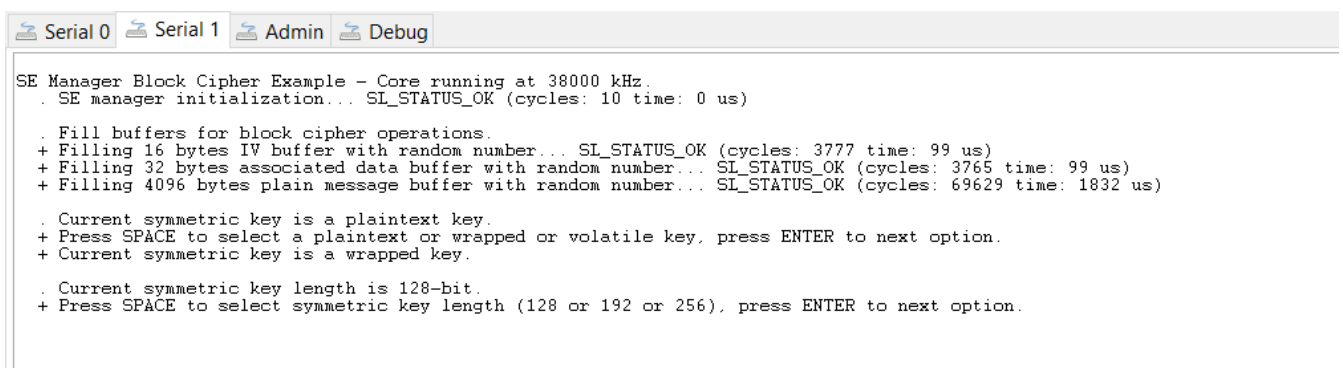
13. Flash to the target device.



14. In the **Debug Adapters** window, right click on the adapter for your device and click **Launch Console**.



15. Click the **Serial 1** tab, and then reset the device. The program will first ask which type of key you want to use: plaintext, wrapped, or volatile. Type a space, and then press **Enter** to select the second option, **wrapped**.



16. Press **Enter** once more to see the keyspec printed to the console. When entering a custom wrapped key into CPMS, this value is the "Key Metadata" value.

```
. Current symmetric key length is 128-bit.
+ Press SPACE to select symmetric key length (128 or 192 or 256), press ENTER to next option.
+ Generating a 128-bit non-exportable symmetric wrapped key... Writing key into flash at 0x00080000...

Keyspec: 0x09008010

. Current data length is 256 bytes.
+ Press SPACE to select data length (256 or 1024 or 4096), press ENTER to next option.
```

17. Press **Enter** two more times to verify that the key can be used without error. Note that if you press **Enter** after this, the program will try to use that key as a ChaCha20-Poly1305 key, and it will fail.

```
Keyspec: 0x09008010

. Current data length is 256 bytes.
+ Press SPACE to select data length (256 or 1024 or 4096), press ENTER to next option.

. Current Hash algorithm for HMAC is SHA1.
+ Press SPACE to select Hash algorithm (SHA1/224/256/384/512) for HMAC, press ENTER to run.

. AES ECB test
+ Encrypting 256 bytes plaintext with 128 bit key... SI_STATUS_OK (cycles: 15379 time: 404 us)
+ Decrypting 256 bytes ciphertext with 128 bit key... SI_STATUS_OK (cycles: 15507 time: 408 us)
+ Comparing decrypted message with plain message... OK

. AES CTR test
+ Encrypting 256 bytes plaintext with 128 bit key... SI_STATUS_OK (cycles: 15527 time: 408 us)
+ Decrypting 256 bytes ciphertext with 128 bit key... SI_STATUS_OK (cycles: 15476 time: 407 us)
+ Comparing decrypted message with plain message... OK

. AES CCM test
+ Encrypting 256 bytes plaintext with 128 bit key... SI_STATUS_OK (cycles: 16101 time: 423 us)
+ Decrypting 256 bytes ciphertext with 128 bit key... SI_STATUS_OK (cycles: 17067 time: 449 us)
+ Comparing decrypted message with plain message... OK

. AES GCM test
+ Encrypting 256 bytes plaintext with 128 bit key... SI_STATUS_OK (cycles: 16333 time: 429 us)
+ Decrypting 256 bytes ciphertext with 128 bit key... SI_STATUS_OK (cycles: 16504 time: 434 us)
+ Comparing decrypted message with plain message... OK

. AES CBC test
+ Encrypting 256 bytes plaintext with 128 bit key... SI_STATUS_OK (cycles: 15333 time: 403 us)
+ Decrypting 256 bytes ciphertext with 128 bit key... SI_STATUS_OK (cycles: 15137 time: 398 us)
+ Comparing decrypted message with plain message... OK

. AES CFB8 test
+ Encrypting 256 bytes plaintext with 128 bit key... SI_STATUS_OK (cycles: 3985747 time: 104 ms)
+ Decrypting 256 bytes ciphertext with 128 bit key... SI_STATUS_OK (cycles: 3985039 time: 104 ms)
+ Comparing decrypted message with plain message... OK

. AES CFB128 test
+ Encrypting 256 bytes plaintext with 128 bit key... SI_STATUS_OK (cycles: 15403 time: 405 us)
+ Decrypting 256 bytes ciphertext with 128 bit key... SI_STATUS_OK (cycles: 15528 time: 408 us)
+ Comparing decrypted message with plain message... OK

. AES CMAC test
+ Generating 16 bytes CMAC on 256 bytes message with 128 bit key... SI_STATUS_OK (cycles: 15491 time: 407 us)

. HMAC test
+ Generating SHA1 HMAC on 256 bytes message with 128 bit key... SI_STATUS_OK (cycles: 14388 time: 378 us)
```

18. Now that the key is wrapped and stored in flash, you want to see that the program can use it without having the plaintext key anywhere in the application. Go back to `app_se_manager_block_cipher.c` and comment out lines 259 to 275 and lines 280 to 286.



```

257 }
258
259 // // YOUR KEY VALUE GOES HERE:
260 // static uint8_t user_key[16] =
261 // {
262 //     0x70, 0xF4, 0x82, 0x4E, 0x49, 0xBD, 0x97, 0xAB,
263 //     0x65, 0x65, 0x32, 0x22, 0xA0, 0x70, 0xB5, 0x16
264 // };
265 //
266 // sl_se_key_descriptor_t plaintext_desc = {
267 //     .type = SL_SE_KEY_TYPE_AES_128,
268 //     .flags = 0,
269 //     .storage.method = SL_SE_KEY_STORAGE_EXTERNAL_PLAINTEXT,
270 //     .storage.location.buffer.pointer = user_key,
271 //     .storage.location.buffer.size = 16
272 // };
273 //
274 // if (sl_se_import_key(&cmd ctx, &plaintext_desc, &symmetric_key_desc) != SL_STATUS_OK)
275 //     return SL_STATUS_FAIL;
276
277 // YOUR KEY ADDRESS GOES HERE:
278 unsigned int wrapped_key_address = 0x00080000;
279
280 // printf("Writing key into flash at 0x%08x...\n", wrapped_key_address);
281 //
282 // // Clear out the old wrapped key
283 // MSC_ErasePage((uint32_t*)wrapped_key_address);
284 //
285 // // Flash the new wrapped key
286 // MSC_WriteWord((uint32_t*)wrapped_key_address, symmetric_key_buf, sizeof(symmetric_key_buf));
287
288 // Update the key descriptor to point to the key in flash
289 symmetric_key_desc.storage.location.buffer.pointer = (uint8_t*)wrapped_key_address;
290

```

19. Now the application simply sets up the key descriptor to point to where you wrote the wrapped key in flash, without knowing the value of the key.
20. Repeat steps 11 to 15 to verify that the wrapped key can still be used. Note that if the flash is erased (by a commander device unlock command, for instance), this application will fail. It needs the wrapped key to be stored in flash by a previous process.

```

. Current symmetric key is a plaintext key.
+ Press SPACE to select a plaintext or wrapped or volatile key, press ENTER to next option.
+ Current symmetric key is a wrapped key.

. Current symmetric key length is 128-bit.
+ Press SPACE to select symmetric key length (128 or 192 or 256), press ENTER to next option.
+ Generating a 128-bit non-exportable symmetric wrapped key...
Keyspec: 0x09008010

. Current data length is 256 bytes.
+ Press SPACE to select data length (256 or 1024 or 4096), press ENTER to next option.

. Current Hash algorithm for HMAC is SHA1.
+ Press SPACE to select Hash algorithm (SHA1/224/256/384/512) for HMAC, press ENTER to run.

. AES ECB test
+ Encrypting 256 bytes plaintext with 128 bit key... SL_STATUS_OK (cycles: 13905 time: 365 us)
+ Decrypting 256 bytes ciphertext with 128 bit key... SL_STATUS_OK (cycles: 15018 time: 395 us)
+ Comparing decrypted message with plain message... OK

. AES CTR test
+ Encrypting 256 bytes plaintext with 128 bit key... SL_STATUS_OK (cycles: 15474 time: 407 us)
+ Decrypting 256 bytes ciphertext with 128 bit key... SL_STATUS_OK (cycles: 15532 time: 408 us)
+ Comparing decrypted message with plain message... OK

. AES CCM test
+ Encrypting 256 bytes plaintext with 128 bit key... SL_STATUS_OK (cycles: 16497 time: 434 us)
+ Decrypting 256 bytes ciphertext with 128 bit key... SL_STATUS_OK (cycles: 16856 time: 443 us)
+ Comparing decrypted message with plain message... OK

. AES GCM test
+ Encrypting 256 bytes plaintext with 128 bit key... SL_STATUS_OK (cycles: 16159 time: 425 us)
+ Decrypting 256 bytes ciphertext with 128 bit key... SL_STATUS_OK (cycles: 16464 time: 433 us)
+ Comparing decrypted message with plain message... OK

. AES CBC test
+ Encrypting 256 bytes plaintext with 128 bit key... SL_STATUS_OK (cycles: 15469 time: 407 us)
+ Decrypting 256 bytes ciphertext with 128 bit key... SL_STATUS_OK (cycles: 15106 time: 397 us)
+ Comparing decrypted message with plain message... OK

. AES CFB8 test
+ Encrypting 256 bytes plaintext with 128 bit key... SL_STATUS_OK (cycles: 3985565 time: 104 ms)
+ Decrypting 256 bytes ciphertext with 128 bit key... SL_STATUS_OK (cycles: 3978599 time: 104 ms)
+ Comparing decrypted message with plain message... OK

. AES CFB128 test
+ Encrypting 256 bytes plaintext with 128 bit key... SL_STATUS_OK (cycles: 15176 time: 399 us)
+ Decrypting 256 bytes ciphertext with 128 bit key... SL_STATUS_OK (cycles: 15550 time: 409 us)
+ Comparing decrypted message with plain message... OK

. AES CMAC test
+ Generating 16 bytes CMAC on 256 bytes message with 128 bit key... SL_STATUS_OK (cycles: 15370 time: 404 us)

. HMAC test
+ Generating SHA1 HMAC on 256 bytes message with 128 bit key... SL_STATUS_OK (cycles: 14034 time: 369 us)

```

## PKI Recommendations

# PKI Recommendations

This section outlines the recommended establishment, management, and security of Public Key Infrastructure (PKI) for business partners and customers of Silicon Labs. PKI plays a pivotal role in ensuring secure communication, data integrity, and authentication within our business ecosystem. This document sets forth recommended practices for the creation, management, and protection of secret keys and certificates by our partners and customers.

## Scope

These recommendations apply to all business partners and customers involved in transactions, communications, or collaborations with Silicon Labs and Silicon Labs Services that necessitate the use of PKI technology.

## Responsibilities

### Business Partners and Customers Responsibilities

- **Creation of Secret Keys and Certificates:** Business partners and customers should generate their secret keys securely and procure associated digital certificates from reputable Certificate Authorities (CAs) or generating their own digital certificates in accordance with the recommendations in this document. It is imperative that secret keys are generated using robust methods and are not shared with unauthorized parties.
- **Protection of Secret Keys:** Business partners and customers should implement comprehensive security measures to safeguard their secret keys against unauthorized access, loss, or theft. This encompasses encryption, access controls, regular key rotation where applicable, and employing secure storage methodologies. Backup and recovery of the secret keys is essential, and should be considered in case of disaster recovery needs. Keys should be stored in a well-managed and protected hardware security module (HSM).
- **Revocation and Renewal:** Business partners and customers should promptly revoke compromised or no longer required certificates and renew certificates before expiration to maintain ongoing security. Affected parties should have a way to determine status of certificate revocation and/or renewal through a hosted certificate revocation list (CRL) or online certificate status protocol (OCSP).
- **Ensure Internal Security:** In addition to the material security, business partners and customers should also maintain effective security around their organization and its operations, staff and contractors. This means maintaining endpoints and infrastructure in a secure way, such as patching operating systems and applications, hardening user applications and restricting administrative privileges. People in the organization and those managing the keys and certificates should be verified as trusted and secure.
- **Audits:** Business partners and customers should conduct regular audits of PKI and CA infrastructure and operations to confirm adherence to these recommendations and industry standards.

## Security Controls

NIST (National Institute of Standards and Technology) is an indispensable tool to navigate and strengthen cybersecurity systems and can be referenced as a guide for further recommendation on these Security Controls.

- **Access Controls:** Business partners and customers should implement access controls to limit access to secret keys and certificates to authorized personnel exclusively. This includes implementing role-based access control (RBAC) and conducting regular access reviews to ensure that only essential individuals have access to sensitive cryptographic materials.
- **Encryption:** All secret keys and sensitive certificate information should be encrypted during transmission and storage using robust cryptographic algorithms and protocols.
- **Key Management:** Business partners and customers should adopt robust key management practices, encompassing key generation, storage, rotation, and destruction, in accordance with industry best practices and standards.

- **Monitoring and Auditing:** Business partners and customers should implement monitoring and auditing mechanisms to monitor access to secret keys and certificates, detect unauthorized activities, and generate audit trails for compliance purposes.

## Revision History

This document will undergo periodic review and updates as necessary to reflect changes in technology, security requirements, or regulatory mandates.

## Contact Information

For inquiries or concerns regarding these recommendations, contact [certificateauthority@silabs.com](mailto:certificateauthority@silabs.com).