AN1218: Series 2 Secure Boot with RTSL

This application note describes the design of Secure Boot with RTSL (Root of Trust and Secure Loader) on Series 2 devices. It also provides examples of how to implement the Secure Boot process.

For more information on using the Gecko Bootloader with Series 2 devices, see the following:

- *UG103.6: Bootloader Fundamentals*
- *UG266: Silicon Labs Gecko Bootloader User's Guide for GSDK 3.2 and Lower*
- *UG489: Silicon Labs Gecko Bootloader User's Guide for GDSK 4.0 and Higher*

KEY POINTS

- Compares the Secure Boot process in Series 1 and Series 2 devices
- Describes the Series 2 Secure Boot with RTSL components and process
- Provides examples of configuring Series 2 devices for the Secure Boot process
- Recovers secure boot failure devices

Table of Contents

1. 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:

Table 1.1. SE Support for Secure Vault Levels

Level (1)	SE Support	Part (2)
Secure Vault High (SVH)	HSE only (HSE-SVH)	Refer to UG103.05 for details on supporting devices.
Secure Vault Mid (SVM)	HSE (HSE-SVM)	
	VSE (VSE-SVM)	
Secure Vault Base (SVB)	N/A	

Note:

1. The features of different Secure Vault levels can be found in Security.
2. UG103.05: 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.

1.1 User Assistance

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

Table 1.2. Key Security Documents

Document	Summary	Applicability
AN1190: Series 2 Secure Debug	How to lock and unlock Series 2 debug access, including background information about the SE	Secure Vault Mid and High
AN1218: Series 2 Secure Boot with RTSL (this document)	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
AN1247: Anti-Tamper Protection Configuration and Use	How to program, provision, and configure the anti-tamper module	Secure Vault High
AN1268: Authenticating Silicon Labs Devices using Device Certificates	How to authenticate a device using secure device certificates and signatures, at any time during the life of the product	Secure Vault High
AN1271: Secure Key Storage	How to securely “wrap” keys so they can be stored in non-volatile storage.	Secure Vault High

1.2 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.

Table 1.3. Reference Documents for Security Keys

Key Name	Customer Programmed	Purpose	Used in
Public Sign key (Sign Key Public)	Yes	Secure Boot binary authentication and/or OTA upgrade payload authentication	• AN1218 (primary) • AN1222
Public Command key (Command Key Public)	Yes	Secure Debug Unlock or Disable Tamper command authentication	• AN1190 (primary) • AN1222 • AN1247
OTA Decryption key (GBL Decryption key) aka AES-128 Key	Yes	Decrypting GBL payloads used for firmware upgrades	• AN1222 (primary) • UG266 • UG489
Attestation key aka Private Device Key	No	Device authentication for secure identity	• AN1268

1.3 SE Firmware

Silicon Labs strongly recommends installing the latest SE firmware on Series 2 devices to support the required security features. Refer to AN1222: Production Programming of Series 2 Devices for the procedure to upgrade the SE firmware and UG103.05: IoT Endpoint Security Fundamentals for the latest SE Firmware shipped with Series 2 devices and modules.

2. Secure Boot Process

2.1 Introduction

Secure Boot is a foundational component of platform security, and without it, other security aspects such as secure storage, secure transport, secure identity, and data confidentiality can often be subverted through the injection of malicious code.

Secure Boot works as a process by which each piece of firmware is validated for authenticity and integrity before it is allowed to run. Each authenticated module can also validate additional modules before executing them, forming a chain of trust. If any module fails its security check, it is not allowed to run, and program control will typically stall in the validating module. In most lightweight IoT systems, the behavior of a Secure Boot failure is to cause the device to stop working until an authentically signed image can be loaded onto it. Whereas this may seem extreme, it is a better outcome than a smart light bulb being repurposed to mine crypto-currency, or a smart speaker being repurposed as a surveillance device on the end user's private conversations.

The first link in the chain of trust is the root of trust. This is often the weakest link in the Secure Boot chain because the root of trust itself is not checked for authenticity or integrity. The security strength of the root of trust lies in its immutability. The strongest roots of trust have their firmware origin in ROM and use a Public Sign Key that is also located in ROM.

Wireless SoC Series 1 and Series 2 devices both use a two-stage boot design consisting of a non-upgradable first stage root of trust followed by an upgradable second stage. In Series 1 devices, the root of trust (also called the first-stage bootloader) is in flash rather than ROM, and the upgradable portion (the main bootloader) is checked for integrity using a CRC32 checksum but is not checked for authenticity using a Public Sign Key. In Series 2 devices, the root of trust is in ROM, and the upgradable portion is checked both for integrity and authenticity.

The Secure Boot with RTSL is implemented by Root code executed by the Hardware Secure Engine (HSE) or the Cortex-M33 operating in Root Mode (VSE). For more information about SE, see section "*Secure Engine Subsystem*" in AN1190: Series 2 Secure Debug.

Silicon Labs provides Custom Part Manufacturing Service (CPMS) to customize the users' security features and settings.

This application note uses the following abbreviations:

- FSB - First Stage Bootloader
- SSB - Second Stage Bootloader
- GBL - Gecko Bootloader
- RTSL - Root of Trust and Secure Loader
- HSM - Hardware Security Module
- OTP - One-Time Programmable
- WSTK - Wireless Starter Kit
- GSDK - Gecko Software Development Kit. For more info refer GSDK.
- ECDSA-P256-SHA256 - Elliptic Curve Digital Signature Algorithm aka ECSDA using a P-256 curve and a SHA256 hash
- PEM (`.pem`) - Privacy Enhanced Mail
- DER (`.der`) - Distinguished Encoding Rules

2.2 ECDSA-P256-SHA256 Secure Boot in Series 1 Devices

The Secure Boot process for Series 1 (SVB) devices originates in flash, typically with the execution of the first stage of GBL. The first stage of GBL checks to see if an upgrade is pending for the second stage of GBL. If so, it processes the upgrade of the second stage and then executes it. Otherwise, it just executes the second stage. If Secure Boot is enabled, the second stage of GBL checks the integrity and authenticity of the application image before executing it. If the integrity check fails, program control remains in the SSB. The following figure illustrates the Secure Boot process on Series 1 devices.

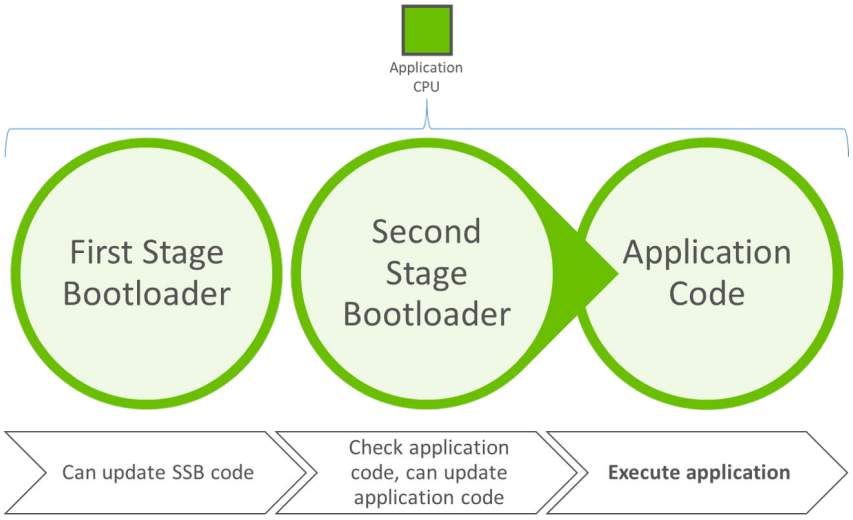


Figure 2.1. Series 1 ECDSA-P256-SHA256 Secure Boot Process

See UG266/UG489 for more information to generate and download signed firmware images using Simplicity Commander.

2.3 ECDSA-P256-SHA256 Secure Boot in Series 2 Devices

For Series 2 devices, the Secure Engine (SE) implements the FSB to authenticate and upgrade the SSB. The GBL implements the SSB (aka Main Bootloader in UG266/UG489) to authenticate and upgrade the application firmware.

Refer to the "*Gecko Bootloader Security Features*" section in UG266/UG489 and ECDSA-P256-SHA256 Secure Boot example for more information about the ECDSA-P256-SHA256 secure boot process in Series 2 devices.

Note: It is possible to have a 2-stage design to skip the SSB between FSB and application. However, the application cannot be upgraded if discarding the SSB, and this application note assumes the SSB is present.

2.3.1 HSE

In HSE-SVM and HSE-SVH devices, the Secure Boot process originates in ROM contained in the security co-processor (HSE). The following figures illustrate the Secure Boot process and flow on Series 2 HSE devices.

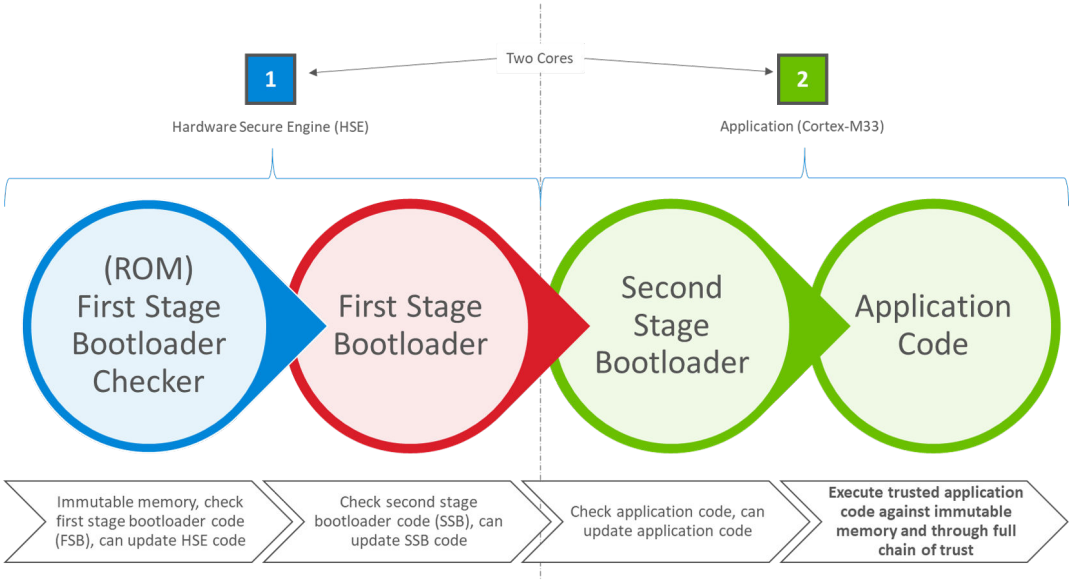
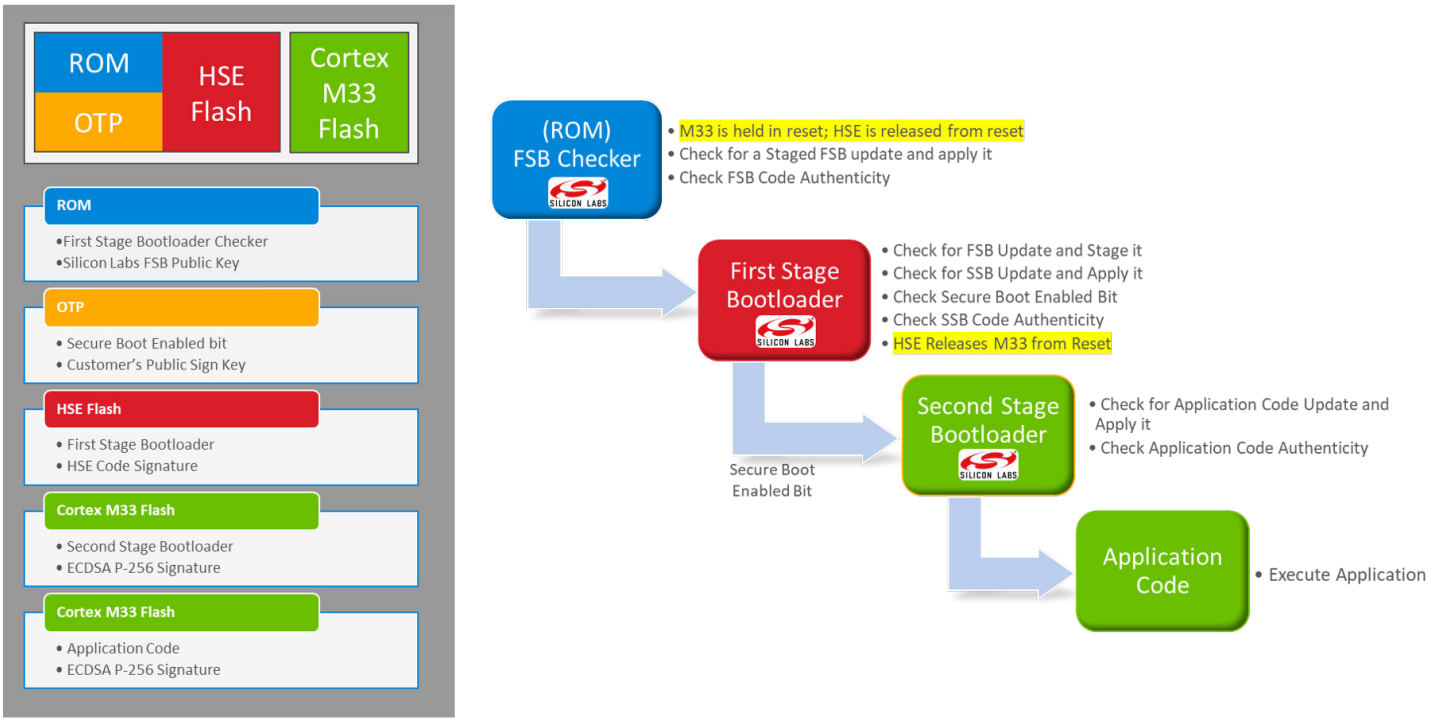


Figure 2.2. Series 2 HSE ECDSA-P256-SHA256 Secure Boot Process



3

Figure 2.3. Series 2 HSE ECDSA-P256-SHA256 Secure Boot Flow

2.3.2 VSE

In VSE-SVM devices, the host MCU (Cortex-M33) assumes an elevated security state out of reset and securely boots itself from code that originates in ROM. The following figures illustrate the Secure Boot process and flow on Series 2 VSE devices.

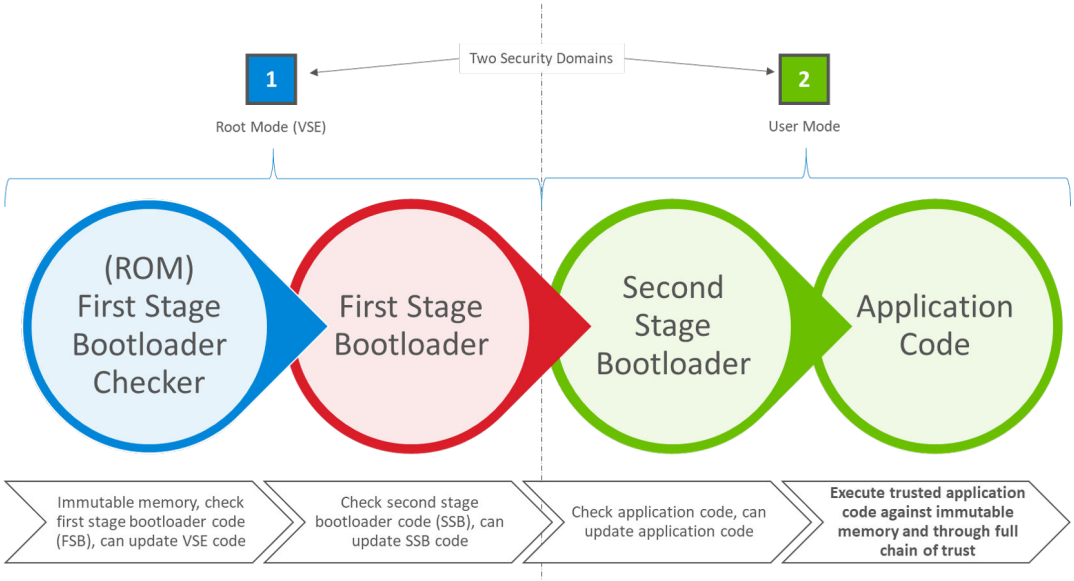


Figure 2.4. Series 2 VSE ECDSA-P256-SHA256 Secure Boot Process

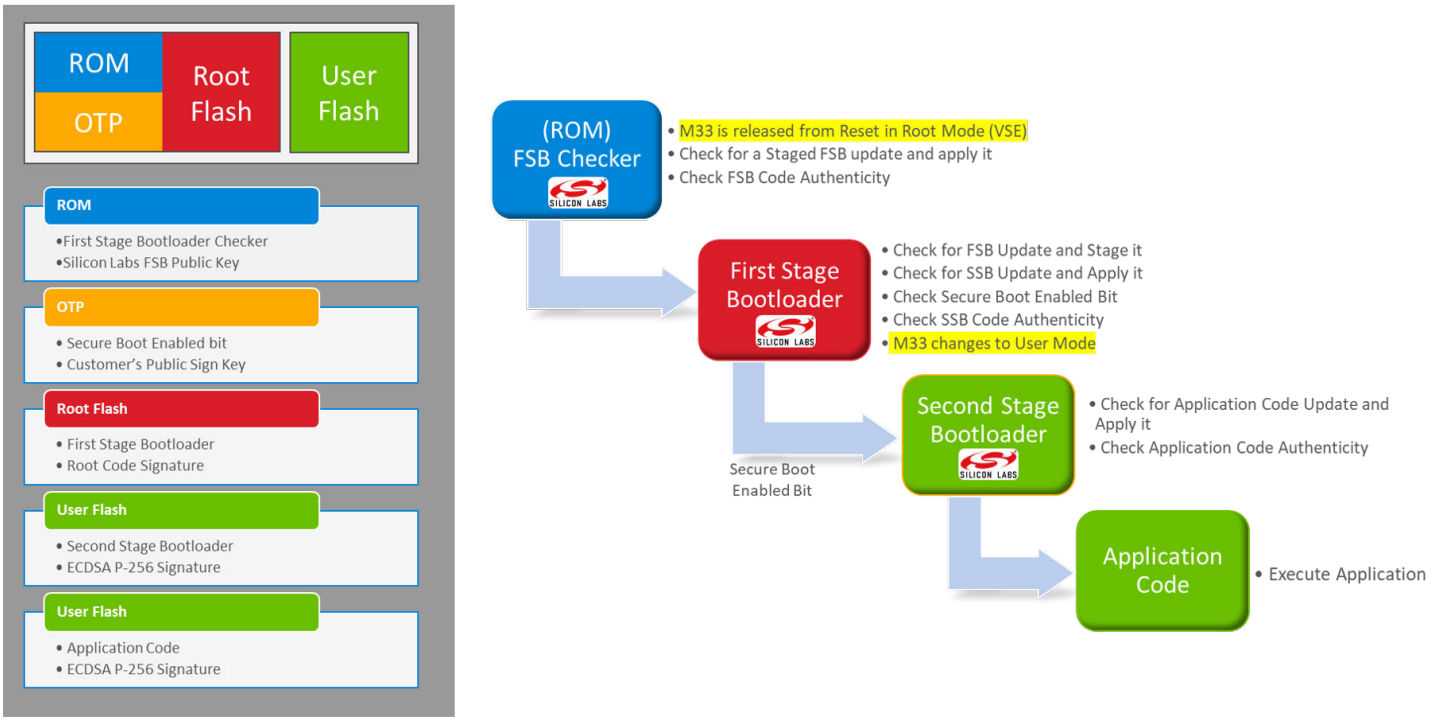


Figure 2.5. Series 2 VSE ECDSA-P256-SHA256 Secure Boot Flow

2.4 Certificate-based Secure Boot in Series 2 Devices

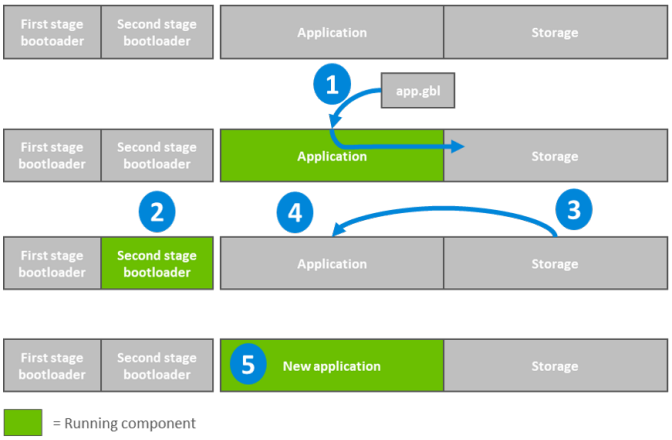
Refer to the "*Gecko Bootloader Security Features*" section in UG266: Silicon Labs Gecko Bootloader User's Guide for GSDK 3.2 and Lower/ UG489: Silicon Labs Gecko Bootloader User's Guide for GSDK 4.0 and Higher and Certificate-Based Secure Boot example for details about the certificate-based Secure Boot process in Series 2 devices.

The certificate-based Secure Boot uses key delegation to minimize the exposure of the Private Sign Key, reducing the chance to revoke the Public Sign Key.

If the certificate's private key is leaked, all devices that have been programmed with that certificate are at risk until they can be updated with an image containing a certificate with a higher version.

2.5 Secure Loader

In Series 2 devices, the Secure Loader is firmware pre-loaded into the chip. Silicon Labs maintains the Secure Loader and deploys through secure upgrade packages. It is the functional equivalent of the first-stage GBL on Series 1 devices (refer to UG266: Silicon Labs Gecko Bootloader User's Guide for GSDK 3.2 and Lower/ UG489: Silicon Labs Gecko Bootloader User's Guide for GSDK 4.0 and Higher for more information). The Secure Loader validates the authenticity and integrity of a staged image before performing an upgrade operation. The Secure Loader requires the staged image to reside on-chip and the staged image must not overlap with the target destination address range. Firmware images that originate from off-chip, either off-chip storage, external NCP host interface, or through an OTA update procedure are expected to be staged either by the application or by SSB before calling the Secure Loader to perform the upgrade.



1. The existing application downloads the upgrade image onto the storage medium of the device (for example, the app.gbl file through OTA).
2. The device reboots into the second stage bootloader, and the bootloader enters firmware upgrade mode.
3. The secure loader in the second stage bootloader verifies the integrity and authenticity of the staged upgrade image in storage, prior to performing the upgrade.
4. The second stage bootloader applies the application upgrade from the upgrade image.
5. The device boots into the application. Application upgrade is complete.

Figure 2.6. Series 2 Secure Loader Example

2.6 Secure Boot Time

Secure boot extends the recovery time from all sources of device reset. The duration of each authentication operation depends on the factors below:

- Computes the SHA-256 digest (32 bytes) of the associated image, which is proportional to the size of the firmware image.
- Verifies the ECDSA-P256 signature of the SHA-256 digest above, which is independent of image size.
- The clock frequency of the Crypto Engine, which is the HSE or CRYPTOACC in VSE devices.

Table 2.1. Authentication Duration

Authentication	Enable/Disable	Duration
FSB code	Enable (cannot disable)	FSB code size dependent
SSB code	Disable (by default)	SSB code size and SE firmware version dependent
Application code	Disable (by default)	Application code size and SSB firmware (GBL) version dependent

Note:

- It will extend the boot time for certificates authentication if using Certificate-Based Secure Boot.
- Refer to device-specific datasheets (like EFR32MG21B) for data about the boot timing of Series 2 devices.
- Refer to 2.7 Secure Boot Configuration on how to enable the SSB and application code authentication.

2.7 Secure Boot Configuration

The following sections describe how to configure the Secure Boot of the SSB (GBL) and application firmware.

2.7.1 SSB

In Series 2 devices, the immutable OTP memory stores the Public Sign Key and Secure Boot Enable flag. The user cannot change its respective value once either is programmed. Once the Public Sign Key is provisioned, it remains provisioned to that key value for the life of the device. Once Secure Boot is enabled, it remains enabled for the life of the device. Both of these assignment operations are **IRREVOCABLE**.

The Public Sign Key used for Series 2 devices is the public portion of an ECDSA key pair over the NIST prime curve P-256. The Public Sign Key is a customer key and is typically provisioned during the initial product manufacturing and device programming phase. It is common for all products that share the same firmware image to be loaded with the same Public Sign Key. The key loaded into the device is a public key and has no confidentiality requirements. The private key associated with that public key, which will be used to sign firmware images or certificates, should be tightly held, ideally secured in the HSM or equivalent key storage instrument.

The user can use Simplicity Commander, SE Manager, or Simplicity Studio to program the Public Sign Key and configure the SSB Secure Boot in SE OTP.

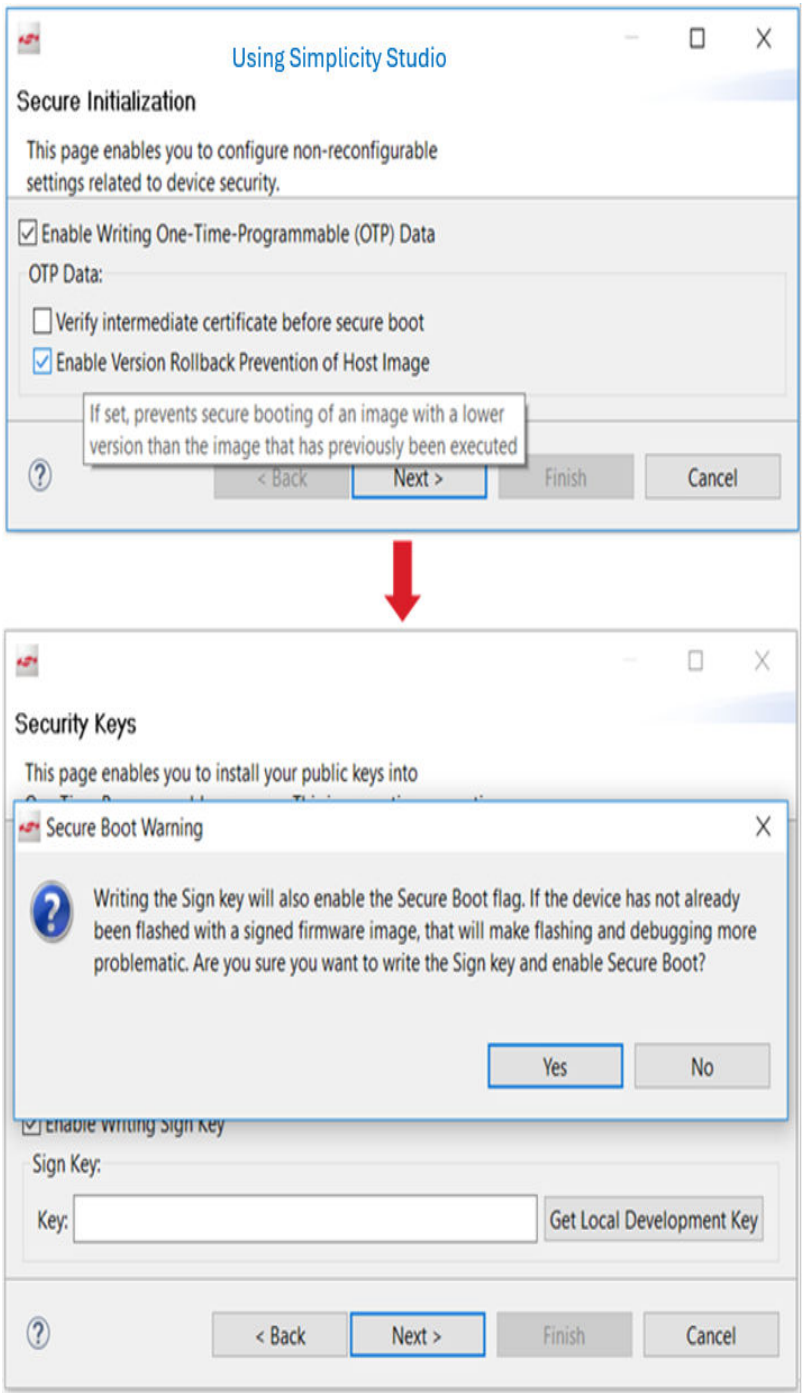


Figure 2.7. Configuration of SSB using Simplicity Studio

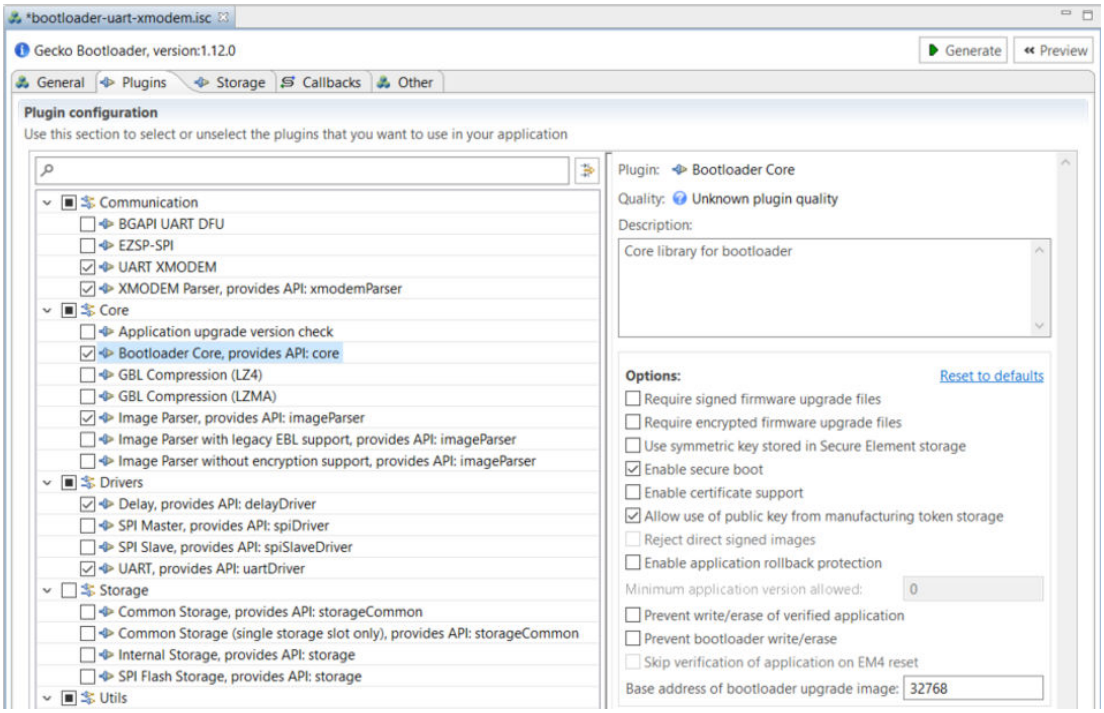
Figure 2.8. Configuration of SSB using Simplicity Commander

2.7.2 Application Firmware

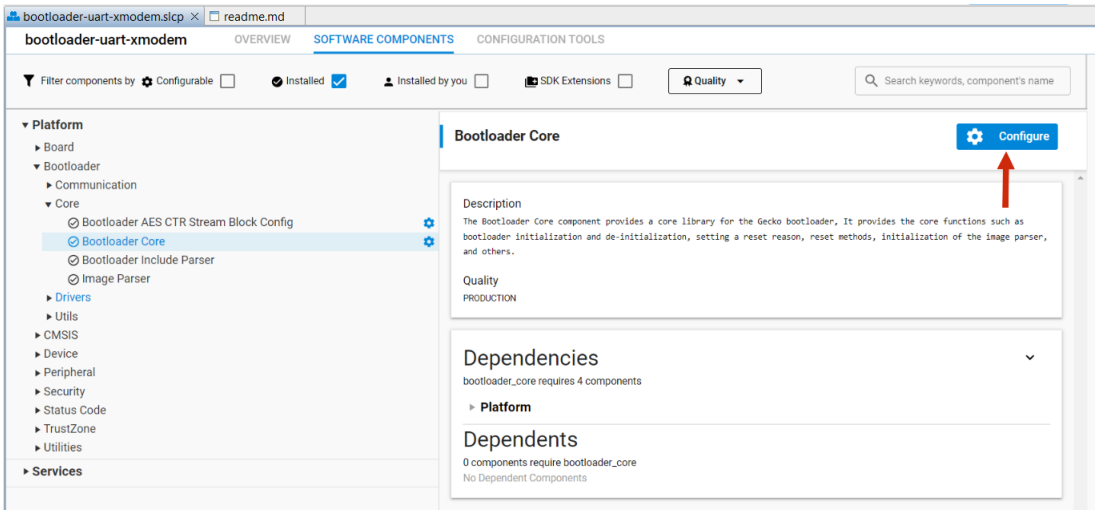
The user can use the AppBuilder or Bootloader-core software component in the GBL project to configure the security options of the application firmware.

The user can reconfigure the Secure Boot configuration of the application firmware by upgrading the GBL with the new custom settings.

Using AppBuilder



Using Bootloader Core Component



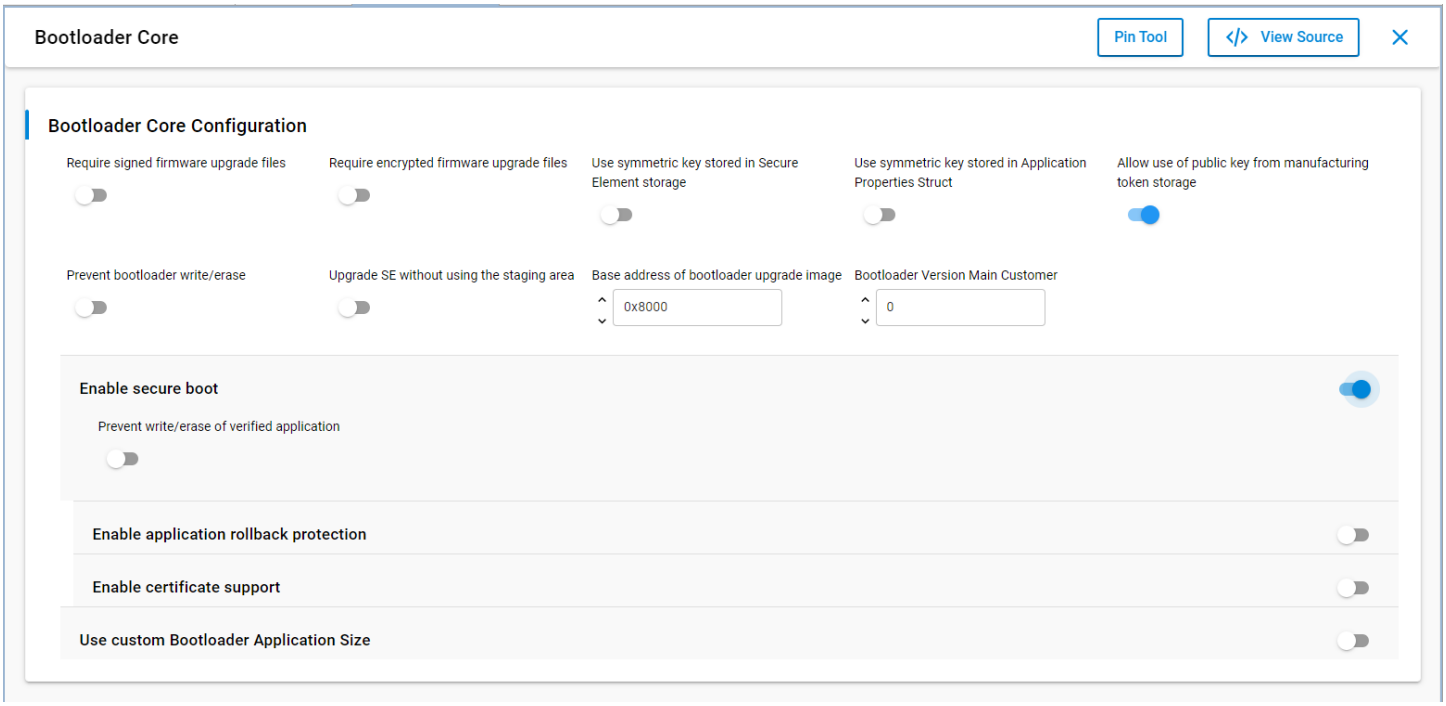


Figure 2.9. Security Options of Application Firmware

3. Examples

3.1 Overview

The following table describes the examples for Series 2 Secure Boot.

Table 3.1. Series 2 Secure Boot Examples

Example	Device (Radio Board)	Minimum SE Firmware	Tool
Provision Public Sign Key and Secure Boot Enabling	EFR32MG21A010F1024IM32 (BRD4181A)	Version 1.2.9	Simplicity Commander
	EFR32MG21A010F1024IM32 (BRD4181A)	Version 1.2.9	SE Manager
	EFR32MG21A010F1024IM32 (BRD4181A)	Version 1.2.9	Simplicity Studio 5
Provision GBL Decryption Key	EFR32MG21A010F1024IM32 (BRD4181A)	Version 1.2.9	Simplicity Commander
	EFR32MG21A010F1024IM32 (BRD4181A)	Version 1.2.9	SE Manager
Signing for ECDSA-P256-SHA256 Secure Boot	EFR32MG21A010F1024IM32 (BRD4181A)	Version 1.2.9	Simplicity Commander
Signing for Certificate-Based Secure Boot	EFR32MG21A010F1024IM32 (BRD4181A)	Version 1.2.9	Simplicity Commander
Generate a GBL Upgrade Image File	EFR32MG21A010F1024IM32 (BRD4181A)	Version 1.2.9	Simplicity Commander
Upgrade to Certificate-Based Secure Boot	EFR32MG21A010F1024IM32 (BRD4181A)	Version 1.2.9	Simplicity Commander
Certificate Revocation	EFR32MG21A010F1024IM32 (BRD4181A)	Version 1.2.9	Simplicity Commander
Upgrade to Secure Boot with RTSL	EFR32MG21A010F1024IM32 (BRD4181A)	Version 1.2.9	SE Manager & Simplicity Commander
Recover Devices when Secure Boot Fails	EFR32MG21A010F1024IM32 (BRD4181A)	Version 1.2.9	Simplicity Commander

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

3.1.1 Using Simplicity Commander

1. This application note uses Simplicity Commander v1.12.0. 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 MCU Programming options.

```
commander --version
```

```
Simplicity Commander 1v12p0b1057

JLink DLL version: 7.52d
Qt 5.12.10 Copyright (C) 2017 The Qt Company Ltd.
EMDLL Version: 0v17p19b0
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 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.

3.1.2 Using an External Tool

The Secure Boot examples use the **OpenSSL** to sign the image files and certificates. The Windows version of OpenSSL can be downloaded from https://slproweb.com/products/Win32OpenSSL.html. This application note uses OpenSSL Version 1.1.1t (Win64).

```
openssl version
```

```
OpenSSL 1.1.1t  22 Sep 2023
```

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.

3.1.3 Using a Platform Example

Simplicity Studio 5 includes the SE Manager platform example for key provisioning and Secure Boot enabling. This application note uses platform example of GSDK v3.2.3. The console output may be different on other versions of the 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.

3.1.4 Generate Key and Signing

This section describes how to generate a key to sign an image file or certificate for Secure Boot.

Using Simplicity Commander

1. Run the `util genkey` command to generate the ECDSA-P256 Sign Key pair (`sign_key.pem` and `sign_pubkey.pem`) and Public Sign Key token file (`sign_pubkey.txt`). The Simplicity Commander can program the Public Sign Key in token file (`sign_pubkey.txt`) to the top page of the main flash.

```
commander util genkey --type ecc-p256 --privkey sign_key.pem --pubkey sign_pubkey.pem
--tokenfile sign_pubkey.txt
```

```
Generating ECC P256 key pair...
Writing private key file in PEM format to sign_key.pem
Writing public key file in PEM format to sign_pubkey.pem
Writing EC tokens to sign_pubkey.txt...
DONE
```

   Note: The same procedure can apply to generate the bootloader certificate and application certificate key pairs for Certificate-Based Secure Boot.

2. Use the `convert` command with the Private Key (like `sign_key.pem`) from step 1 to sign an image file or certificate. Refer to 3.4.3 Signing for ECDSA-P256-SHA256 Secure Boot and 3.4.4 Signing for Certificate-Based Secure Boot for more information about the Simplicity Commander signing process.

Using an HSM and Simplicity Commander

1. The user can use HSM to generate the ECDSA-P256 Sign Key pair. The Private Sign Key is securely held in HSM and the Public Sign Key can be exported in a specific format (like `sign_pubkey.pem`).

   Note: The same procedure can apply to generate the bootloader certificate and application certificate key pairs for Certificate-Based Secure Boot.

2. Use the `util keytotoken` command to convert the Public Sign Key from step 1 to token format (`sign_pubkey.txt`). The Simplicity Commander can program the Public Sign Key in token file (`sign_pubkey.txt`) to the top page of the main flash.

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

```
Writing EC tokens to sign_pubkey.txt...
DONE
```

3. Use the `convert` command with `--extsign` option to prepare an unsigned image or certificate for HSM.
4. Use the Private Key from step 1 to generate a signature for the unsigned image or certificate from step 3.
5. Use the `convert` command with the signature from step 4 to generate a signed image or certificate. Refer to 3.4.3 Signing for ECDSA-P256-SHA256 Secure Boot and 3.4.4 Signing for Certificate-Based Secure Boot for more information about the HSM and Simplicity Commander signing process.

   Note: The Simplicity Commander v1.11.0 or above supports signature in DER format. The older version of Simplicity Commander can only handle signatures in Raw format.

3.2 Provision Public Sign Key and Secure Boot Enabling

The Public Sign Key in SE OTP is used to verify the host image signature or certificate during Secure Boot. The user should provision this key before setting the Secure Boot enabled flag in SE OTP. On HSE-SVH devices, the user requires to provision the anti-tamper protection configuration with Secure Boot settings.

If the user sets the SECURE_BOOT_ANTI_ROLLBACK option with Secure Boot, the SE will store the version counter (4 bytes) for anti-rollback of GBL (SSB) to SE flash and check the bootloader version during an upgrade and on every boot. The version counter will not roll to 0 if it reaches the maximum value (bootloader cannot upgrade anymore). The anti-rollback does not prevent flashing an older signed GBL hex image to the device.

The following table describes the anti-rollback protection on signed GBL when `SECURE_BOOT_ANTI_ROLLBACK` is enabled or disabled.

- The GBL handles the anti-rollback protection when upgrading the GBL through the GBL upgrade image file (`.gbl`).
- The SE handles the anti-rollback protection (if `SECURE_BOOT_ANTI_ROLLBACK` enabled) when booting the GBL.

Table 3.2. Anti-Rollback

Action	SECURE_BOOT_ANTI_ROLLBACK Disable	SECURE_BOOT_ANTI_ROLLBACK Enable
Use a GBL upgrade image file	Reject upgrade if an equal or lower GBL version is detected.	Reject upgrade if an equal or lower GBL version is detected.
Flash and boot a GBL hex image	Accept to flash and boot regardless of the GBL version.	Accept to flash regardless of the GBL version. But it cannot boot if a lower GBL version is detected.

Note: It needs to execute a mass erase (`commander device masserase` or `commander security erasedevice` then reset) before flashing a GBL hex image (`.s37`) to the device if SECURE_BOOT_PAGE_LOCK_NARROW or SECURE_BOOT_PAGE_LOCK_FULL option in SE OTP is enabled.

For simplicity, the Secure Boot examples in this application note do not enable the following options for Secure Boot.

- `SECURE_BOOT_PAGE_LOCK_NARROW`
- `SECURE_BOOT_PAGE_LOCK_FULL`

3.2.1 Simplicity Commander

The following procedures assume the required files are in the same folder.

1. Follow the procedures in 3.1.4 Generate Key and Signing to generate the ECDSA-P256 Sign Key pair (`sign_key.pem` and `sign_pubkey.pem`) and Public Sign Key token file (`sign_pubkey.txt`).
2. Run the `security writekey` command to provision the Public Sign Key (`sign_pubkey.pem`). The Public Sign Key cannot be changed once written.

```
commander security writekey --sign sign_pubkey.pem --device EFR32MG21A010F1024 --serialno 440048205
```

```
Device has serial number 00000000000000014b457fffe045b21

================================================================================
Please look through any warnings before proceeding.
THIS IS A ONE-TIME command, all code to be run on the device must be signed by this key.
Type 'continue' and hit enter to proceed or Ctrl-C to abort:
================================================================================
continue
DONE
```

3. Run the `security readkey` command to verify the Public Sign Key with the Public Sign Key in the token file (`sign_pubkey.txt`).

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

```
C4AF4AC69AAB9512DB50F7A26AE5B4801183D85417E729A56DA974F4E08A562C
DE6019DEA9411332DC1A743372D170B436238A34597C410EA177024DE20FC819
DONE
```

4. For Series 2 VSE devices (like EFR32MG22C224F512IM40), run the `flash` command to program the Public Sign Key in the token file (`sign_pubkey.txt`) to the top page of the main flash for ECDSA-P256-SHA256 Secure Boot. It is optional on Series 2 HSE devices.

```
commander flash --tokengroup znet --tokenfile sign_pubkey.txt --device EFR32MG22C224F512IM40 --serialno 440048205
```

```
Writing 8192 bytes starting at address 0x0007e000
Comparing range 0x0007E000 - 0x0007FFFF (8 KiB)
Programming range 0x0007E000 - 0x0007FFFF (8 KiB)
DONE
```

Note: The MCU Series 2 devices (like EFM32PG22C200F512IM40) require Simplicity Commander Version 1.12.2 or above to support the `flash --tokengroup znet` command.

5. Run the `security genconfig` command to generate the `user_configuration.json` file for secure boot.

```
commander security genconfig --nostore --outfile user_configuration.json --device EFR32MG21A010F1024
--serialno 440048205
```

```
DONE
```

Table 3.3. Security Configuration Description

Name	Description
SECURE_BOOT_ENABLE	If set, verifies the host image on the Cortex-M33 before releasing the Cortex-M33 from reset.
SECURE_BOOT_VERIFY_CERTIFICATE	If set, requires certificate-based signing of the host image.
SECURE_BOOT_ANTI_ROLLBACK	If set, prevents secure upgrading to a host image with a lower version than the image that is currently stored in flash.

Name	Description
SECURE_BOOT_PAGE_LOCK_NARROW	If set, locks flash pages that have been validated by the Secure Boot process to prevent re-flashing by other means than through the SE. Write/erase locks pages from 0 through the page where the Secure Boot host image signature is located, not including the last page if the signature is not on a page boundary.
SECURE_BOOT_PAGE_LOCK_FULL	If set, locks flash pages that have been validated by the Secure Boot process to prevent re-flashing by other means than through the SE. Write/erase locks pages from 0 through the page where the Secure Boot host image signature is located, including the last page if the signature is not on a page boundary.

Note: The host image is the firmware in the device's flash starting address. It is usually the GBL.

6. Use a text editor to modify the default secure boot settings to the desired configurations used in this application note.

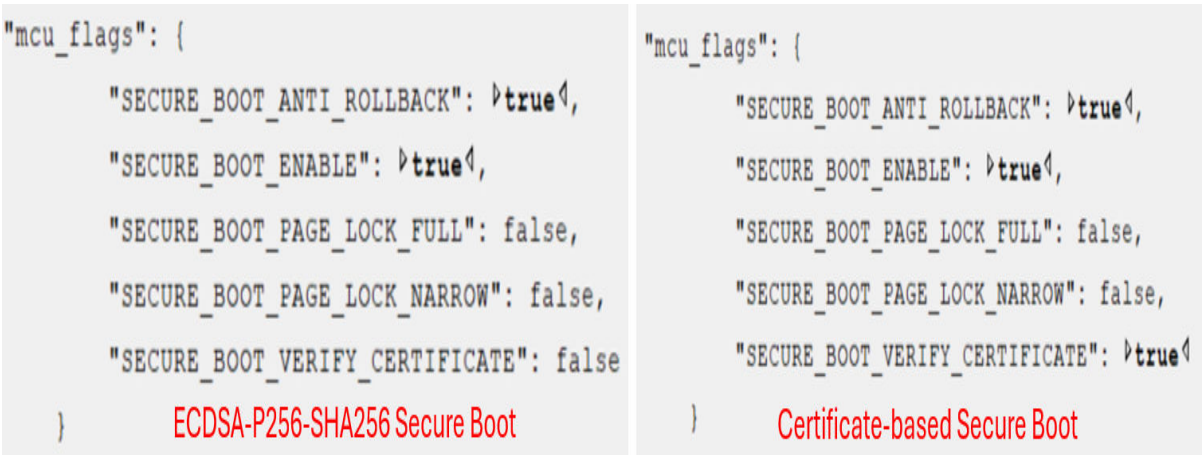


Figure 3.1. Secure boot settings

Note: If `SECURE_BOOT_ENABLE` is false, the SE will ignore the other four options regardless they are true or false. The EFR32xG23 and future Series 2 devices do not allow this setting to program to the SE OTP.

7. Follow the procedure in 3.4.3 Signing for ECDSA-P256-SHA256 Secure Boot to flash a bootloader image for signature based secure boot.

   OR
   Follow the procedure in in 3.4.4 Signing for Certificate-Based Secure Boot to flash a bootloader image for certificate based secure boot.
8. Run the `security writeconfig` command to program the secure boot configuration to the SE OTP. The user can execute this command once per device.

```
commander security writeconfig --configfile user_configuration.json --device EFR32MG21A010F1024
--serialno 440048205
```

```
================================================================================

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
```

9. Run the `security readconfig` command to check the secure boot configuration of the device.

```
commander security readconfig --serialno 440048205
```

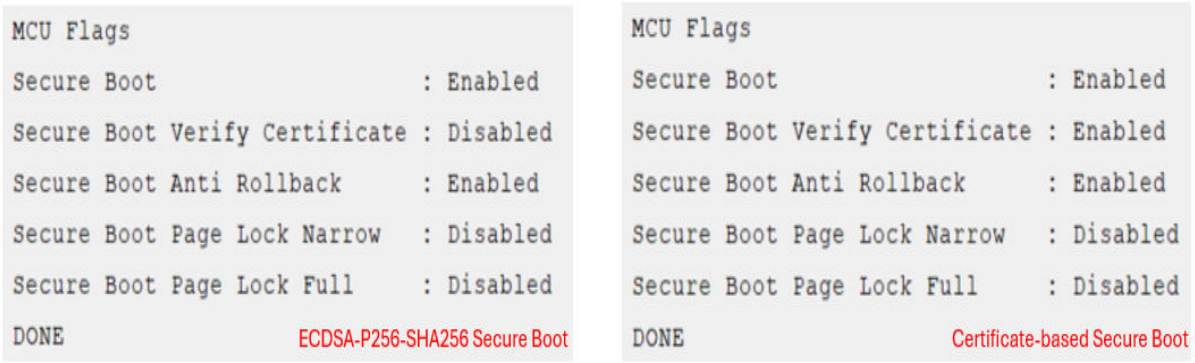


Figure 3.2. Secure Boot configuration

3.2.2 SE Manager Key Provisioning Platform Example

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

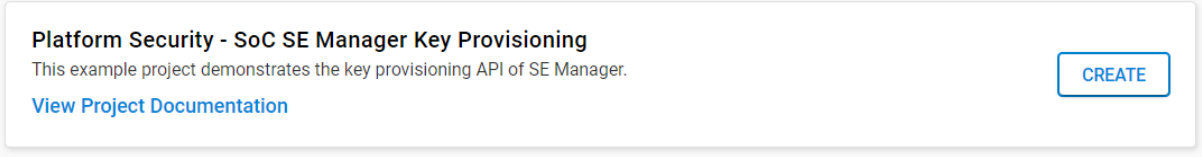


Figure 3.3. Key Provisioning Sample Application

1. Modify the default Public Sign Key in `public_sign_key[PUB_KEY_SIZE]` array in `app_process.c` to the desired values.

```
/// Public sign key
SL_ALIGN(4) static const uint8_t public_sign_key[PUB_KEY_SIZE] = {
  0xc4, 0xaf, 0x4a, 0xc6, 0x9a, 0xab, 0x95, 0x12,
  0xdb, 0x50, 0xf7, 0xa2, 0x6a, 0xe5, 0xb4, 0x80,
  0x11, 0x83, 0xd8, 0x54, 0x17, 0xe7, 0x29, 0xa5,
  0x6d, 0xa9, 0x74, 0xf4, 0xe0, 0x8a, 0x56, 0x2c,
  0xde, 0x60, 0x19, 0xde, 0xa9, 0x41, 0x13, 0x32,
  0xdc, 0x1a, 0x74, 0x33, 0x72, 0xd1, 0x70, 0xb4,
  0x36, 0x23, 0x8a, 0x34, 0x59, 0x7c, 0x41, 0x0e,
  0xa1, 0x77, 0x02, 0x4d, 0xe2, 0x0f, 0xc8, 0x19
};
```

2. Modify the default secure boot settings in `init_se_otp_conf()` function in `app_se_manager_key_provisioning.c` to the desired configuration.

```
  // Overwrite secure boot options in SL_SE_OTP_INIT_DEFAULT if necessary.
  otp_init.enable_secure_boot = true;
  otp_init.verify_secure_boot_certificate = false;
  otp_init.enable_anti_rollback = true;
  otp_init.secure_boot_page_lock_narrow = false;
  otp_init.secure_boot_page_lock_full = false;
```

Note: If `enable_secure_boot` is false, the SE will ignore the other four options regardless of whether they are true or false. The EFR32xG23 and future Series 2 devices do not allow this setting to program to the SE OTP.

3. Follow the procedures in 3.4.2 Generate an Unsigned Application Image to generate the unsigned application image if the GBL is present in the device.
4. Build the project and run the application. Follow the procedures in 3.4.3 Signing for ECDSA-P256-SHA256 Secure Boot or 3.4.4 Signing for Certificate-Based Secure Boot if a signed application image is required.
5. Then press `SPACE` to skip the programming of the AES-128 key (HSE devices only). Optional to press `ENTER` to program the hard-coded GBL Decryption Key to HSE OTP.

```
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.
```

6. Press `ENTER` to program the hard-coded Public Sign Key to SE OTP.

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

7. Press ENTER to confirm the operation. The SE returns SL_STATUS_INVALID_PARAMETER if the Public Sign Key is present in SE OTP.

```
. Initialize public sign key... SL_STATUS_OK (cycles: 56052 time: 1475 us)

. Get public sign key... SL_STATUS_OK (cycles: 8450 time: 222 us)
+ The public sign key (64 bytes):
  C4 AF 4A C6 9A AB 95 12 DB 50 F7 A2 6A E5 B4 80
  11 83 D8 54 17 E7 29 A5 6D A9 74 F4 E0 8A 56 2C
  DE 60 19 DE A9 41 13 32 DC 1A 74 33 72 D1 70 B4
  36 23 8A 34 59 7C 41 0E A1 77 02 4D E2 0F C8 19

. Press ENTER to program public command key in SE OTP or press SPACE to skip.
```

```
. Initialize public sign key... SL_STATUS_INVALID_PARAMETER (cycles: 4375 time: 115 us)

. Get public sign key... SL_STATUS_OK (cycles: 8435 time: 221 us)
+ The public sign key (64 bytes):
  C4 AF 4A C6 9A AB 95 12 DB 50 F7 A2 6A E5 B4 80
  11 83 D8 54 17 E7 29 A5 6D A9 74 F4 E0 8A 56 2C
  DE 60 19 DE A9 41 13 32 DC 1A 74 33 72 D1 70 B4
  36 23 8A 34 59 7C 41 0E A1 77 02 4D E2 0F C8 19

. Press ENTER to program public command key in SE OTP or press SPACE to skip.
```

8. Press SPACE to skip the programming of the Public Command Key. Optional to press ENTER to program the hard-coded Public Command Key to SE OTP.

```
. Get public command key... SL_STATUS_FAIL (cycles: 4126 time: 108 us)

. Press ENTER to initialize SE OTP for secure boot configuration or press SPACE to skip.
```

9. Press ENTER to program the secure boot configuration.

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

10. Press ENTER to confirm the operation. The SE returns SL_STATUS_COMMAND_IS_INVALID if an invalid setting from step 2 or the secure boot configuration has been programmed in SE OTP.

```
. Initialize SE OTP... SL_STATUS_OK (cycles: 267256 time: 7033 us)
+ Read SE OTP configuration... SL_STATUS_OK (cycles: 6865 time: 180 us)
+ Secure boot                   : Enabled
+ Secure boot verify certificate : Disabled
+ Secure boot anti-rollback      : Enabled
+ Secure boot page lock narrow   : Disabled
+ Secure boot page lock full     : Disabled

. SE manager deinitialization... SL_STATUS_OK (cycles: 5 time: 0 us)
```

```
. Initialize SE OTP... SL_STATUS_COMMAND_IS_INVALID (cycles: 3989 time: 104 us)

. SE manager deinitialization... SL_STATUS_OK (cycles: 5 time: 0 us)
```

3.2.3 Simplicity Studio

The security operations are performed in the Security Settings of Simplicity Studio. This application note uses Simplicity Studio v5.2.3.1. 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.

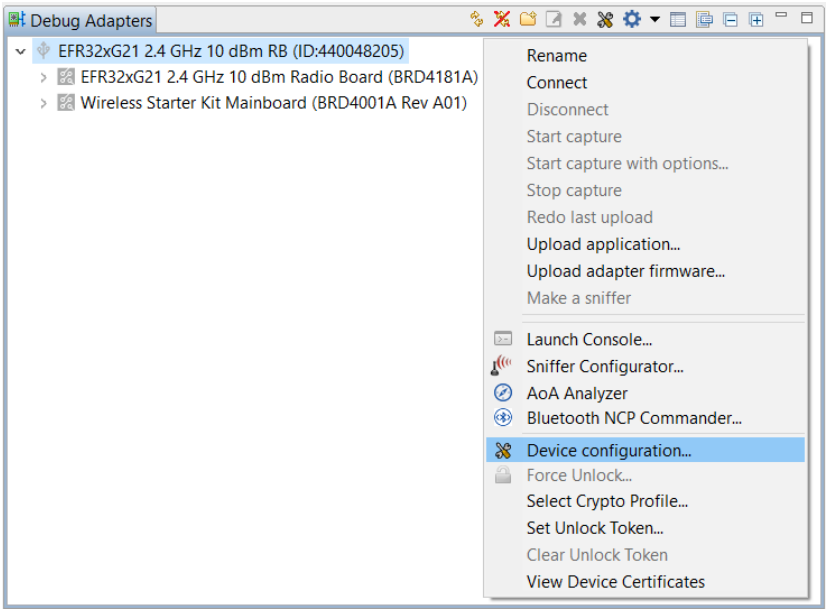


Figure 3.4. Debug Adapters Context Menu

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.

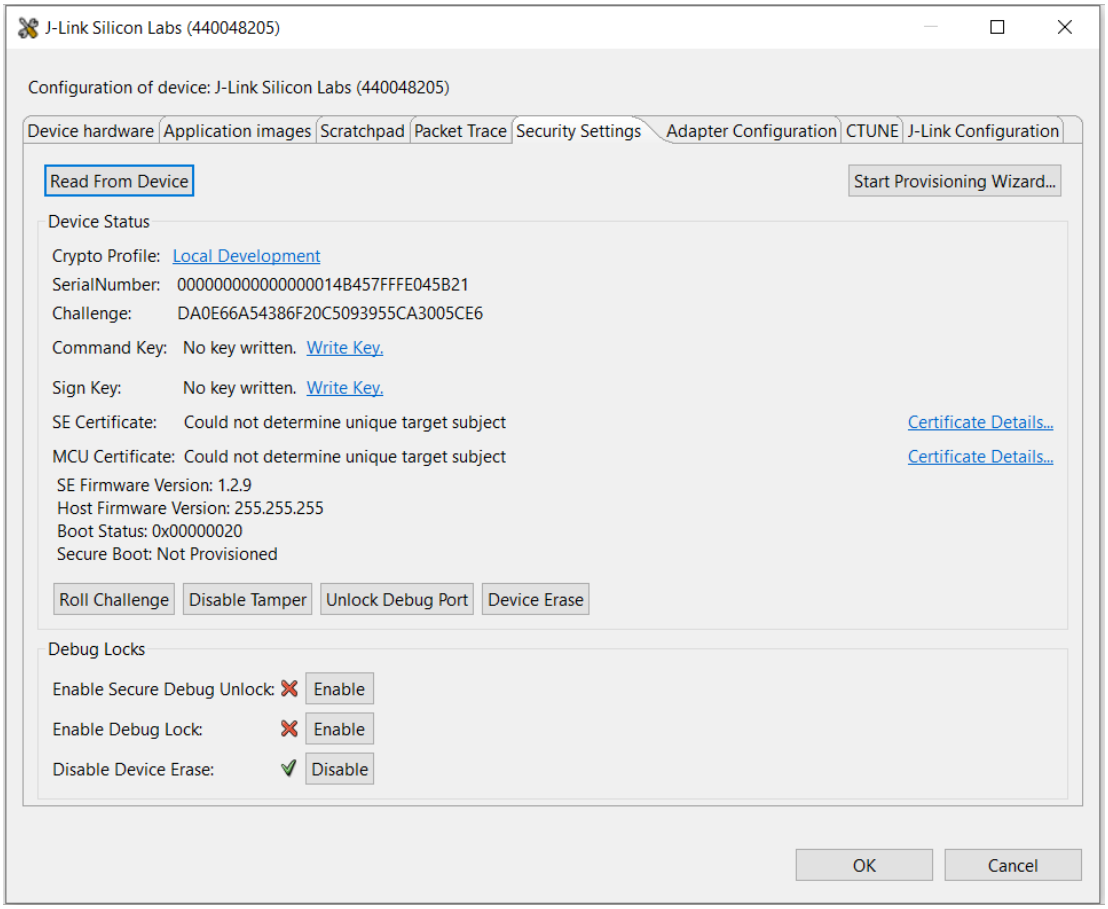


Figure 3.5. Configuration on Selected Device

3. Click [**Start Provisioning Wizard…**] in the upper right corner to display the **Secure Initialization** dialog box. Checking the **Enable Version Rollback Prevention of Host Image** option is recommended. The **Verify intermediate certificate before secure boot** option is for Certificate-based Secure Boot.

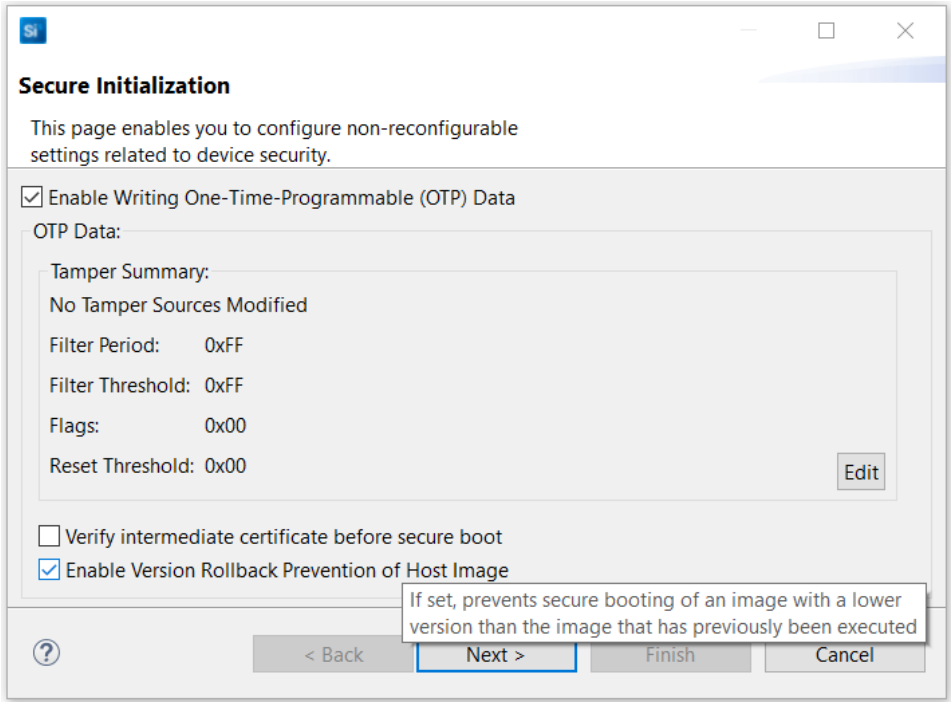


Figure 3.6. Secure Initialization Dialog Box

Note: The SECURE_BOOT_PAGE_LOCK_NARROW and SECURE_BOOT_PAGE_LOCK_FULL options are not yet available in Simplicity Studio.

4. Click [**Next >**]. The **Security Keys** dialog box is displayed.

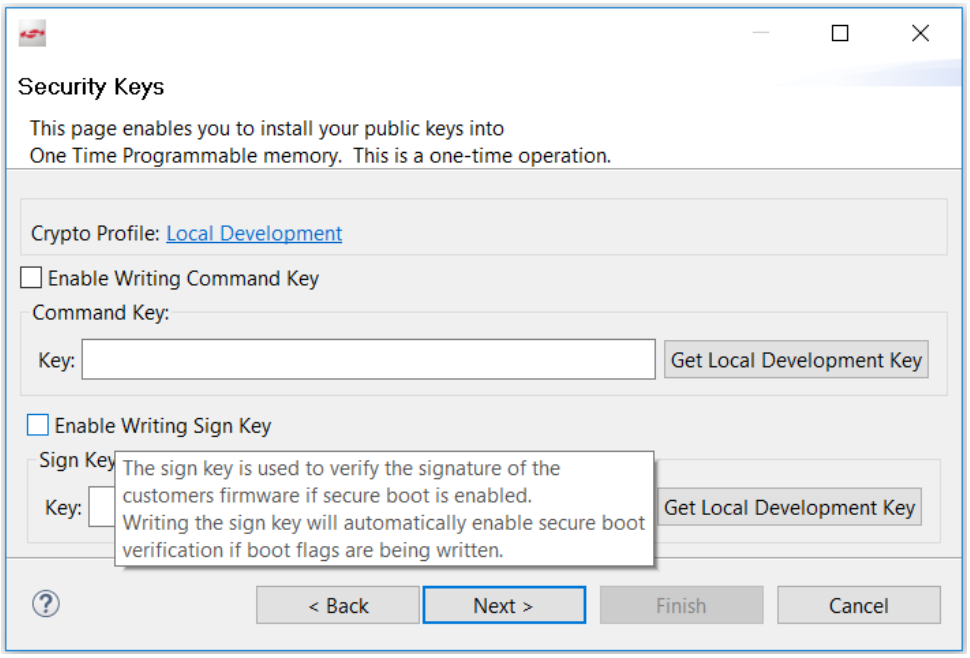


Figure 3.7. Security Keys Dialog Box

5. Checking the **Enable Writing Sign Key** checkbox will automatically enable Secure Boot. The following **Secure Boot Warning** is displayed. Click [**Yes**] to confirm.

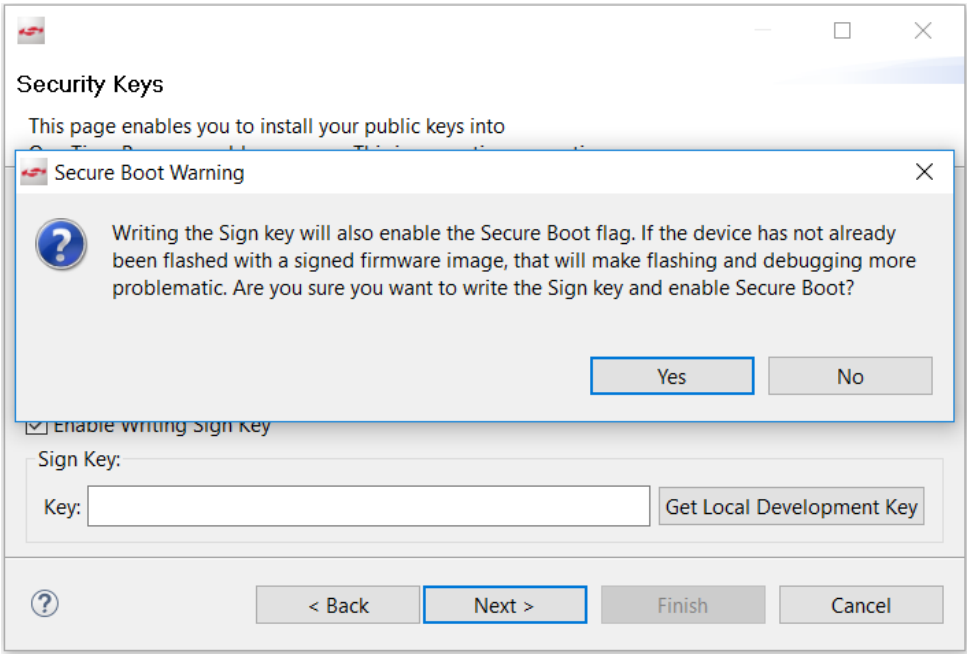


Figure 3.8. Secure Boot Warning

6. Open the Public Sign Key token file (`sign_pubkey.txt`).

```
MFG_SIGNED_BOOTLOADER_KEY_X : 997011ED1708580BD4A6B7F8AD6EE19B0B8722611FB76A3A5702D5141180E101
MFG_SIGNED_BOOTLOADER_KEY_Y : 0AC8673C8ACC26EE2B534C004F4A4B7EBBC23D04506DD66E3EF0DDC81E3CA55E
```

7. Copy Public Sign Key (X-point `9970...` first, then Y-point `0AC8...`) to **Key:** box under **Sign Key:**.

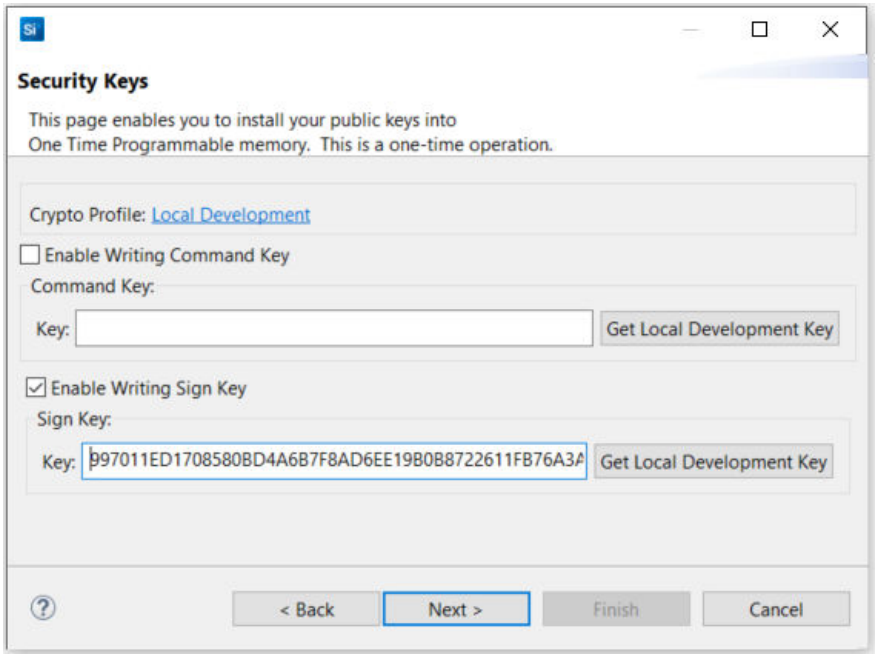


Figure 3.9. Enter Public Sign Key

8. Click [**Next >**]. The **Secure Locks** dialog box is displayed. When Secure Boot is enabled, the **Debug Locks** are not set by default. Refer to AN1190: Series 2 Secure Debug for more information about these locks.

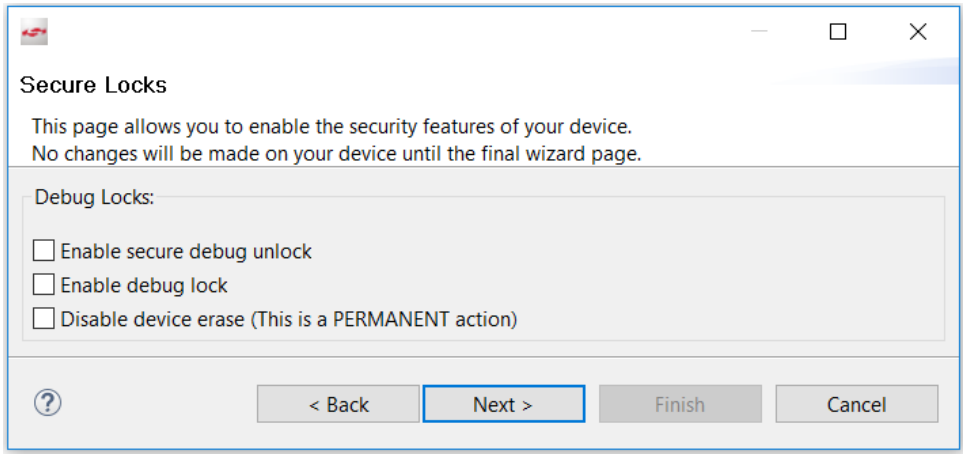


Figure 3.10. Security Locks Dialog Box

9. Click [**Next >**] to display the **Summary** dialog box.

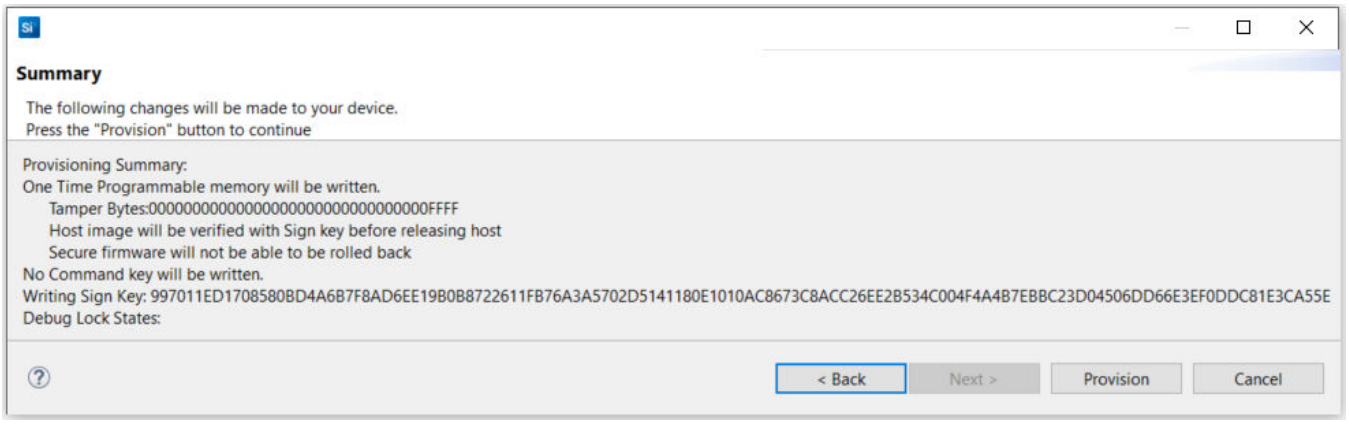


Figure 3.11. Summary Dialog Box

10. If the information displayed is correct, click [**Provision**]. Click [**Yes**] to confirm. The Public Sign Key and Secure Boot configuration cannot be changed once written.

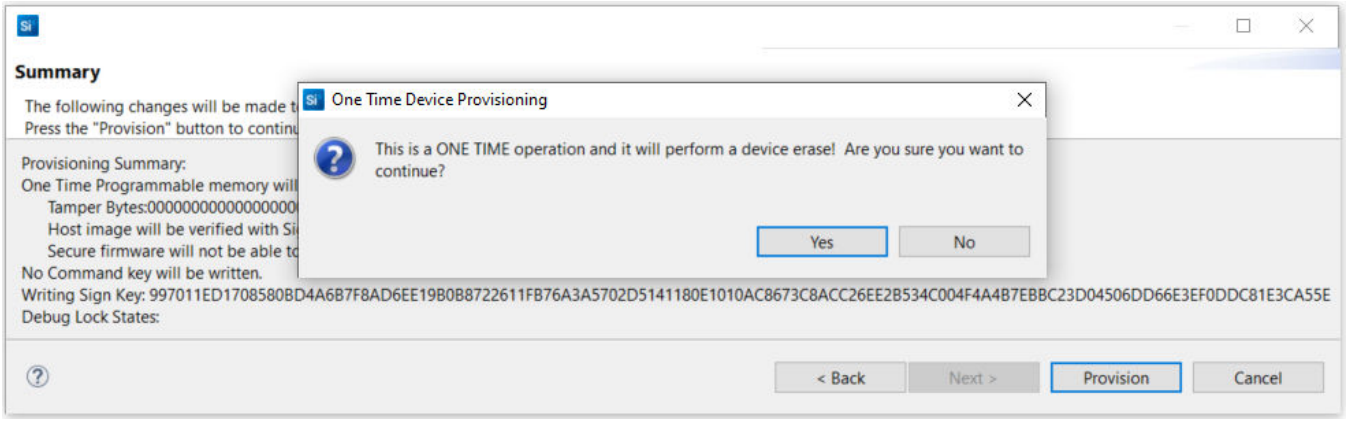


Figure 3.12. One Time Device Provisioning Window

11. The **Summary** dialog box displays the **Provisioning Status**.

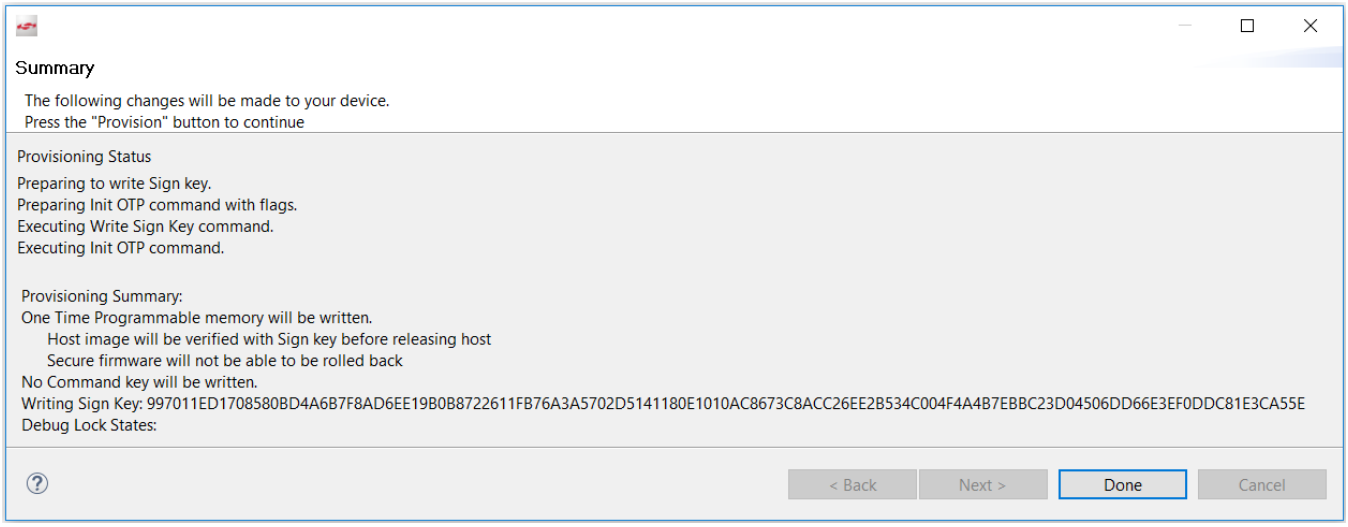


Figure 3.13. Provisioning Status

12. Click [**Done**] to exit the provisioning process. The device configuration is updated.

3.3 Provision GBL Decryption Key

The GBL Decryption Key is used to decrypt the GBL upgrade image file payloads during firmware upgrade. The user should provision this key before enabling the **Require encrypted firmware upgrade files** option in 3.4.1.1 AppBuilder or 3.4.1.2 Bootloader-core Software Component.

The following figure and table describe two (VSE) or three (HSE) options to select which GBL Decryption Key for GBL upgrade image file (bootloader, SE, or application) decryption.

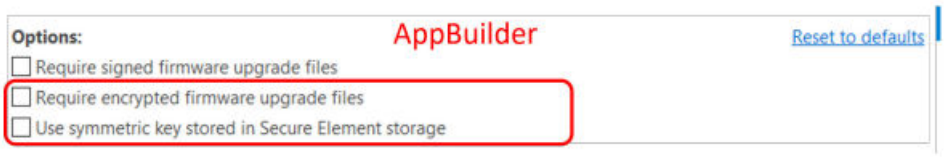


Figure 3.14. AppBuilder Options

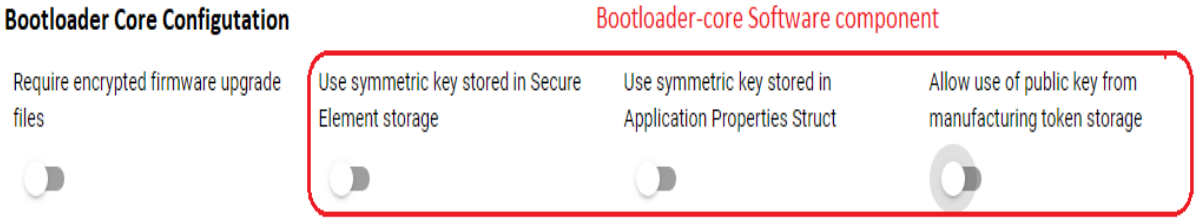


Figure 3.15. Bootloader Software Component Options

GBL Decryption Key Selection Options

Option for GBL Decryption Key Selection	GBL Decryption Key Storage
1. Use symmetric key stored in Secure Element storage (HSE devices only and GSDK ≥ v3.0).	The 128-bit symmetric key stored in HSE OTP is used for GBL upgrade image file decryption.
2. Use symmetric key stored in Application Properties Struct (GSDK ≥ v4.1).	The 128-bit symmetric key stored in the GBL Application Properties Struct is used for GBL upgrade image file decryption. The key is stored in the Secure flash if TrustZone is implemented.
3. Default storage if none of the above options are selected.	The 128-bit symmetric key stored on the top page of the main flash is used for GBL upgrade image file decryption. The key is stored in the Non-secure flash if TrustZone is implemented.

Note:

- Option 2 requires `ApplicationProperties_t` struct v1.2 or higher (defined in `application_properties.h` in the Windows folder `C:\Users\<PC USER NAME>\SimplicityStudio\SDKs\gecko_sdk\platform\bootloader\api`) in GSDK v4.1 or higher to store the GBL Decryption Key.

```
/// Major version number of the AppliationProperties_t struct
#define APPLICATION_PROPERTIES_VERSION_MAJOR (1UL)
/// Minor version number of the AppliationProperties_t struct
#define APPLICATION_PROPERTIES_VERSION_MINOR (2UL)
```

```
/// Application Properties struct
typedef struct {
  /// @brief Magic value indicating this is an ApplicationProperties_t struct.
  /// Must equal @ref APPLICATION_PROPERTIES_MAGIC
  uint8_t magic[16];
  /// Version number of this struct
  uint32_t structVersion;
  /// Type of signature this application is signed with
  uint32_t signatureType;
  /// Location of the signature. Typically points to the end of the application
  uint32_t signatureLocation;
  /// Information about the application
  ApplicationData_t app;
  /// Pointer to information about the certificate
  ApplicationCertificate_t *cert;
  /// Pointer to Long Token Data Section
  uint8_t *longTokenSectionAddress;
  /// Parser Decryption Key
  const uint8_t decryptKey[16];
} ApplicationProperties_t;
```

- Option 2 must be implemented before signing the GBL image for ECDSA-P256-SHA256 or certificate-based Secure Boot.
- The options for the GBL Decryption Key are mutually exclusive. Either one of the two (VSE) or three (HSE) key storages will be selected for decryption.
- From a security point of view, it is highly recommended to use or upgrade to option 1 for HSE devices and option 2 for VSE devices.
- If the GBL Decryption Key in the selected option is compromised, the simple way is to upgrade the GBL to option 2 (if the existing option is 1 or 3) with the new GBL Decryption Key.

3.3.1 Simplicity Commander

The following procedures describe how to program the GBL Decryption Key for the options below. All procedures assume the required files are in the same folder.

- Use symmetric key stored in Secure Element storage (HSE devices only and GSDK ≥ v3.0)
- Use symmetric key stored in Application Properties Struct (GSDK ≥ v4.1)
- Default Storage on the Top Page of the Main Flash

1. Generate a 128-bit Symmetric Key

   Run the `util genkey` to generate the token file for the GBL Decryption Key.

```
commander util genkey --type aes-ccm --outfile aes_key.txt
```

```
Using Windows' Cryptographic random number generator
DONE
```

   The `aes_key.txt` contains the randomly generated AES-128 key. Use the text editor to replace the randomly generated key in `aes_key.txt` with the desired GBL Decryption Key as below.

```
# Key randomly generated by 'util genkey'
TOKEN_MFG_SECURE_BOOTLOADER_KEY: 81A5E21FA15286F1DF445C2CC120FA3F
```

2. (**Use symmetric key stored in Secure Element storage**) Run the `security writekey` to provision the GBL Decryption Key. The GBL Decryption Key cannot be changed once written.

```
commander security writekey --decrypt aes_key.txt --device EFR32MG21A010F1024 --serialno 440030580
```

```
Device has serial number 0000000000000000000d6ffffead3d94

================================================================================
Please look through any warnings before proceeding.
THIS IS A ONE-TIME command, any encrypting of GBL files must be done with this key.
Type 'continue' and hit enter to proceed or Ctrl-C to abort:
================================================================================
continue
DONE
```

   Note: It cannot read back the GBL Decryption Key from the HSE OTP.

3. (**Use symmetric key stored in Application Properties Struct**) Run the `convert` command to program the GBL Decryption Key to the `Application Properties Struct` of the GBL.

```
commander convert bootloader-uart-xmodem.s37 --aeskey aes_key.txt --outfile bootloader-uart-xmodem.s37
```

```
Parsing file bootloader-uart-xmodem.s37...
Writing to bootloader-uart-xmodem.s37...
Overwriting file: bootloader-uart-xmodem.s37...
DONE
```

   Note:
   - The `--aeskey` option in 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.

4. (**Default Storage on the Top Page of the Main Flash**) Run the `flash` command to program the GBL Decryption Key in the token file to the top page of the main flash.

```
commander flash --tokengroup znet --tokenfile aes_key.txt --device EFR32MG21A010F1024 --serialno 440030580
```

```
Writing 8192 bytes starting at address 0x000fe000
Comparing range 0x000FE000 - 0x000FFFFF (8 KB)
Programming range 0x000FE000 - 0x000FFFFF (8 KB)
DONE
```

Note: The MCU Series 2 VSE devices (like EFM32PG22C200F512IM40) require Simplicity Commander Version 1.12.2 or above to support the `flash --tokengroup znet` command.

3.3.2 SE Manager Key Provisioning Platform Example

This example only applies to option 1 for HSE devices. Click the `View Project Documentation` link to open the `readme` file.

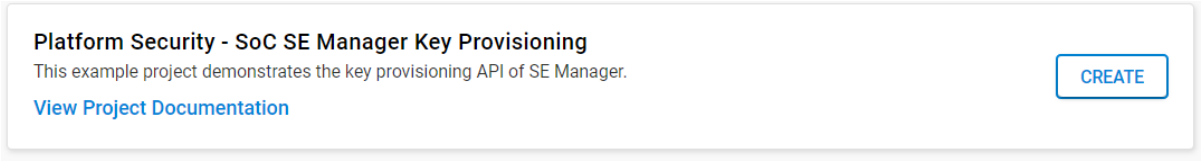


Figure 3.16. Key Provisioning Sample Application

1. Modify the default GBL Decryption Key in `aes_key[16]` array in `app_process.c` to the desired values.

```
/// 128-bit AES key
SL_ALIGN(4) static const uint8_t aes_key[16] = {
  0x81, 0xa5, 0xe2, 0x1f, 0xa1, 0x52, 0x86, 0xf1,
  0xdf, 0x44, 0x5c, 0x2c, 0xc1, 0x20, 0xfa, 0x3f
};
```

2. Modify the `ciphertext[16]` array in `app_process.c` to the expected ciphertext for AES ECB on 16 bytes zero plaintext to verify the GBL Decryption Key in step 1.

```
/// Ciphertext to verify 128-bit AES key
static const uint8_t ciphertext[16] = {
  0x66, 0xd2, 0x0f, 0x99, 0x65, 0x3e, 0xa8, 0xd0,
  0x83, 0x05, 0xa6, 0x39, 0xd4, 0x4e, 0x98, 0xa6
};
```

3. Follow the procedures in 3.4.2 Generate an Unsigned Application Image to generate the unsigned application image if the GBL is present in the device.
4. Build the project and run the application. Follow the procedures in 3.4.3 Signing for ECDSA-P256-SHA256 Secure Boot or 3.4.4 Signing for Certificate-Based Secure Boot if a signed application image is required.
5. Then press `ENTER` to program the hard-coded GBL Decryption Key to HSE OTP.

```
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.
  + Warning: The 128-bit AES key in SE OTP cannot be changed once written!
  + Press ENTER to confirm or press SPACE to skip if you are not sure.
```

6. Press `ENTER` to confirm the operation. The program either returns `SL_STATUS_OK` or `SL_STATUS_INVALID_PARAMETER` (already present) and performs AES ECB encryption to verify the GBL Decryption Key in HSE OTP.

```
  . Initialize 128-bit AES key... SL_STATUS_OK (cycles: 39059 time: 1027 us)

  . Encrypt 16 bytes plaintext with 128-bit AES OTP key... SL_STATUS_OK (cycles: 11013 time: 289 us)
  + Compare encrypted message with expected ciphertext... OK

  . Press ENTER to program public sign key in SE OTP or press SPACE to skip.
```

```
  . Initialize 128-bit AES key... SL_STATUS_INVALID_PARAMETER (cycles: 4474 time: 117 us)

  . Encrypt 16 bytes plaintext with 128-bit AES OTP key... SL_STATUS_OK (cycles: 11001 time: 289 us)
  + Compare encrypted message with expected ciphertext... OK

  . Press ENTER to program public sign key in SE OTP or press SPACE to skip.
```

7. Press SPACE to skip the programming of the Public Sign Key.

```
. Get public sign key... SL_STATUS_FAIL (cycles: 4126 time: 108 us)

. Press ENTER to program public command key in SE OTP or press SPACE to skip.
```

8. Press SPACE to skip the programming of the Public Command Key.

```
. Get public command key... SL_STATUS_FAIL (cycles: 4126 time: 108 us)

. Press ENTER to initialize SE OTP for secure boot configuration or press SPACE to skip.
```

9. Press SPACE to skip the programming of the secure boot configuration.

```
. SE manager deinitialization... SL_STATUS_OK (cycles: 10 time: 0 us)
```

3.4 Secure Boot

The user should usually not enable the Secure Boot during the development phase to avoid a clash on debugging. The Secure Boot feature is enabled near firmware release and uses the following sections to validate the configuration and system functionality.

3.4.1 Generate an Unsigned GBL Image

There are two ways to configure the application firmware through a GBL project.

1. Use AppBuilder (`.isc` file) in **GSDK v3.2 and lower**.
2. Use Bootloader-core software component (`.slcp` file) in **GSDK v4.0 and higher**.

The following notes apply to the AppBuilder and Bootloader-core software component.

- Enabling the **Allow use of public key from manufacturing token storage** option is mandatory on VSE devices (cannot be disabled in AppBuilder and is discarded in Bootloader-core software component) for ECDSA-P256-SHA256 Secure Boot. The HSE device ignores this default enabled option if the Public Sign Key has been provisioned in OTP.
- The GBL stores the application version counter at the end of the bootloader flash space if **Enable application rollback protection** option (GSDK ≥ v3.0) is enabled. The GBL checks the application version during an upgrade and on every boot. The anti-rollback does not prevent flashing an older application image to the device.
- The **Minimum application version allowed** option (GSDK ≥ v3.0) configures the minimum application version that should be allowed to boot. This option maintains the application version counter that will reset to 0 after upgrading the GBL.
- The **Enable application rollback protection** option is not applicable if the SECURE_BOOT_PAGE_LOCK_FULL in SE OTP is enabled. See section "*Secure Boot with Application Rollback Protection*" in UG266 (for GSDK v3.2 and lower) or UG489 (for GSDK v4.0 and higher) for details about the application rollback protection.
- The GBL size and starting address are device-dependent. For more information about the bootloader size and starting address on Series 2 devices, see section "Memory Space For Bootloading" in UG103.6: Bootloader Fundamentals.

Refer to 3.4.5 Generate a GBL Upgrade Image File for the **Require signed firmware upgrade files** and **Require encrypted firmware upgrade files** options on the GBL upgrade image file. For simplicity, the Secure Boot examples in this application note do not enable these options. Refer to UG266/UG489 for information about these options.

The following sections describe how to build the unsigned GBL image from the **UART XMODEM Bootloader** (GSDK < v4.1) or **Bootloader - NCP UART XMODEM** (GSDK ≥ v4.1).

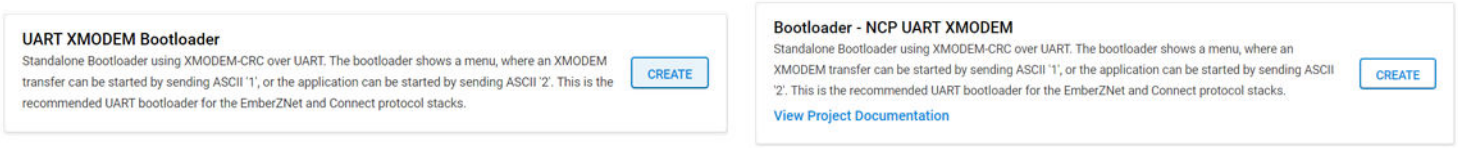


Figure 3.17. Bootloader options

3.4.1.1 AppBuilder

This application note uses UART XMODEM Bootloader example v1.12.0 in GSDK v3.2.3. The procedures and pictures may be different for the other versions of this example.

1. Create a UART XMODEM Bootloader project.
2. The **Plugins** tab in AppBuilder (`bootloader-uart-xmodem.isc`) shows the default configurations for the UART XMODEM Bootloader example.
3. Use **Bootloader Core, provides API: core** in the **Plugins** tab to set up the application firmware configurations.
   a. This application note uses the configuration below for ECDSA-P256-SHA256 Secure Boot.

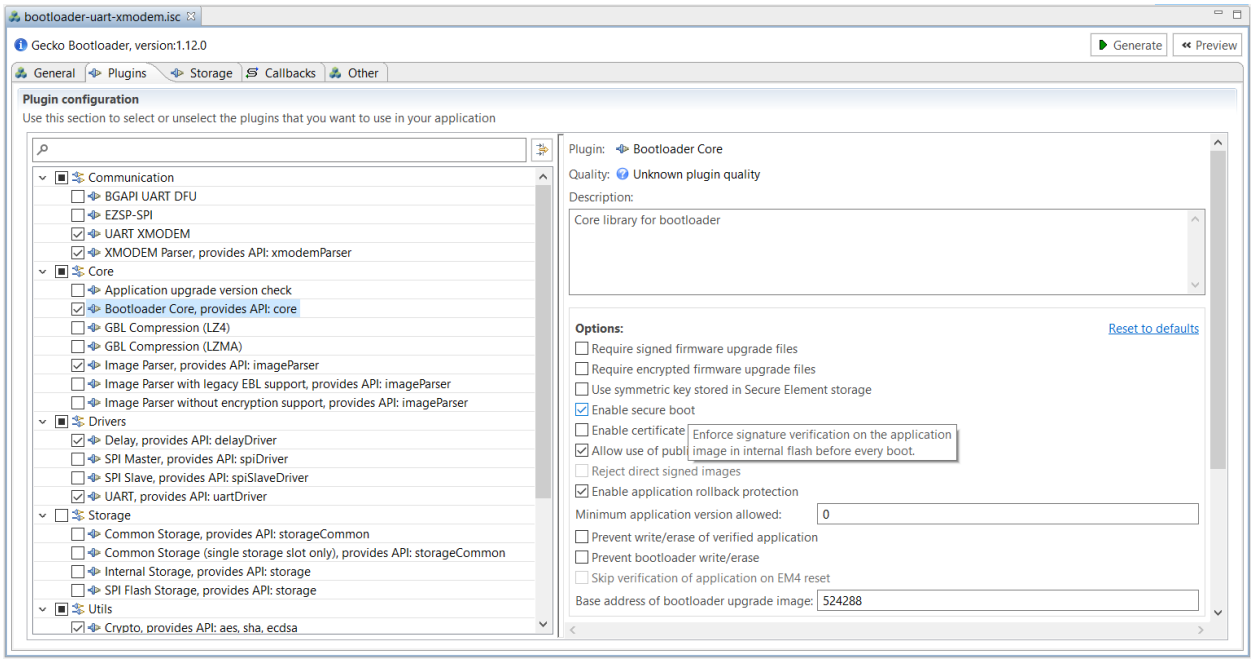


Figure 3.18. ECDSA-P256-SHA256 Secure Boot configuration

   b. This application note uses the configuration below for Certificate-based Secure Boot.

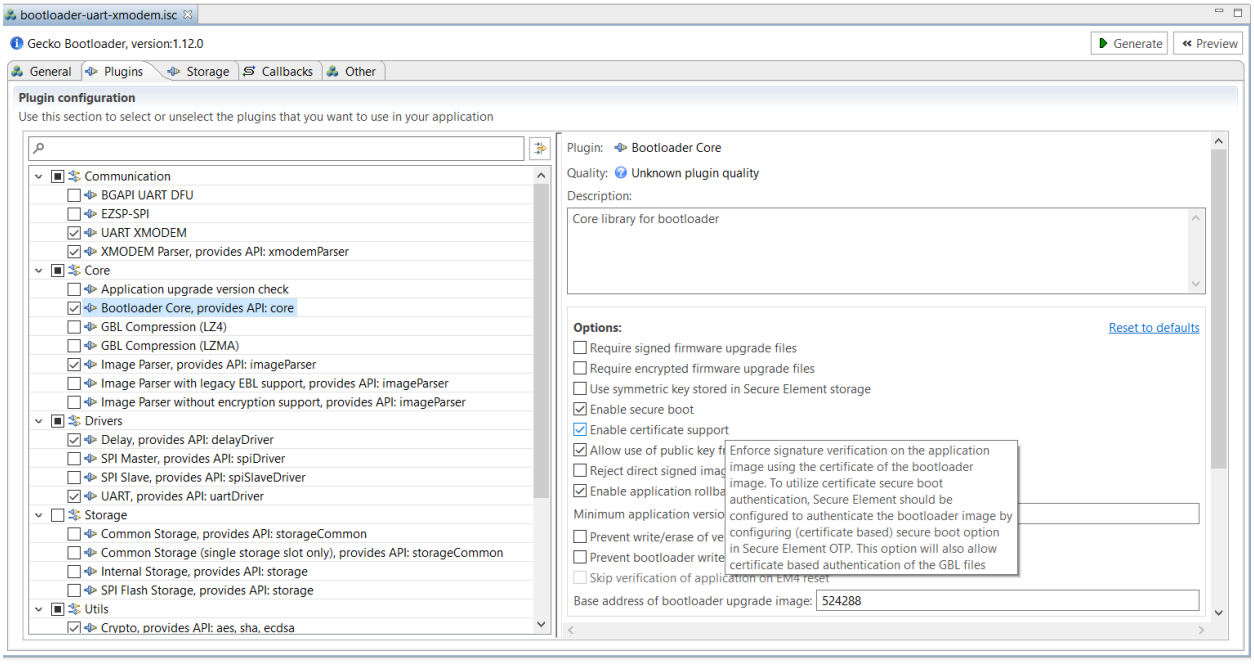


Figure 3.19. Certifiacte-based Secure Boot configuration

4. Enter a higher version number (default is 0) to the macro **BOOTLOADER_VERSION_MAIN_CUSTOMER** → **Value** in the **Additional Macros** field on the **Other** tab for anti-rollback protection of GBL.

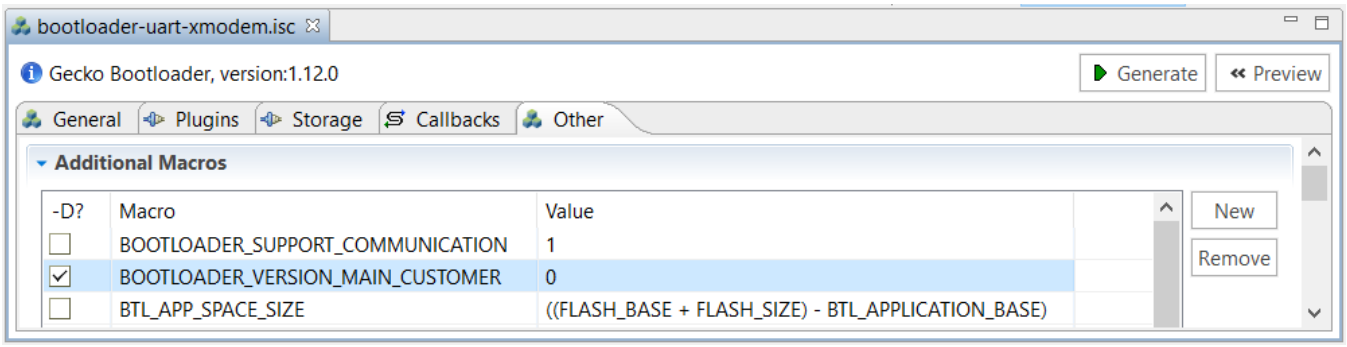


Figure 3.20. Bootloader version macro

5. The default setting of GBL will overwrite the existing application image when upgrading the GBL or SE. It forces to update the application image even without changes on the firmware. Use the Appbuilder settings below to keep the existing application image when upgrading the GBL or SE.
   a. Enter the required application image size to the macro **BTL_APP_SPACE_SIZE** → **Value** in the **Additional Macros** field on the **Other** tab. Check the **-D?** checkbox to add this definition to the project.

      This application note uses `507904` (496 kB) to replace the default value of `((FLASH_BASE + FLASH_SIZE) - BTL_APPLICATION_BASE)`.

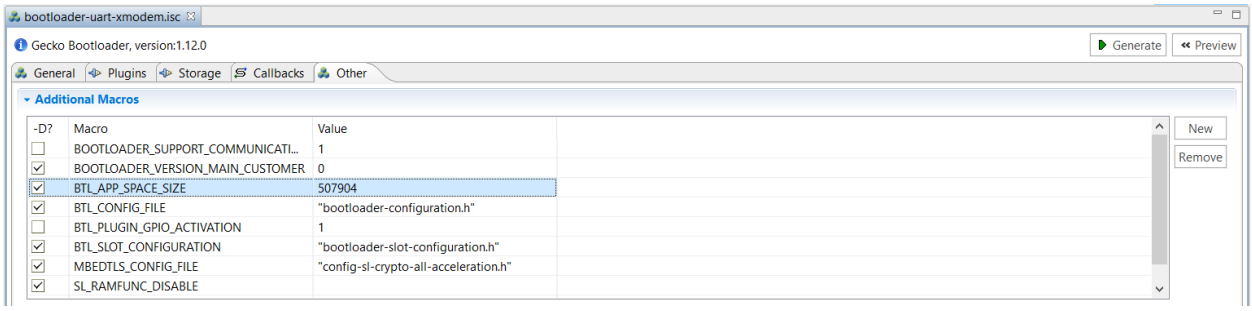


Figure 3.21. Bootloader application size macro

   b. The **Base address of bootloader upgrade image** ≥ (**BTL_APP_SPACE_SIZE** + size of the GBL).

      The example in this application note uses EFR32MG21A010F1024:

      `Base address of bootloader upgrade image` = `507904` (496 kB) + `16384` (16 kB for GBL) = `524288` (512 kB)

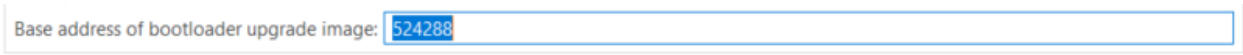


Figure 3.22. Base address of bootlader upgrade image

      Note: The default value of `Base address of bootloader upgrade image` is `32768` (32 kB).

   c. The (**Base address of bootloader upgrade image** + size of the GBL or SE + upgrade file overhead) ≤ the available size of the device main flash for application use (see project linker file for details).

      The example in this application note uses EFR32MG21A010F1024:

      512 kB (Base address of bootloader upgrade image) + 16 kB (GBL) or 48 kB (SE) + overhead bytes < 1024 kB (size of main flash)

      For more information about the size of the GBL and SE, see section "*Storage Space Size Configuration*" in UG266.

   Note: It requires GBL v1.11.0 or above to support this feature.

6. Click [**Generate**] in the right upper corner.

7. In the **Generation Successful** dialog, click [**OK**].
8. Build the project to generate the unsigned GBL image file (`bootloader-uart-xmodem.s37`).
9. **(Optional)** Run the `util appinfo` command to check all available information (application properties) in an unsigned GBL image. The `App version` is the GBL version for the SECURE_BOOT_ANTI_ROLLBACK option.

```
commander util appinfo bootloader-uart-xmodem.s37
```

```
Parsing file bootloader-uart-xmodem.s37...
Found application properties in image.
Application properties info:
Application properties location : 0x00002b1c
Signature location              : 0x00002d08
Signature type                  : No signature
Long token section address      : Not set (0x00000000)

Application data info:
For Series 2 devices: If rollback prevention is enabled in the OTP configuration, the device will not boot
if the device has seen an application with a higher version number.
App type                        : Bootloader (APPLICATION_TYPE_BOOTLOADER)
App version                     : 0x010c0000
Product ID                      : Not set (0x00000000000000000000000000000000)

No certificate found in image.
For Series 2 devices: If the configuration flag SECURE_BOOT_VERIFY_CERTIFICATE is set or a device has
previously seen certificate based signing, it will not accept direct signing.
DONE
```

3.4.1.2 Bootloader-core Software Component

This application note uses UART XMODEM Bootloader example v2.0.0 in GSDK v4.0. The procedures and pictures may be different for the other versions of this example.

1. Create a UART XMODEM Bootloader project.
2. Checking the **Installed Components** under the **SOFTWARE COMPONENTS** tab shows the list of installed components (`bootloader-uart-xmodem.slcp`) in the UART XMODEM Bootloader example.
3. Click [**Configure**] in the **Bootloader-core** component to open the **Bootloader Core Configuration**.

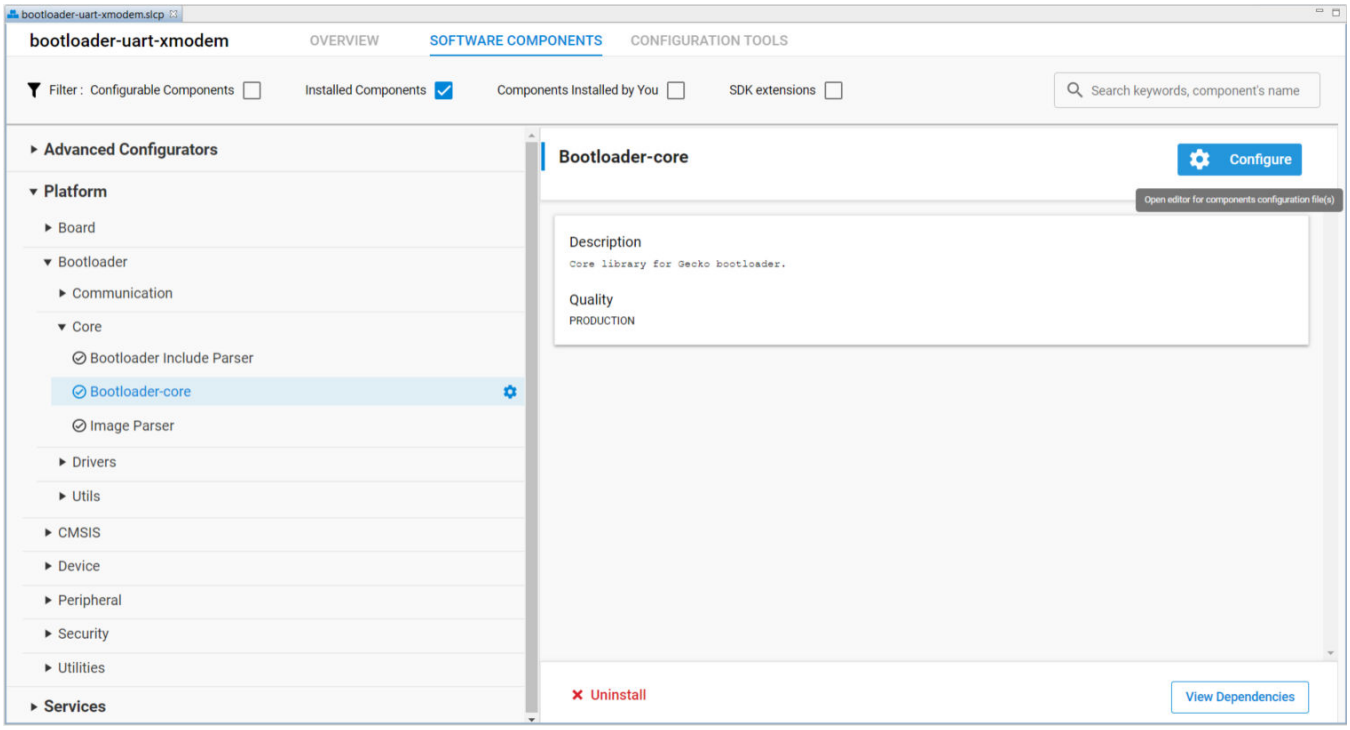


Figure 3.23. Bootloader-core component

4. Use **Bootloader Core Configuration** in **Bootloader-core** to set up the application firmware configurations.
   a. This application note uses the configuration below for ECDSA-P256-SHA256 Secure Boot.

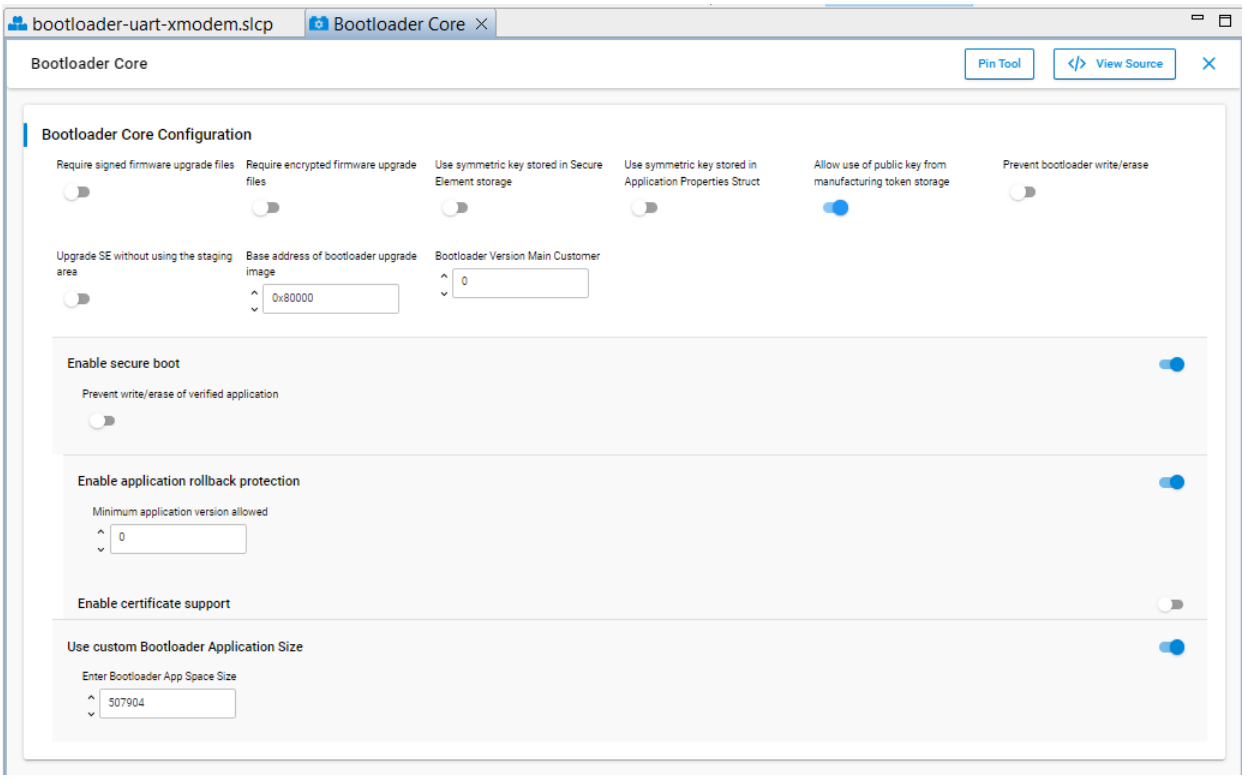


Figure 3.24. Bootloader Core configuration for ECDSA-P256-SHA256 Secure Boot

   b. This application note uses the configuration below for Certificate-based Secure Boot.

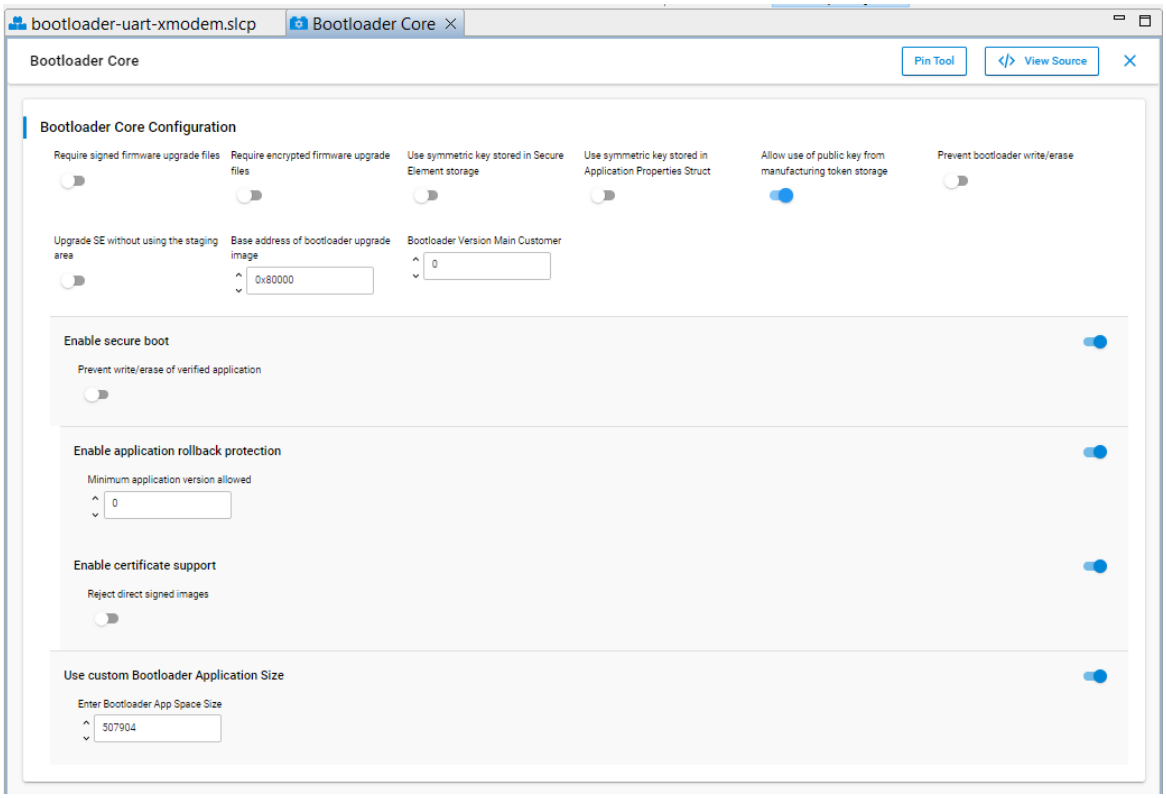
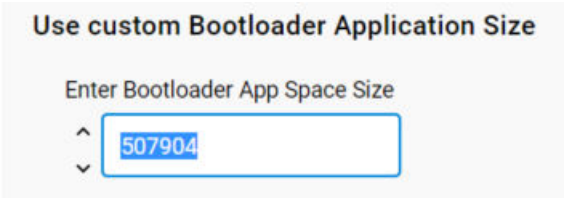


Figure 3.25. Bootloader Core configuration for Certificate-based Secure Boot

5. Enter a higher version number (default is 0) to **BOOTLOADER VERSION MAIN CUSTOMER** for anti-rollback protection of GBL.

6. The default setting of GBL will overwrite the existing application image when upgrading the GBL or SE. It forces to update the application image even without changes on the firmware. Use the Bootloader-core settings below to keep the existing application image when upgrading the GBL or SE.
   a. Enter the required application image size to the **Enter Bootloader App Space Size** dialog box after enabling the **Use custom Bootloader Application Size** option.

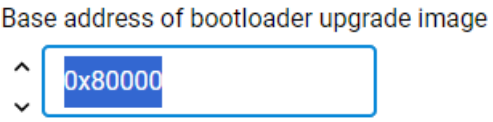
      This application note uses `507904` (496 kB) to replace the default value of `0`.



   b. The **Base address of bootloader upgrade image** ≥ (**Enter Bootloader App Space Size** + size of the GBL).

      The example in this application note uses EFR32MG21A010F1024:

      `Base address of bootloader upgrade image` = `507904` (496 kB) + `16384` (16 kB for GBL) = `524288` or 0x80000 (512 kB)



      Note: The default value of `Base address of bootloader upgrade image` is `32768` or 0x8000 (32 kB).

   c. The (**Base address of bootloader upgrade image** + size of the GBL or SE + upgrade file overhead) ≤ the available size of the device main flash for application use (see project linker file for details).

      The example in this application note uses EFR32MG21A010F1024:

      512 kB (Base address of bootloader upgrade image) + 16 kB (GBL) or 48 kB (SE) + overhead bytes < 1024 kB (size of main flash)

      For more information about the size of the GBL and SE, see section "*Storage Space Size Configuration*" in UG489.
7. Click [**X**] in the right upper corner to exit the **Bootloader Core Configuration**.
8. Build the project to generate the unsigned GBL image file (`bootloader-uart-xmodem.s37`).
9. **(Optional)** Run the `convert` command to program the GBL Decryption Key to the `Application Properties Struct` if this GBL Decryption Key option in GBL (GSDK ≥ v4.1) is selected.

```
commander convert bootloader-uart-xmodem.s37 --aeskey aes_key.txt --outfile bootloader-uart-xmodem.s37
```

```
Parsing file bootloader-uart-xmodem.s37...
Writing to bootloader-uart-xmodem.s37...
Overwriting file: bootloader-uart-xmodem.s37...
DONE
```

10. **(Optional)** Run the `util appinfo` command to check all available information (application properties) in an unsigned GBL image. The `App version` is the GBL version for the SECURE_BOOT_ANTI_ROLLBACK option.

```
commander util appinfo bootloader-uart-xmodem.s37
```

```
Parsing file bootloader-uart-xmodem.s37...
Found application properties in image.
Application properties info:
Application properties location : 0x00002b30
Signature location              : 0x00002c44
Signature type                  : No signature
Long token section address      : Not set (0x00000000)

Application data info:
For Series 2 devices: If rollback prevention is enabled in the OTP configuration, the device will not boot if the
device has seen an application with a higher version number.
App type                        : Bootloader (APPLICATION_TYPE_BOOTLOADER)
App version                     : 0x02000000
Product ID                      : Not set (0x00000000000000000000000000000000)

No certificate found in image.
For Series 2 devices: If the configuration flag SECURE_BOOT_VERIFY_CERTIFICATE is set or a device has previously seen
certificate based signing, it will not accept direct signing.
DONE
```

Note: For the TrustZone-aware bootloaders, the unsigned GBL image is the combined image of Secure and Non-secure bootloaders. The **Bootloader-core** component is installed in the Secure bootloader.

3.4.2 Generate an Unsigned Application Image

This section describes how to generate an unsigned application image for the GBL.

1. For Series 2 devices, the application image should place on the main flash page after the GBL. For more information about the application starting address, see section "Memory Space For Bootloading" in UG103.6: Bootloader Fundamentals.
2. **(Simplicity Studio 4)** Refer to the section "*Creating Applications for Use with the Bootloader*" in AN0042: USB/UART Bootloader or the Knowledge Article (Simplicity IDE) in Silicon Labs Community to set up the application image start address.
3. **(Simplicity Studio 5)** The user can use the **Bootloader Application Interface** component to set up the start address of the application image. This application note uses **Platform - Blink Bare-metal** example in GSDK v3.2.3. The procedures and pictures may be different on other versions of the GSDK. The following steps can apply to other platform examples in GSDK.
   a. Create a Platform - Blink Bare-metal project.
   b. The **Software Components** tab shows the list of available components (`blink_bardmetal.slcp`) that the user can install in the project.
   c. Select **Platform** → **Bootloader** → **Bootloader Application Interface**.
   d. Click [**Install**].

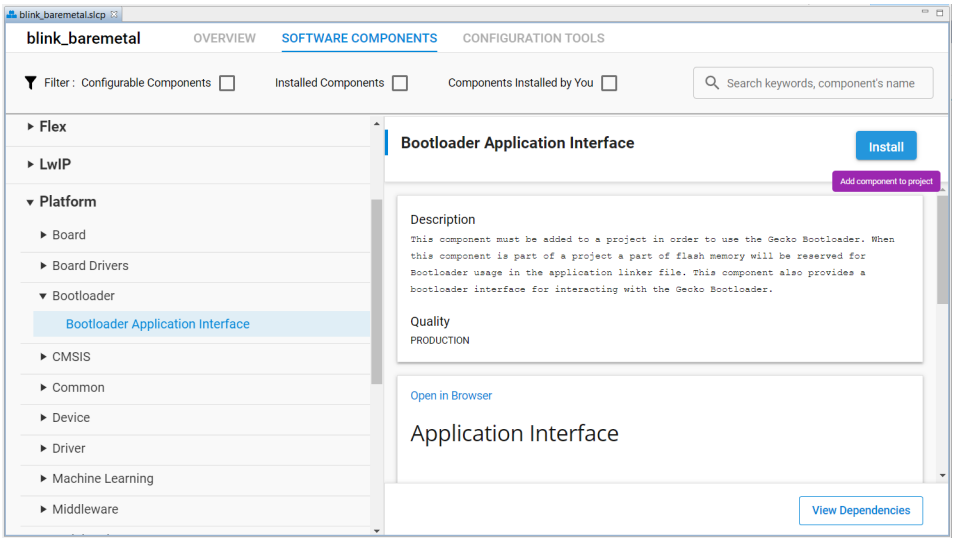


Figure 3.26. Bootloader Application Interface

Note: For the wireless protocol stack example, the Bootloader Application Interface component is already present in the project.

4. The application image should contain an `ApplicationProperties_t` struct (defined in `application_properties.h` in the Windows folder below) declaring the application version, capabilities, and other metadata.

   For GSDK v3.2 and lower: `C:\SiliconLabs\SimplicityStudio\v5\developer\sdks\gecko_sdk_suite\<GSDK VERSION>\platform\bootloader\api`

   For GSDK v4.0 and higher: `C:\Users\<PC USER NAME>\SimplicityStudio\SDKs\gecko_sdk\platform\bootloader\api`

   Below is an example source file `app_properties.c` with `ApplicationProperties_t` struct for Secure Boot on GSDK v3.2 and lower.

```
#include <stddef.h>
#include "application_properties.h"

const ApplicationProperties_t sl_app_properties = {
  .magic = APPLICATION_PROPERTIES_MAGIC,
  .structVersion = APPLICATION_PROPERTIES_VERSION,
  .signatureType = APPLICATION_SIGNATURE_NONE,
  .signatureLocation = 0,
  .app = {
    .type = APPLICATION_TYPE_MCU,
    .version = 1UL,
    .capabilities = 0UL,
    .productId = {0U},
  },
};
```

   The `signatureType` and `signatureLocation` are filled by Simplicity Commander when signing the application image using the `convert` command.

5. The following table describes how to add the `app_properties.c` file in step 4 to Platform - Blink Bare-metal project. For the wireless protocol stack example, the `app_properties.c` file with `ApplicationProperties_t` struct is already present in the project.

Table 3.4. App properties for older Simplicity Studio and GSDK versions

Simplicity Studio 4 & Simplicity Studio 5 with GSDK v3.2 and lower	Simplicity Studio 5 with GSDK v4.0 and higher
Manually added	Automatically added after installing the Bootloader Application Interface component in step 3 to the project.

   Note: Refer to the Knowledge Article in Silicon Labs Community to add `app_properties.c` to the project in Simplicity Studio 4.

6. **(Simplicity Studio 4 & Simplicity Studio 5 with GSDK v3.2 and lower)** Enter a higher version number to `.version` in `app_properties.c` for anti-rollback protection (if enabled) of the application.

7. **(Simplicity Studio 5 with GSDK v4.0 and higher)** Click [**Configure**] in the **App Properties** component under **Platform** → **Bootloader** to open the **App Properties** configuration. The example below uses GSDK v4.0. The procedures and pictures may be different on other versions of the GSDK.

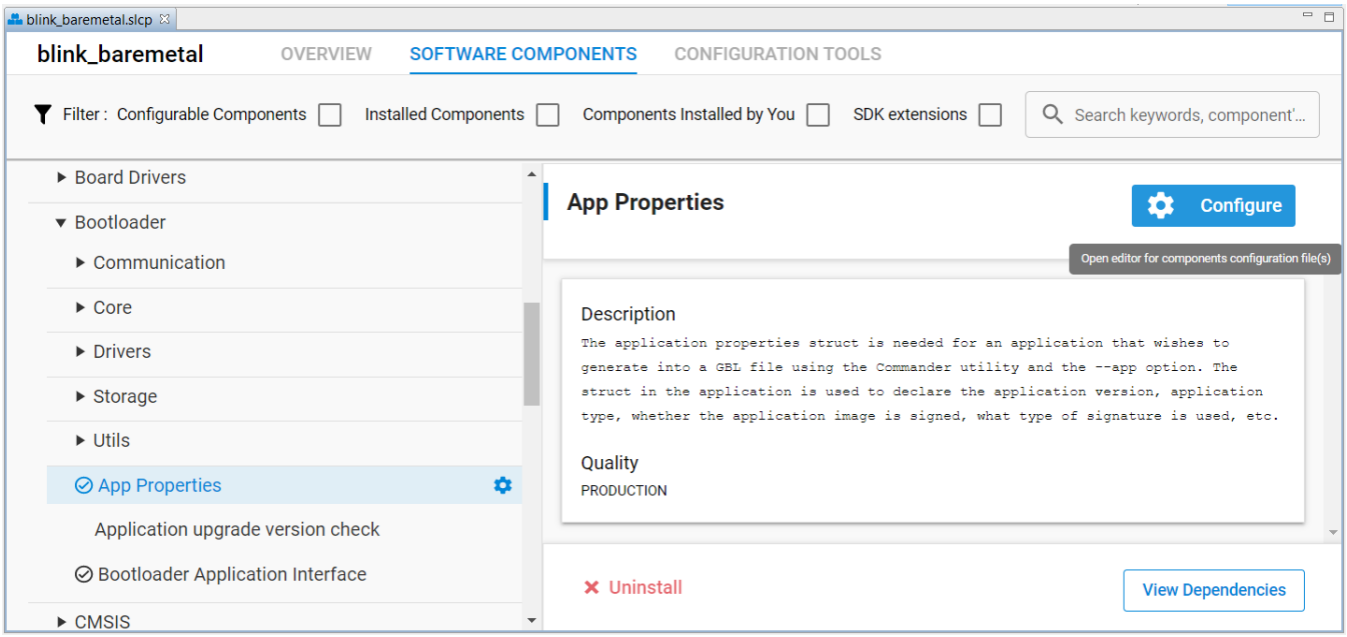


Figure 3.27. App Properties component

Enter a higher version number to **Version number for this application** dialog box in **App Properties settings** for anti-rollback protection (if enabled) of the application.

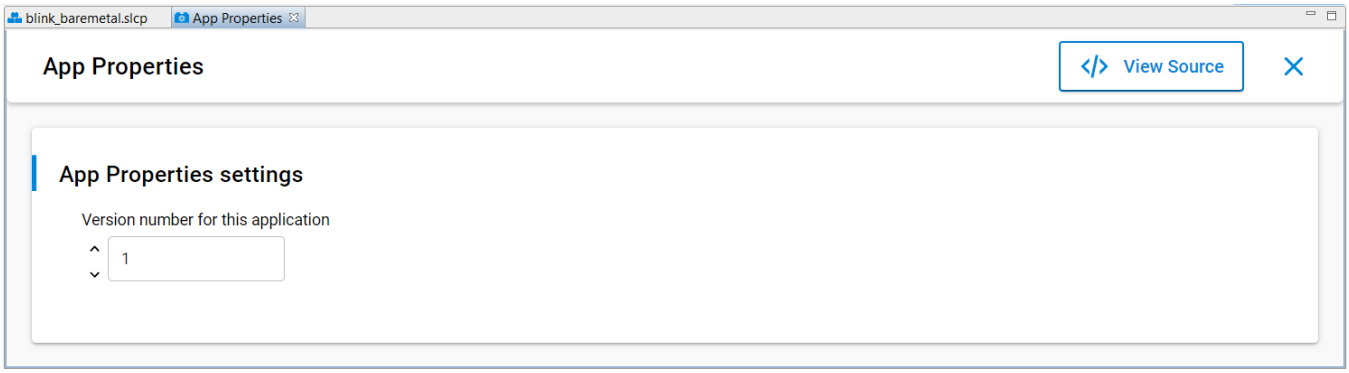


Figure 3.28. App properties version number dialog box

Note: The `app_properties.c` is in the Windows folder below.

```
C:\Users\<PC USER NAME>\SimplicityStudio\SDKs\gecko_sdk\platform\bootloader\app_properties
```

8. Build the project to generate the unsigned application image file (`blink_bardmetal.s37`).

9. **(Optional)** Run the `util appinfo` command to check all available information about `ApplicationProperties_t` struct in an unsigned application image. The `App version` is for the **Enable application rollback protection** option in the AppBuilder or Bootloader-core software component.

```
commander util appinfo blink_baremetal.s37
```

```
Parsing file blink_baremetal.s37...
Found application properties in image.
Application properties info:
Application properties location : 0x00006198
Signature location              : Not set (0x00000000)
Signature type                  : No signature
Long token section address      : Not set (0x00000000)

Application data info:
For Series 2 devices: If rollback prevention is enabled in the OTP configuration, the device will not boot
if the device has seen an application with a higher version number.
App type                        : MCU application (APPLICATION_TYPE_MCU)
App version                     : 0x00000001
Product ID                      : Not set (0x00000000000000000000000000000000)

No certificate found in image.
For Series 2 devices: If the configuration flag SECURE_BOOT_VERIFY_CERTIFICATE is set or a device has
previously seen certificate based signing, it will not accept direct signing.
DONE
```

Note: For the TrustZone-aware applications, the unsigned application image is the combined image of Secure and Non-secure applications. The `ApplicationProperties_t` struct is located in the Secure application.

3.4.3 Signing for ECDSA-P256-SHA256 Secure Boot

The following figure describes the signing and verification for ECDSA-P256-SHA256 Secure Boot.

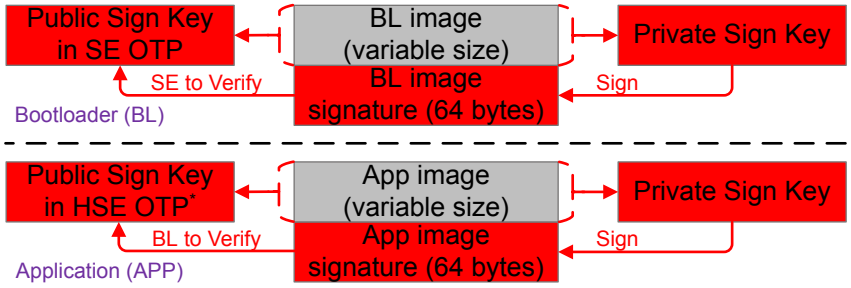


Figure 3.29. ECDSA-P256-SHA256 Sign and Verify

Note:

- *The bootloader cannot access the Public Sign Key in VSE OTP to verify the application image. Therefore VSE devices need to store a Public Sign Key copy on the top page of the main flash (see section "*Key Storage*" in UG266/UG489).

Table 3.5. Public Sign Key usage for VSE and HSE Devices

Device	FSB to Verify the Bootloader Image	SSB to Verify the Application Image
HSE	Use the Public Sign Key in HSE OTP	Use the Public Sign Key in HSE OTP
VSE	Use the Public Sign Key in VSE OTP	Use the Public Sign Key on the top page of the main flash

The HSE device ignores the default enabled Allow use of public key from manufacturing token storage option once the Public Sign Key has been provisioned.

- To have better protection on the Public Sign Key. The certificate-based Secure Boot is strongly recommended on VSE devices since the SSB does not require accessing the Public Sign Key to verify the application signature.

The following sections provide two methods to sign the bootloader image and application image files. All procedures assume the required files are in the same folder.

1. Using Simplicity Commander
2. Using an HSM and Simplicity Commander

Bootloader Image File

1. If the SE OTP is not provisioned, follow the procedures in 3.2.1 Simplicity Commander to set up the ECDSA-P256-SHA256 Secure Boot configuration for the bootloader.
2. Follow the procedures in 3.4.1.1 AppBuilder or 3.4.1.2 Bootloader-core Software Component to set up the ECDSA-P256-SHA256 Secure Boot configuration for the user application to generate an unsigned bootloader image.

3. **(Using Simplicity Commander)** Run the `convert` command with **Private Sign Key** to overwrite the unsigned bootloader image file with the signed bootloader image file (`bootloader-uart-xmodem.s37`).

```
commander convert bootloader-uart-xmodem.s37 --secureboot --keyfile sign_key.pem --verify sign_pubkey.pem
--outfile bootloader-uart-xmodem.s37
```

```
Parsing file bootloader-uart-xmodem.s37...
Found Application Properties at 0x00002b1c
Writing Application Properties signature pointer to point to 0x00002d08
Setting signature type in Application Properties: 0x00000001
Image SHA256: c53bb8a3fd88a5071bfb71444324bb136b276160318488ff89011bbd269e114e
R = AB62F3A52B13D137FBCC6A2176D4D1852E06B6E4E6B2673DC251FC491450CBDA
S = 9C7C7AF2624165FD90FB3B114E3FA6FE4F4C5625B15C9F3D50DCB04DD06A7B19

Verifying signed image...
Writing to bootloader-uart-xmodem.s37...
Overwriting file: bootloader-uart-xmodem.s37...
DONE
```

4. **(Using an HSM and Simplicity Commander)** Run the `convert` command with **--extsign** option to generate an external signing bootloader image file (`bootloader-uart-xmodem.extsign`).

```
commander convert bootloader-uart-xmodem.s37 --secureboot --extsign --outfile bootloader-uart-xmodem
```

```
Parsing file bootloader-uart-xmodem.s37...
Found Application Properties at 0x00002b1c
Writing Application Properties signature pointer to point to 0x00002d08
Setting signature type in Application Properties: 0x00000001
Writing to bootloader-uart-xmodem.extsign...
DONE
```

Use an HSM containing the Private Sign Key to generate the signature for the external signing bootloader image. This example uses the OpenSSL with the **Private Sign Key** to simulate this process. The signature is in the `bl_signature.der`.

```
openssl dgst -sha256 -binary -sign sign_key.pem -out bl_signature.der bootloader-uart-xmodem.extsign
```

Run the `convert` command with the **bootloader image signature** to overwrite the unsigned bootloader image file with the signed bootloader image file (`bootloader-uart-xmodem.s37`).

```
commander convert bootloader-uart-xmodem.s37 --secureboot --signature bl_signature.der
--verify sign_pubkey.pem --outfile bootloader-uart-xmodem.s37
```

```
Parsing file bootloader-uart-xmodem.s37...
Parsing signature file bl_signature.der...
R = 0E9FC64F41B55367894908D3ADAC40E8D145E33224C4BAA8151EC3EFD107A154
S = F56230AA6484E55270F22A4D164377CA918F66A367656AB6E10CB3F58641CE84
Found Application Properties at 0x00002b1c
Writing Application Properties signature pointer to point to 0x00002d08
Setting signature type in Application Properties: 0x00000001

Verifying signed image...
Writing to bootloader-uart-xmodem.s37...
Overwriting file: bootloader-uart-xmodem.s37...
DONE
```

5. **(Optional)** Run the `util appinfo` command to check all available information about `ApplicationProperties_t` struct in a signed GBL image.

```
commander util appinfo bootloader-uart-xmodem.s37
```

```
Parsing file bootloader-uart-xmodem.s37...
Found application properties in image.
Application properties info:
Application properties location : 0x00002b30
Signature location              : 0x00002c44
Signature type                  : ECDSA-P256
Long token section address      : Not set (0x00000000)

Application data info:
For Series 2 devices: If rollback prevention is enabled in the OTP configuration, the device will not boot
if the device has seen an application with a higher version number.
App type                        : Bootloader (APPLICATION_TYPE_BOOTLOADER)
App version                     : 0x02000000
Product ID                      : Not set (0x00000000000000000000000000000000)

No certificate found in image.
For Series 2 devices: If the configuration flag SECURE_BOOT_VERIFY_CERTIFICATE is set or a device has
previously seen certificate based signing, it will not accept direct signing.
DONE
```

6. The signed bootloader image file (`.s37`) can be used for production programming or for generating a GBL upgrade image file for bootloader upgrade.
7. Run the `flash` command to program the signed bootloader image (`bootloader-uart-xmodem.s37`) to the device if the device does not have a bootloader.

```
commander flash bootloader-uart-xmodem.s37 --device EFR32MG21A010F1024 --serialno 440048205
```

```
Parsing file bootloader-uart-xmodem.s37...
Writing 16384 bytes starting at address 0x00000000
Comparing range 0x00000000 - 0x00003FFF (16 KiB)
Programming range 0x00000000 - 0x00001FFF (8 KiB)
Programming range 0x00002000 - 0x00003FFF (8 KiB)
DONE
```

Application Image File

1. Follow the procedures in 3.4.2 Generate an Unsigned Application Image to generate an unsigned application image for the bootloader.
2. **(Using Simplicity Commander)** Run the `convert` command with **Private Sign Key** to overwrite the unsigned application image file with the signed application image file (`blink_baremetal.s37`).

```
commander convert blink_baremetal.s37 --secureboot --keyfile sign_key.pem --verify sign_pubkey.pem
--outfile blink_baremetal.s37
```

```
Parsing file blink_baremetal.s37...
Found Application Properties at 0x000061bc
Writing Application Properties signature pointer to point to 0x000064d8
Setting signature type in Application Properties: 0x00000001
Image SHA256: 8b58ec567126aa1f6baa88afc916581477745aca6f47697ec093512fc30dcc6f
R = 056E3AA36BD882B5467D44A56DB7CC1AEE44D45BC9B98FAB05BE2C032573A1F7
S = BE1D27CE7877D0BC761C0F02690CC74251EBE3A458474C573C21B3A738A03577

Verifying signed image...
Writing to blink_baremetal.s37...
Overwriting file: blink_baremetal.s37...
DONE
```

3. **(Using an HSM and Simplicity Commander)** Run the `convert` command with **--extsign** option to generate an external signing application image file (`blink_baremetal.extsign`).

```
commander convert blink_baremetal.s37 --secureboot --extsign --outfile blink_baremetal
```

```
Parsing file blink_baremetal.s37...
Found Application Properties at 0x00006198
Writing Application Properties signature pointer to point to 0x0000643c
Setting signature type in Application Properties: 0x00000001
Writing to blink_baremetal.extsign...
DONE
```

Use an HSM containing the Private Sign Key to generate the signature for the external signing application image. This example uses the OpenSSL with the **Private Sign Key** to simulate this process. The signature is in the `app_signature.der`.

```
openssl dgst -sha256 -binary -sign sign_key.pem -out app_signature.der blink_baremetal.extsign
```

Run the `convert` command with the **application image signature** to overwrite the unsigned application image file with the signed application image file. (`blink_baremetal.s37`).

```
commander convert blink_baremetal.s37 --secureboot --signature app_signature.der --verify sign_pubkey.pem
--outfile blink_baremetal.s37
```

```
Parsing file blink_baremetal.s37...
Parsing signature file app_signature.der...
R = BD5BDC866CE67DA104B1E7B686C45B7BF96F2643154D37ACC63DACDF69C27E89
S = 2DD3BFFAC857A5B0BD8C9B4DDB23D21944D062F8E431D36541B84EF411C1CC92
Found Application Properties at 0x000061bc
Writing Application Properties signature pointer to point to 0x000064d8
Setting signature type in Application Properties: 0x00000001

Verifying signed image...
Writing to blink_baremetal.s37...
Overwriting file: blink_baremetal.s37...
DONE
```

4. **(Optional)** Run the `util appinfo` command to check all available information about `ApplicationProperties_t` struct in a signed application image.

```
commander util appinfo blink_baremetal.s37
```

```
Parsing file blink_baremetal.s37...
Found application properties in image.
Application properties info:
Application properties location : 0x000061bc
Signature location              : 0x000064d8
Signature type                  : ECDSA-P256
Long token section address      : Not set (0x00000000)

Application data info:
For Series 2 devices: If rollback prevention is enabled in the OTP configuration, the device will not boot
if the device has seen an application with a higher version number.
App type                        : MCU application (APPLICATION_TYPE_MCU)
App version                     : 0x00000001
Product ID                      : Not set (0x00000000000000000000000000000000)

No certificate found in image.
For Series 2 devices: If the configuration flag SECURE_BOOT_VERIFY_CERTIFICATE is set or a device has
previously seen certificate based signing, it will not accept direct signing.
DONE
```

5. The signed application image file (`.s37`) can be used for production programming or for generating a GBL upgrade image file for application upgrade.

3.4.4 Signing for Certificate-Based Secure Boot

The following figure describes the signing and verification for certificate-based Secure Boot. The user can freely switch between standard and advanced certificate-based Secure Boot by upgrading the application firmware without and with the application certificate.

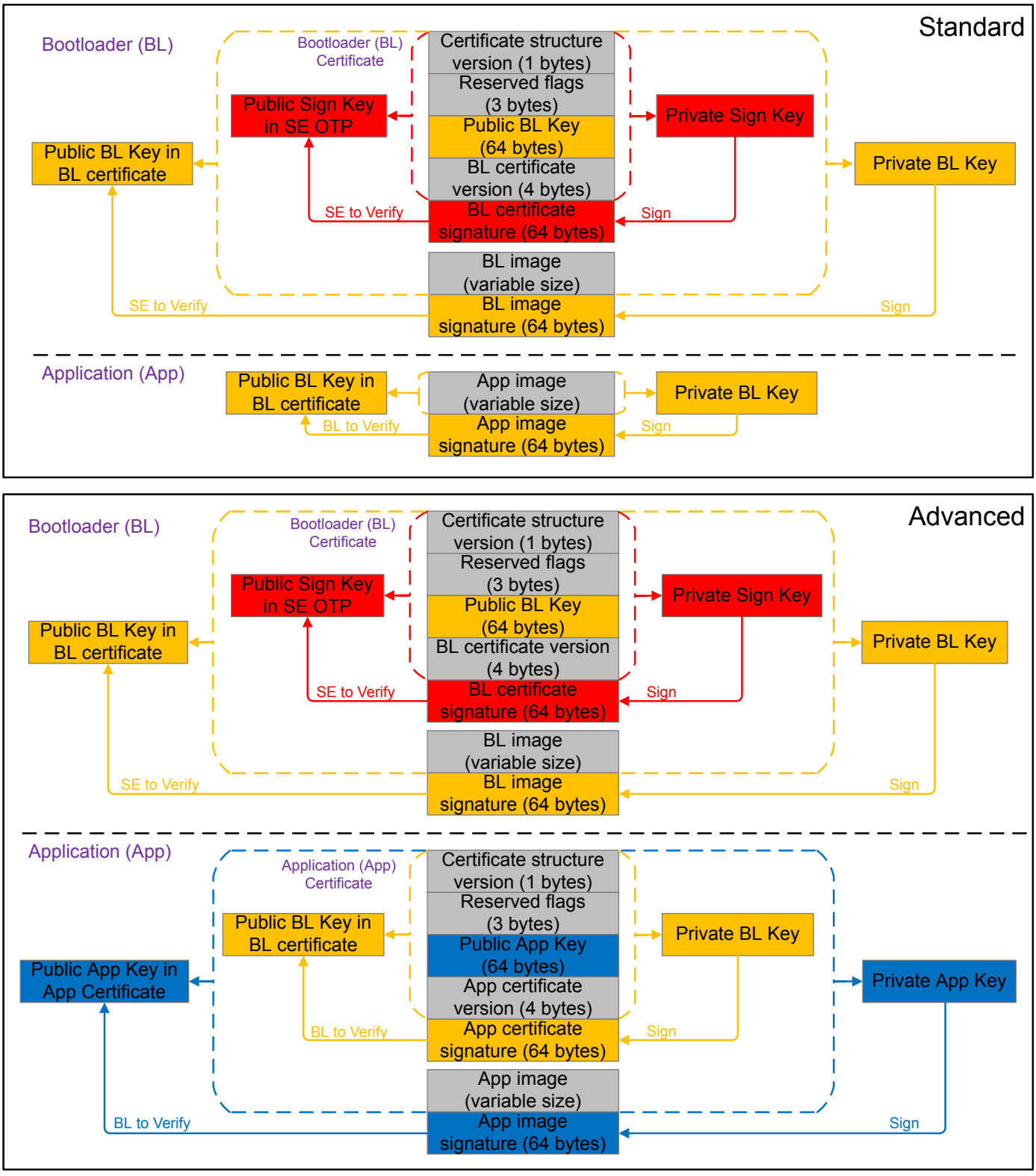


Figure 3.30. Certificate-Based Sign and Verify

Certificate

The following table describes the elements of a certificate.

Table 3.6. Certificate Structure

Element	Description
Certificate structure version	The version of the certificate structure.
Reserved flags	Reserved in the current certificate structure version.
Certificate public key	ECDSA-P256 public key, X and Y coordinates concatenated, used to validate the image.
Certificate version	The version of the running certificate.
Certificate signature	ECDSA-P256 signature, used for the authentication of the public key and the certificate version.

Note:

- The `application_properties.h` in the Windows folder below defines the parameters of the certificate structure (`ApplicationCertificate_t`).

  For GSDK v3.2 and lower: `C:\SiliconLabs\SimplicityStudio\v5\developer\sdks\gecko_sdk_suite\<GSDK VERSION>\platform\bootloader\api`

  For GSDK v4.0 and higher: `C:\Users\<PC USER NAME>\SimplicityStudio\SDKs\gecko_sdk\platform\bootloader\api`
- The certificate is not in X.509 format.

Private/Public Key Pair

The following table describes two Private/Public Key pairs used in certificates for certificate-based Secure Boot. The user can use Simplicity Commander or HSM to generate these key pairs.

Table 3.7. Certificates and Key Pairs for Certificate-Based Secure Boot Examples

Certificate	Private Key	Public Key	Description
Bootloader (bl_cert.bin) (1)	bl_cert_key.pem (Private Bootloader Key)	bl_cert_pubkey.pem (Public Bootloader Key)	The bootloader certificate is signed by the Private Sign Key corresponding to the Public Sign Key in SE OTP.
Application (app_cert.bin) (2)	app_cert_key.pem (Private Application Key)	app_cert_pubkey.pem (Public Application Key)	The application certificate is signed by the Private Bootloader Key corresponding to the Public Bootloader Key in the bootloader certificate.

Note:

1. a. Certificate version in the bootloader certificate < certificate version in SE flash - the certificate is rejected.
   b. Certificate version in the bootloader certificate = certificate version in SE flash - the certificate is accepted.
   c. Certificate version in the bootloader certificate > certificate version in SE flash - the certificate is accepted. The certificate version in SE flash is updated to match (revocation mechanism).
2. The certificate version in the application certificate is compared with the certificate version in the bootloader certificate. The application certificate is accepted if its version is equal to or higher than the certificate version in the bootloader certificate.

The following sections provide two methods to sign the bootloader image and application image files. All procedures assume the required files are in the same folder.

1. Using Simplicity Commander
2. Using an HSM and Simplicity Commander

Bootloader Image File

1. If the SE OTP is not provisioned, follow the procedures in 3.2.1 Simplicity Commander to set up the certificate-based Secure Boot configuration for the bootloader.
2. Follow the procedures in 3.4.1.1 AppBuilder or 3.4.1.2 Bootloader-core Software Component to set up the certificate-based Secure Boot configuration for the user application to generate an unsigned bootloader image.
3. **(Using Simplicity Commander)** Run the `util gencert` command with **Public Bootloader Key** and **Private Sign Key** to generate the bootloader certificate (`bl_cert.bin`). Refer to Table 3.7 Certificates and Key Pairs for Certificate-Based Secure Boot Examples on page 53 for details about the `--cert-version` for bootloader certificate.

```
commander util gencert --cert-type secureboot --cert-version 1 --cert-pubkey bl_cert_pubkey.pem
--sign sign_key.pem --outfile bl_cert.bin
```

```
Successfully signed certificate
DONE
```

Run the `convert` command with **Bootloader Certificate** and **Private Bootloader Key** to overwrite the unsigned bootloader image file with the signed bootloader image file (`bootloader-uart-xmodem.s37`).

```
commander convert bootloader-uart-xmodem.s37 --secureboot --certificate bl_cert.bin
--keyfile bl_cert_key.pem --outfile bootloader-uart-xmodem.s37
```

```
Parsing file bootloader-uart-xmodem.s37...
Writing certificate to location 0x00002cf0
Private key matches public key in certificate.
Found Application Properties at 0x00002d78
Writing Application Properties signature pointer to point to 0x00002f64
Setting signature type in Application Properties: 0x00000001
Image SHA256: 3cf574b688853a801e8dc98687414db27f886c60c55dbf7fea2d47633df94e8d
R = C866592B4CB7BAD9EFC35985F1B9D52C65C26453D4808597EEEFFB16DC4AA962
S = 94CAA21ED5D7772F96BBF4D24A0711A94DCCB6D4D38DFA45182876B9BE2A8DE3

Verifying signed image...
Writing to bootloader-uart-xmodem.s37...
Overwriting file: bootloader-uart-xmodem.s37...
DONE
```

4. **(Using an HSM and Simplicity Commander)** Run the `util gencert` command with **Public Bootloader Key** and **--extsign** option to generate an external signing bootloader certificate (`bl_cert.extsign`). Refer to Table 3.7 Certificates and Key Pairs for Certificate-Based Secure Boot Examples on page 53 for details about the `--cert-version` for bootloader certificate.

```
commander util gencert --cert-type secureboot --cert-version 1 --cert-pubkey bl_cert_pubkey.pem
--extsign --outfile bl_cert
```

```
DONE
```

Use an HSM containing the Private Sign Key to generate the signature for the external signing bootloader certificate. This example uses the OpenSSL with the **Private Sign Key** to simulate this process. The signature is in the `bl_cert_signature.der`.

```
openssl dgst -sha256 -binary -sign sign_key.pem -out bl_cert_signature.der bl_cert.extsign
```

Run the `util signcert` command with the **bootloader certificate signature** to generate the bootloader certificate (`bl_cert.bin`).

```
commander util signcert bl_cert.extsign --cert-type secureboot --signature bl_cert_signature.der
--verify sign_pubkey.pem --outfile bl_cert.bin
```

```
R = 065A58EA6CE6BBA44F3C59C6D255A901DBBC55FA97F261658B2026ABC8CD9680
S = 8A0011AA6393BC284B13C8313EE6772030DE07E213E74CA0FEA740F3D33E6518
Successfully verified signature
Successfully signed certificate
DONE
```

Run the `convert` command with **Bootloader Certificate** and **--extsign** option to generate an external signing bootloader image file (`bootloader-uart-xmodem.extsign`).

```
commander convert bootloader-uart-xmodem.s37 --secureboot --certificate bl_cert.bin --extsign
--outfile bootloader-uart-xmodem
```

```
Parsing file bootloader-uart-xmodem.s37...
Writing certificate to location 0x00002cf0
Found Application Properties at 0x00002d78
Writing Application Properties signature pointer to point to 0x00002f64
Setting signature type in Application Properties: 0x00000001
Writing to bootloader-uart-xmodem.extsign...
DONE
```

Use an HSM containing the Private Bootloader Key to generate the signature for the external signing bootloader image. This example uses the OpenSSL with the **Private Bootloader Key** to simulate this process. The signature is in the `bl_signature.der`.

```
openssl dgst -sha256 -binary -sign bl_cert_key.pem -out bl_signature.der bootloader-uart-xmodem.extsign
```

Run the `convert` command with the **Bootloader Certificate** and **bootloader image signature** to overwrite the unsigned bootloader image file with the signed bootloader image file (`bootloader-uart-xmodem.s37`).

```
commander convert bootloader-uart-xmodem.s37 --secureboot --certificate bl_cert.bin
--signature bl_signature.der --outfile bootloader-uart-xmodem.s37
```

```
Parsing file bootloader-uart-xmodem.s37...
Writing certificate to location 0x00002cf0
Parsing signature file bl_signature.der...
R = 7AAA17E8F875F2572AF7692079ED2C48A1329D3AA2E616E7E13007CE68C32A21
S = 6E5F1ACD929A1AC05DE9F4BC5CBDE0E076B32EDA353C5E571C7850AAB86BCCFB
Found Application Properties at 0x00002d78
Writing Application Properties signature pointer to point to 0x00002f64
Setting signature type in Application Properties: 0x00000001

Verifying signed image...
Writing to bootloader-uart-xmodem.s37...
Overwriting file: bootloader-uart-xmodem.s37...
DONE
```

5. **(Optional)** Run the `util verifysign` command with **Public Sign Key** to verify that the **Bootloader Certificate and image** were correctly signed.

```
commander util verifysign bootloader-uart-xmodem.s37 --verify sign_pubkey.pem
```

```
Parsing file bootloader-uart-xmodem.s37...
Found application properties at 0x00002d78
Found certificate at 0x00002cf0
Successfully verified certificate signature with verification key.
Using certificate key to verify application signature.
Successfully verified application signature.
DONE
```

6. **(Optional)** Run the `util appinfo` command to check all available information about `ApplicationProperties_t` struct in a signed GBL image.

```
commander util appinfo bootloader-uart-xmodem.s37
```

```
Parsing file bootloader-uart-xmodem.s37...
Found application properties in image.
Application properties info:
Application properties location : 0x00002d00
Signature location              : 0x00002e14
Signature type                  : ECDSA-P256
Long token section address      : Not set (0x00000000)

Application data info:
For Series 2 devices: If rollback prevention is enabled in the OTP configuration, the device will not boot
if the device has seen an application with a higher version number.
App type                        : Bootloader (APPLICATION_TYPE_BOOTLOADER)
App version                     : 0x02000000
Product ID                      : Not set (0x00000000000000000000000000000000)

Found certificate at 0x00002c78
Application certificate info:
Certificate located at          : 0x00002c78
Certificate version             : 0x00000001
Certificate key                 : 0xb1bc6f6fa56640ed522b2ee0f5b3cf7e5d48f60be8148f0dc08440f0a4e1dca4
                                  7c04119ed6a1be31b7707e5f9d001a659a051003e95e1b936f05c37ea793ad63
Certificate signature           : 0xef3b53368d4cd7821eb30a96140bbde8840378cfea30687a8c10642e1c7728fd
                                  309f976adf46e4eac62a2233f0c1f08f4e58344bdec61775b5282ceb351bb3d0
DONE
```

7. The signed bootloader image file (`.s37`) can be used for production programming or for generating a GBL upgrade image file for bootloader upgrade.
8. Run the `flash` command to program the signed bootloader image (`bootloader-uart-xmodem.s37`) to the device if the device does not have a bootloader.

```
commander flash bootloader-uart-xmodem.s37 --device EFR32MG21A010F1024 --serialno 440048205
```

```
Parsing file bootloader-uart-xmodem.s37...
Writing 16384 bytes starting at address 0x00000000
Comparing range 0x00000000 - 0x00003FFF (16 KiB)
Programming range 0x00000000 - 0x00001FFF (8 KiB)
Programming range 0x00002000 - 0x00003FFF (8 KiB)
DONE
```

Application Image File (Standard Certificate-Based)

1. Follow the procedures in 3.4.2 Generate an Unsigned Application Image to generate an unsigned application image for the bootloader.

2. **(Using Simplicity Commander)** Run the `convert` command with **Private Bootloader Key** to overwrite the unsigned application image file with the signed application image file (`blink_baremetal.s37`).

```
commander convert blink_baremetal.s37 --secureboot --keyfile bl_cert_key.pem --verify bl_cert_pubkey.pem
--outfile blink_baremetal.s37
```

```
Parsing file blink_baremetal.s37...
Found Application Properties at 0x000061bc
Writing Application Properties signature pointer to point to 0x000064d8
Setting signature type in Application Properties: 0x00000001
Image SHA256: 8b58ec567126aa1f6baa88afc916581477745aca6f47697ec093512fc30dcc6f
R = 994739A26AB520A88A5550F1643AE263D88A952F185F96EE7021FA43DEA6138C
S = 65B7112715E2F999A6B216C32D3331AB63B2D31A0A1311DF36EEE62269F8D6AA

Verifying signed image...
Writing to blink_baremetal.s37...
Overwriting file: blink_baremetal.s37...
DONE
```

3. **(Using an HSM and Simplicity Commander)** Run the `convert` command with **--extsign** option to generate an external signing application image file (`blink_baremetal.extsign`).

```
commander convert blink_baremetal.s37 --secureboot --extsign --outfile blink_baremetal
```

```
Parsing file blink_baremetal.s37...
Found Application Properties at 0x000061bc
Writing Application Properties signature pointer to point to 0x000064d8
Setting signature type in Application Properties: 0x00000001
Writing to blink_baremetal.extsign...
DONE
```

Use an HSM containing the Private Bootloader Key to generate the signature for the external signing application image. This example uses the OpenSSL with the **Private Bootloader Key** to simulate this process. The signature is in the `app_signature.der`.

```
openssl dgst -sha256 -binary -sign bl_cert_key.pem -out app_signature.der blink_baremetal.extsign
```

Run the `convert` command with the **application image signature** to overwrite the unsigned application image file with the signed application image file. (`blink_baremetal.s37`).

```
commander convert blink_baremetal.s37 --secureboot --signature app_signature.der
--verify bl_cert_pubkey.pem --outfile blink_baremetal.s37
```

```
Parsing file blink_baremetal.s37...
Parsing signature file app_signature.der...
R = 8DA79B020E954D24C23423D80627E046E44052736F6546902F016D64464E82DE
S = 9D5A1CC424E97A5AD0352A4EEA6BBF565FED5FC61FF99E63AA73DFFEAD9EE399
Found Application Properties at 0x000061bc
Writing Application Properties signature pointer to point to 0x000064d8
Setting signature type in Application Properties: 0x00000001

Verifying signed image...
Writing to blink_baremetal.s37...
Overwriting file: blink_baremetal.s37...
DONE
```

4. **(Optional)** Run the `util verifysign` command with **Public Bootloader Key** to verify that the application image file was correctly signed.

```
commander util verifysign blink_baremetal.s37 --verify bl_cert_pubkey.pem
```

```
Parsing file blink_baremetal.s37...
Found application properties at 0x000061bc
Did not find application certificate in file
If the configuration flag SECURE_BOOT_VERIFY_CERTIFICATE is set or a device has previously seen
certificate based signing, it will not accept direct signing.
Successfully verified application signature.
DONE
```

5. **(Optional)** Run the `util appinfo` command to check all available information about `ApplicationProperties_t` struct in a signed application image.

```
commander util appinfo blink_baremetal.s37
```

```
Parsing file blink_baremetal.s37...
Found application properties in image.
Application properties info:
Application properties location : 0x000061bc
Signature location              : 0x000064d8
Signature type                  : ECDSA-P256
Long token section address      : Not set (0x00000000)

Application data info:
For Series 2 devices: If rollback prevention is enabled in the OTP configuration, the device will not boot
if the device has seen an application with a higher version number.
App type                        : MCU application (APPLICATION_TYPE_MCU)
App version                     : 0x00000001
Product ID                      : Not set (0x00000000000000000000000000000000)

No certificate found in image.
For Series 2 devices: If the configuration flag SECURE_BOOT_VERIFY_CERTIFICATE is set or a device has
previously seen certificate based signing, it will not accept direct signing.
DONE
```

6. The signed application image file (`.s37`) can be used for production programming or for generating a GBL upgrade image file for application upgrade.

Application Image File (Advanced Certificate-Based)

1. Follow the procedures in 3.4.2 Generate an Unsigned Application Image to generate an unsigned application image for the bootloader.

2. **(Using Simplicity Commander)** Run the `util gencert` command with **Public Application Key** and **Private Bootloader Key** to generate the application certificate (`app_cert.bin`). Refer to Table 3.7 Certificates and Key Pairs for Certificate-Based Secure Boot Examples on page 53 for details about the `--cert-version` for application certificate.

```
commander util gencert --cert-type secureboot --cert-version 1 --cert-pubkey app_cert_pubkey.pem
--sign bl_cert_key.pem --outfile app_cert.bin
```

```
Successfully signed certificate
DONE
```

Run the `convert` command with **Application Certificate** and **Private Application Key** to overwrite the unsigned application image file with the signed application image file. (`blink_baremetal.s37`). This command will inject the application certificate into the application image before signing.

```
commander convert blink_baremetal.s37 --secureboot --certificate app_cert.bin --keyfile app_cert_key.pem
--outfile blink_baremetal.s37
```

```
Parsing file blink_baremetal.s37...
Writing certificate to location 0x000064d8
Private key matches public key in certificate.
Found Application Properties at 0x000061bc
Writing Application Properties signature pointer to point to 0x00006560
Setting signature type in Application Properties: 0x00000001
Image SHA256: 38fd11214c36abf3bb4c4eeda8cfdd2ca2ac2ff1e07072d555a06c74700a23f5
R = 6B4E3BB454513CAA4569415AE8F79453973AAC7FD1FC4914284B65010F3790A6
S = 1657CAAABED5798801872610383 58C83B1780A67CC41475370D94ED4445A5557

Verifying signed image...
Writing to blink_baremetal.s37...
Overwriting file: blink_baremetal.s37...
DONE
```

3. **(Using an HSM and Simplicity Commander)** Run the `util gencert` command with **Public Application Key** and **--extsign** option to generate an external signing application certificate (`app_cert.extsign`). Refer to Table 3.7 Certificates and Key Pairs for Certificate-Based Secure Boot Examples on page 53 for details about the `--cert-version` for application certificate.

```
commander util gencert --cert-type secureboot --cert-version 1 --cert-pubkey app_cert_pubkey.pem
--extsign --outfile app_cert
```

```
DONE
```

Use an HSM containing the Private Bootloader Key to generate the signature for the external signing application certificate. This example uses the OpenSSL with the **Private Bootloader Key** to simulate this process. The signature is in the `app_cert_signature.der`.

```
openssl dgst -sha256 -binary -sign bl_cert_key.pem -out app_cert_signature.der app_cert.extsign
```

Run the `util signcert` command with the **application certificate signature** to generate the application certificate (`app_cert.bin`).

```
commander util signcert app_cert.extsign --cert-type secureboot --signature app_cert_signature.der
--verify bl_cert_pubkey.pem --outfile app_cert.bin
```

```
R = 279D4FA1B801D108F82E30B0CF1164BF597549287290BD3883C5847B91095CCE
S = 567F0E219D2089EF4D79C3D94E43D2FADFE1899B71492ED358E6A1B46AE8162F
Successfully verified signature
Successfully signed certificate
DONE
```

Run the `convert` command with the **Application Certificate** and **--extsign** option to generate an external signing application image file (`blink_baremetal.extsign`).

```
commander convert blink_baremetal.s37 --secureboot --certificate app_cert.bin --extsign --outfile blink_baremetal
```

```
Parsing file blink_baremetal.s37...
Writing certificate to location 0x000064d8
Found Application Properties at 0x000061bc
Writing Application Properties signature pointer to point to 0x00006560
Setting signature type in Application Properties: 0x00000001
Writing to blink_baremetal.extsign...
DONE
```

Use an HSM containing the Private Application Key to generate the signature for the external signing application image. This example uses the OpenSSL with the **Private Application Key** to simulate this process. The signature is in the `app_signature.der`.

```
openssl dgst -sha256 -binary -sign app_cert_key.pem -out app_signature.der blink_baremetal.extsign
```

Run the `convert` command with the **Application Certificate** and **application image signature** to overwrite the unsigned application image file with the signed application image file. (`blink_baremetal.s37`). This command will inject the application certificate into the application image before signing.

```
commander convert blink_baremetal.s37 --secureboot --certificate app_cert.bin --signature app_signature.der
--verify app_cert_pubkey.pem --outfile blink_baremetal.s37
```

```
Parsing file blink_baremetal.s37...
Writing certificate to location 0x000064d8
Parsing signature file app_signature.der...
R = ADC2BAB959EC76CB2806C4649277669CF6E62A92ABFBBF20E551971449A8FCE0
S = B6B80130F95F62671372D1B2E471C82ADB08946C1F8938EED11F9822CE763A54
Found Application Properties at 0x000061bc
Writing Application Properties signature pointer to point to 0x00006560
Setting signature type in Application Properties: 0x00000001

Verifying signed image...

Verifying signed image...
Writing to blink_baremetal.s37...
Overwriting file: blink_baremetal.s37...
DONE
```

4. **(Optional)** Run the `util verifysign` command with **Public Bootloader Key** to verify that the **Application Certificate and image** were correctly signed.

```
commander util verifysign blink_baremetal.s37 --verify bl_cert_pubkey.pem
```

```
Parsing file blink_baremetal.s37...
Found application properties at 0x000061bc
Found certificate at 0x000064d8
Successfully verified certificate signature with verification key.
Using certificate key to verify application signature.
Successfully verified application signature.
DONE
```

5. **(Optional)** Run the `util appinfo` command to check all available information about `ApplicationProperties_t` struct in a signed application image.

```
commander util appinfo blink_baremetal.s37
```

```
Parsing file blink_baremetal.s37...
Found application properties in image.
Application properties info:
Application properties location : 0x000061bc
Signature location              : 0x00006560
Signature type                  : ECDSA-P256
Long token section address      : Not set (0x00000000)

Application data info:
For Series 2 devices: If rollback prevention is enabled in the OTP configuration, the device will not boot
if the device has seen an application with a higher version number.
App type                        : MCU application (APPLICATION_TYPE_MCU)
App version                     : 0x00000001
Product ID                      : Not set (0x00000000000000000000000000000000)

Found certificate at 0x000064d8
Application certificate info:
Certificate located at          : 0x000064d8
Certificate version             : 0x00000001
Certificate key                 : 0xe562003cd86e225decfd35712e431a19ecd5031a079b06c1d473620a6be9f57a
                                  879820100fee074f28b5885fd6759f480b62aaa0717f96e245aab6635cfb1e11
Certificate signature           : 0x039aaba62b5258e68d16e167c3a611c719c542bb3483f5d4b522472b06adf30f
                                  8cfcc484bf8551a208256e3d2d8c9194a7d2ac551e2cac659a99822308a40aa6
DONE
```

6. The signed application image file (`.s37`) can be used for production programming or for generating a GBL upgrade image file for application upgrade.

3.4.5 Generate a GBL Upgrade Image File

This section describes how to generate the GBL upgrade image files for updating the bootloader, Secure Engine, and application firmware.

Note:

1. Following the procedures in 3.4.1.1 AppBuilder or 3.4.1.2 Bootloader-core Software Component to avoid overwriting the existing application image (if necessary) when upgrading the bootloader or SE.
2. For a standalone bootloader with communication interface, the user can only generate three separate GBL upgrade image files containing bootloader, SE, and application images.
3. For an application bootloader with storage, the user can generate a single GBL upgrade image file (see example below) with a combination of bootloader, SE, and application images.

```
commander gbl create all.gbl --app app.s37 --bootloader bl.s37 --seupgrade se.seu
```

4. A signed GBL upgrade image file is required if the user enables the **Require signed firmware upgrade files** option in 3.4.1.1 AppBuilder or 3.4.1.2 Bootloader-core Software Component. The following table shows which private key(s) can be used to sign the GBL upgrade image file (bootloader, SE, or application) on HSE and VSE devices. The VSE devices store a Public Sign Key copy on the top page of the main flash to verify the GBL upgrade image file for ECDSA-P256-SHA256 Secure Boot.

Table 3.8. Private Key(s) usage to sign GBL upgrade image file

Secure Boot	HSE	VSE
ECDSA-P256-SHA256	Private Sign Key	Private Sign Key (Public Sign Key in main flash)
Certificate-Based	Private Sign Key or Private Bootloader Key	Private Bootloader Key

5. An encrypted GBL upgrade image file is required if the user enables the **Require encrypted firmware upgrade files** option in 3.4.1.1 AppBuilder or 3.4.1.2 Bootloader-core Software Component. Refer to 3.3 Provision GBL Decryption Key on how to provision the GBL Decryption Key for this option.
6. For an application bootloader with storage, the user can enable the **Upgrade SE without using the staging area** option in **GSDK v4.1.1 or higher** to directly fetch the SE image from the GBL upgrade image file in storage instead of copying the image to the pre-configured upgrade location.

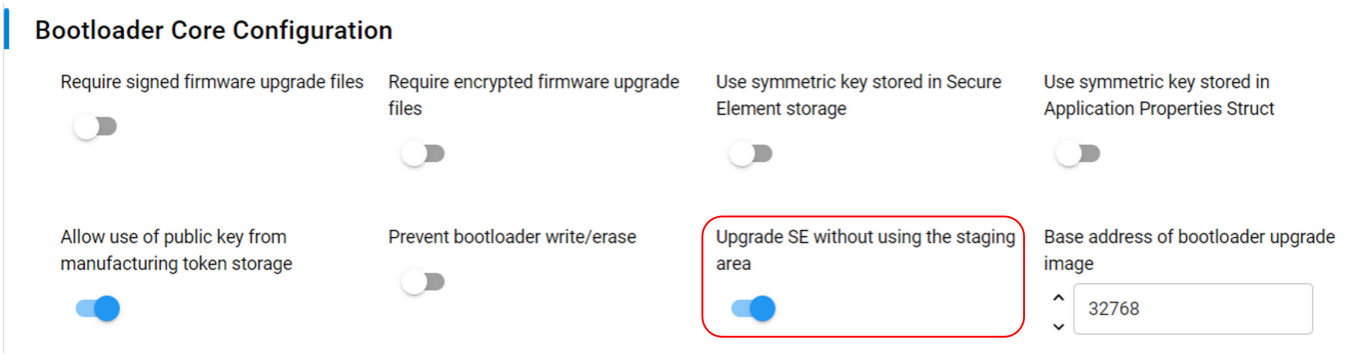


Figure 3.31. Upgrade SE without using the staging area option

To use the above option, the SE image cannot be in the encrypted part of the GBL upgrade image file if the **Require encrypted firmware upgrade files** option is enabled. Use the `--seunencrypted` option in **Simplicity Commander v1.13.0 or higher** (see example below) to generate an encrypted GBL upgrade image file with a SE image outside the encrypted part of the file.

```
commander gbl create se-upgrade.gbl --seupgrade secure-element.seu --seunencrypted --app myapp.s37
--encrypt aes_key.txt
```

The following sections provide two methods to sign the bootloader, Secure Engine, and application upgrade image files if the **Require signed firmware upgrade files** option is enabled.

1. Using Simplicity Commander
2. Using an HSM and Simplicity Commander

The sections also include encryption examples with an AES-128 key (like `aes_key.txt`) for the **Require encrypted firmware upgrade files** option. All procedures assume the required files are in the same folder.

Bootloader Upgrade

1. **(Unsigned)** Run the `gbl create` command with **--bootloader** option to generate the bootloader GBL upgrade image file (`bootloader-uart-xmodem.gbl`) with the signed bootloader image file (`bootloader-uart-xmodem.s37`) from 3.4.3 Signing for ECDSA-P256-SHA256 Secure Boot or 3.4.4 Signing for Certificate-Based Secure Boot.

   Without encryption:

```
commander gbl create bootloader-uart-xmodem.gbl --bootloader bootloader-uart-xmodem.s37
```

```
Initializing GBL file...
Adding bootloader to GBL...
Writing GBL file bootloader-uart-xmodem.gbl...
DONE
```

   With encryption:

```
commander gbl create bootloader-uart-xmodem.gbl --bootloader bootloader-uart-xmodem.s37
--encrypt aes_key.txt
```

```
Initializing GBL file...
Adding bootloader to GBL...
Encrypting GBL...
Writing GBL file bootloader-uart-xmodem.gbl...
DONE
```

2. **(Signed - Using Simplicity Commander)** Run the `gbl create` command with **--bootloader** option to generate the signed bootloader GBL upgrade image file (`bootloader-uart-xmodem.gbl`) with **Private Sign Key** or **Private Bootloader Key** and the signed bootloader image file (`bootloader-uart-xmodem.s37`) from 3.4.3 Signing for ECDSA-P256-SHA256 Secure Boot or 3.4.4 Signing for Certificate-Based Secure Boot.

   Without encryption:

```
commander gbl create bootloader-uart-xmodem.gbl --bootloader bootloader-uart-xmodem.s37 --sign sign_key.pem
```

```
commander gbl create bootloader-uart-xmodem.gbl --bootloader bootloader-uart-xmodem.s37
--sign bl_cert_key.pem
```

```
Initializing GBL file...
Adding bootloader to GBL...
Signing GBL...
Image SHA256: 3eb09993ffca5f9b34df3f38b65ab9d2f6619b828b014a186516016d4bbd80f7
R = C21E0C19254AC4F62374BBCA65DEBB42C7349384F5527330CD030A51DC2170F7
S = E1680C3670DE68D731086845E2726EF3BF07B96EB54AA2DB2F390F60BDB6DAB2
Writing GBL file bootloader-uart-xmodem.gbl...
DONE
```

   With encryption:

```
commander gbl create bootloader-uart-xmodem.gbl --bootloader bootloader-uart-xmodem.s37 --sign sign_key.pem
--encrypt aes_key.txt
```

```
commander gbl create bootloader-uart-xmodem.gbl --bootloader bootloader-uart-xmodem.s37
--sign bl_cert_key.pem --encrypt aes_key.txt
```

```
Initializing GBL file...
Adding bootloader to GBL...
Encrypting GBL...
Signing GBL...
Image SHA256: a2ef5e19e6b2ec327010af9fcb7de861c6a311987b7d2c39ef6439ca8b9999c4
R = B74972171109A05A9B1B45E1C8A2BCD57EAB1EA8F8A7936BBDB14CF6FA36B28C
S = 9B673083684B01C2D05BD702A4D29A6EDCF88C471C3FA8B71DDE00AE85861930
Writing GBL file bootloader-uart-xmodem.gbl...
DONE
```

3. **(Signed - Using an HSM and Simplicity Commander)** Run the `gbl create` command with **--bootloader** and **--extsign** options to generate an external signing bootloader GBL upgrade image file (`bootloader-uart-xmodem.extsign`) with the signed bootloader image file (`bootloader-uart-xmodem.s37`) from 3.4.3 Signing for ECDSA-P256-SHA256 Secure Boot or 3.4.4 Signing for Certificate-Based Secure Boot.

   Without encryption:

```
commander gbl create bootloader-uart-xmodem --bootloader bootloader-uart-xmodem.s37 --extsign
```

```
Initializing GBL file...
Adding bootloader to GBL...
Preparing GBL for external signing...
Writing GBL file bootloader-uart-xmodem.extsign...
DONE
```

   With encryption:

```
commander gbl create bootloader-uart-xmodem --bootloader bootloader-uart-xmodem.s37 --extsign
--encrypt aes_key.txt
```

```
Initializing GBL file...
Adding bootloader to GBL...
Encrypting GBL...
Preparing GBL for external signing...
Writing GBL file bootloader-uart-xmodem.extsign...
DONE
```

   Use an HSM containing the Private Sign Key or Private Bootloader Key to generate the signature for the external signing bootloader GBL upgrade image file. This example uses the OpenSSL with the **Private Sign Key** or **Private Bootloader Key** to simulate this process. The signature is in the `gbl_signature.der`.

```
openssl dgst -sha256 -binary -sign sign_key.pem -out gbl_signature.der bootloader-uart-xmodem.extsign
```

```
openssl dgst -sha256 -binary -sign bl_cert_key.pem -out gbl_signature.der bootloader-uart-xmodem.extsign
```

   Run the `gbl sign` command with the **signature** above to generate a signed bootloader GBL upgrade image file (`bootloader-uart-xmodem.gbl`).

```
commander gbl sign bootloader-uart-xmodem.extsign --signature gbl_signature.der --verify sign_pubkey.pem
--outfile bootloader-uart-xmodem.gbl
```

```
commander gbl sign bootloader-uart-xmodem.extsign --signature gbl_signature.der
--verify bl_cert_pubkey.pem --outfile bootloader-uart-xmodem.gbl
```

```
Reading GBL data from bootloader-uart-xmodem.extsign...
Parsing signature file gbl_signature.der...
R = 90F0A3C0D5D9ED2DC10EB3F55595FF21AB31307DC6283E3F3B7494A30FB741D4
S = 2765041F515A960F048CA250BFAB92031D4D1E569FB3F917C9329E7362C17B51
Writing signature to GBL...
Verifying GBL...
Successfully verified GBL signature
Writing GBL file bootloader-uart-xmodem.gbl...
DONE
```

4. Follow the procedures in 3.4.6 Upload a GBL Upgrade Image File to upgrade the bootloader with the bootloader GBL upgrade image file.

Secure Engine Upgrade

1. **(Unsigned)** Run the `gbl create` command with **--seupgrade** option to generate the SE GBL upgrade image file (`s2c1_se_fw_upgrade_1v2p9.gbl`) with the SE image file (`s2c1_se_fw_upgrade_1v2p9.seu`).

   Without encryption:

```
commander gbl create s2c1_se_fw_upgrade_1v2p9.gbl --seupgrade s2c1_se_fw_upgrade_1v2p9.seu
```

```
Initializing GBL file...
Adding Secure Element upgrade image to GBL...
Writing GBL file s2c1_se_fw_upgrade_1v2p9.gbl...
DONE
```

   With encryption:

```
commander gbl create s2c1_se_fw_upgrade_1v2p9.gbl --seupgrade s2c1_se_fw_upgrade_1v2p9.seu
--encrypt aes_key.txt
```

```
Initializing GBL file...
Adding Secure Element upgrade image to GBL...
Encrypting GBL...
Writing GBL file s2c1_se_fw_upgrade_1v2p9.gbl...
DONE
```

2. **(Signed - Using Simplicity Commander)** Run the `gbl create` command with **--seupgrade** option to generate the signed SE GBL upgrade image file (`s2c1_se_fw_upgrade_1v2p9.gbl`) with **Private Sign Key** or **Private Bootloader Key** and the SE image file (`s2c1_se_fw_upgrade_1v2p9.seu`).

   Without encryption:

```
commander gbl create s2c1_se_fw_upgrade_1v2p9.gbl --seupgrade s2c1_se_fw_upgrade_1v2p9.seu
--sign sign_key.pem
```

```
commander gbl create s2c1_se_fw_upgrade_1v2p9.gbl --seupgrade s2c1_se_fw_upgrade_1v2p9.seu
--sign bl_cert_key.pem
```

```
Initializing GBL file...
Adding Secure Element upgrade image to GBL...
Signing GBL...
Image SHA256: 599d7fc35996b4715441b642709ed262525d09d811d4726e423c0d605ec0f0bf
R = EF8EC2DDEDDF44DF88FEAD4ED0A9FDC6351B4D745D5A05BFB87204791871A525
S = FCB26EF005D97E8C5341153A210AE9927E1CF646A3E473FFB90DA8C857E6421F
Writing GBL file s2c1_se_fw_upgrade_1v2p9.gbl...
DONE
```

   With encryption:

```
commander gbl create s2c1_se_fw_upgrade_1v2p9.gbl --seupgrade s2c1_se_fw_upgrade_1v2p9.seu
--sign sign_key.pem --encrypt aes_key.txt
```

```
commander gbl create s2c1_se_fw_upgrade_1v2p9.gbl --seupgrade s2c1_se_fw_upgrade_1v2p9.seu
--sign bl_cert_key.pem --encrypt aes_key.txt
```

```
Initializing GBL file...
Adding Secure Element upgrade image to GBL...
Encrypting GBL...
Signing GBL...
Image SHA256: a5ab368c99c49503a7dfb6aef1724dc3f883eabddcac7b089148035483e24322
R = 289F05910A8E0735648260FA7A1C67731CA86FB2DFCB9B405EC8D297892915A7
S = F21D18351442D0E5F49CFBEDA2C0EFA8B7F0911B4B6216EB48250CB5889ECAFD
Writing GBL file s2c1_se_fw_upgrade_1v2p9.gbl...
DONE
```

3. **(Signed - Using an HSM and Simplicity Commander)** Run the `gbl create` command with **--seupgrade** and **--extsign** options to generate an external signing SE GBL upgrade image file (`s2c1_se_fw_upgrade_1v2p9.extsign`) with the SE image file (`s2c1_se_fw_upgrade_1v2p9.seu`).

   Without encryption:

```
commander gbl create s2c1_se_fw_upgrade_1v2p9 --seupgrade s2c1_se_fw_upgrade_1v2p9.seu --extsign
```

```
Initializing GBL file...
Adding Secure Element upgrade image to GBL...
Preparing GBL for external signing...
Writing GBL file s2c1_se_fw_upgrade_1v2p9.extsign...
DONE
```

   With encryption:

```
commander gbl create s2c1_se_fw_upgrade_1v2p9 --seupgrade s2c1_se_fw_upgrade_1v2p9.seu --extsign --encrypt
aes_key.txt
```

```
Initializing GBL file...
Adding Secure Element upgrade image to GBL...
Encrypting GBL...
Preparing GBL for external signing...
Writing GBL file s2c1_se_fw_upgrade_1v2p9.extsign...
DONE
```

   Use an HSM containing the Private Sign Key or Private Bootloader Key to generate the signature for the external signing SE GBL upgrade image file. This example uses the OpenSSL with the **Private Sign Key** or **Private Bootloader Key** to simulate this process. The signature is in the `gbl_signature.der`.

```
openssl dgst -sha256 -binary -sign sign_key.pem -out gbl_signature.der s2c1_se_fw_upgrade_1v2p9.extsign
```

```
openssl dgst -sha256 -binary -sign bl_cert_key.pem -out gbl_signature.der s2c1_se_fw_upgrade_1v2p9.extsign
```

   Run the `gbl sign` command with the **signature** above to generate a signed SE GBL upgrade image file (`s2c1_se_fw_upgrade_1v2p9.gbl`).

```
commander gbl sign s2c1_se_fw_upgrade_1v2p9.extsign --signature gbl_signature.der --verify sign_pubkey.pem
--outfile s2c1_se_fw_upgrade_1v2p9.gbl
```

```
commander gbl sign s2c1_se_fw_upgrade_1v2p9.extsign --signature gbl_signature.der
--verify bl_cert_pubkey.pem --outfile s2c1_se_fw_upgrade_1v2p9.gbl
```

```
Reading GBL data from s2c1_se_fw_upgrade_1v2p9.extsign...
Parsing signature file gbl_signature.der...
R = 2798B98194EE02717C738B5866ABD8D234D0F0E096E90495D371D2507D8E1C67
S = 19F2586E2C6177D6B4EEC708E006F67334C989D0398D4233C686C98ECB6992FB
Writing signature to GBL...
Verifying GBL...
Successfully verified GBL signature
Writing GBL file s2c1_se_fw_upgrade_1v2p9.gbl...
DONE
```

4. Follow the procedures in 3.4.6 Upload a GBL Upgrade Image File to upgrade the SE with the SE GBL upgrade image file.

Note:

- The `sign_key.pem`/`sign_pubkey.pem` key pair is for 3.4.3 Signing for ECDSA-P256-SHA256 Secure Boot, and the `bl_cert_key.pem`/`bl_cert_pubkey.pem` key pair is for 3.4.4 Signing for Certificate-Based Secure Boot.
- Trying to apply a lower version of the SE image file (`.seu`) to the device will be ignored.

Application Upgrade

1. **(Unsigned)** Run the `gbl create` command with **--app** option to generate the application GBL upgrade image file (`blink_baremetal.gbl`) with the signed application image file (`blink_baremetal.s37`) from 3.4.3 Signing for ECDSA-P256-SHA256 Secure Boot or 3.4.4 Signing for Certificate-Based Secure Boot.

   Without encryption:

```
commander gbl create blink_baremetal.gbl --app blink_baremetal.s37
```

```
Parsing file blink_baremetal.s37...
Initializing GBL file...
Adding application to GBL...
Writing GBL file blink_baremetal.gbl...
DONE
```

   With encryption:

```
commander gbl create blink_baremetal.gbl --app blink_baremetal.s37 --encrypt aes_key.txt
```

```
Parsing file blink_baremetal.s37...
Initializing GBL file...
Adding application to GBL...
Encrypting GBL...
Writing GBL file blink_baremetal.gbl...
DONE
```

2. **(Signed - Using Simplicity Commander)** Run the `gbl create` command with **--app** option to generate the signed application GBL upgrade image file (`blink_baremetal.gbl`) with **Private Sign Key** or **Private Bootloader Key** and the signed application image file (`blink_baremetal.s37`) from 3.4.3 Signing for ECDSA-P256-SHA256 Secure Boot or 3.4.4 Signing for Certificate-Based Secure Boot.

   Without encryption:

```
commander gbl create blink_baremetal.gbl --app blink_baremetal.s37 --sign sign_key.pem
```

```
commander gbl create blink_baremetal.gbl --app blink_baremetal.s37 --sign bl_cert_key.pem
```

```
Parsing file blink_baremetal.s37...
Initializing GBL file...
Adding application to GBL...
Signing GBL...
Image SHA256: 116c1be47d799ab75afc7b3f4c9a8023e5cd031103b1d28c578eebfaf1ad73d2
R = CE4D85C058301A2437440E00385D97E496F1D8B5CAFFB8C184F8A88B5266E3E9
S = 90BBF754EBC0AB343CC32AA06ADED85F9D12D1A67CA6608F9085137142000A40
Writing GBL file blink_baremetal.gbl...
DONE
```

   With encryption:

```
commander gbl create blink_baremetal.gbl --app blink_baremetal.s37 --sign sign_key.pem
--encrypt aes_key.txt
```

```
commander gbl create blink_baremetal.gbl --app blink_baremetal.s37 --sign bl_cert_key.pem
--encrypt aes_key.txt
```

```
Parsing file blink_baremetal.s37...
Initializing GBL file...
Adding application to GBL...
Encrypting GBL...
Signing GBL...
Image SHA256: 24092ed828e6fffc41e7ed40c046b80789ef2337da0d7373a15e59d27e07e0fc
R = 6143F307119402DC55C63220D54542B84EBEFB324963C63796A37B9845482B35
S = AE3644D59DF3A27F45B335CB4F79D2347364958E0F152AF745FB7042537D1B6A
Writing GBL file blink_baremetal.gbl...
DONE
```

3. **(Signed - Using an HSM and Simplicity Commander)** Run the `gbl create` command with **--app** and **--extsign** options to generate an external signing application GBL upgrade image file (`blink_baremetal.extsign`) with the signed application image file (`blink_baremetal.s37`) from 3.4.3 Signing for ECDSA-P256-SHA256 Secure Boot or 3.4.4 Signing for Certificate-Based Secure Boot.

   Without encryption:

```
commander gbl create blink_baremetal --app blink_baremetal.s37 --extsign
```

```
Parsing file blink_baremetal.s37...
Initializing GBL file...
Adding application to GBL...
Preparing GBL for external signing...
Writing GBL file blink_baremetal.extsign...
DONE
```

   With encryption:

```
commander gbl create blink_baremetal --app blink_baremetal.s37 --extsign --encrypt aes_key.txt
```

```
Parsing file blink_baremetal.s37...
Initializing GBL file...
Adding application to GBL...
Encrypting GBL...
Preparing GBL for external signing...
Writing GBL file blink_baremetal.extsign...
DONE
```

   Use an HSM containing the Private Sign Key or Private Bootloader Key to generate the signature for the external signing application GBL upgrade image file. This example uses the OpenSSL with the **Private Sign Key** or **Private Bootloader Key** to simulate this process. The signature is in the `gbl_signature.der`.

```
openssl dgst -sha256 -binary -sign sign_key.pem -out gbl_signature.der blink_baremetal.extsign
```

```
openssl dgst -sha256 -binary -sign bl_cert_key.pem -out gbl_signature.der blink_baremetal.extsign
```

   Run the `gbl sign` command with the **signature** above to generate a signed application GBL upgrade image file (`blink_baremetal.gbl`).

```
commander gbl sign blink_baremetal.extsign --signature gbl_signature.der --verify sign_pubkey.pem
--outfile blink_baremetal.gbl
```

```
commander gbl sign blink_baremetal.extsign --signature gbl_signature.der --verify bl_cert_pubkey.pem
--outfile blink_baremetal.gbl
```

```
Reading GBL data from blink_baremetal.extsign...
Parsing signature file gbl_signature.der...
R = 533499660E24F1620EF25D862FB607F46E9E4ECC41CBDECBE77C64EF1970D96A
S = FA8901878218F5F1DB0FAF8B074CE98A27C63FFDE63730CD49EE47E847B9811D
Writing signature to GBL...
Verifying GBL...
Successfully verified GBL signature
Writing GBL file blink_baremetal.gbl...
DONE
```

4. Follow the procedures in 3.4.6 Upload a GBL Upgrade Image File to upgrade the application with the application GBL upgrade image file.

Note:

- The Simplicity Commander v1.11.0 or above supports GBL upgrade image file in `util verifysign` command.

```
commander util verifysign blink_baremetal.gbl --verify sign_pubkey.pem
```

```
Successfully verified GBL signature
DONE
```

- The Simplicity Commander v1.12.0 or above fixes a bug introduced in v1.11.0 when using the `--extsign` option on the GBL upgrade image file.

3.4.6 Upload a GBL Upgrade Image File

This section describes how to use UART XMODEM Bootloader v2.0.0 in GSDK v4.0 to upload a GBL upgrade image file (`.gbl`) to the device. The procedures and pictures may be different for the other versions of this example.

The GBL upgrade image file uses a proprietary format to store the upgrade image for a firmware upgrade. Use the `gbl create` command to generate the GBL upgrade image file for bootloader, application, and Secure Engine. Refer to UG266/UG489 and 3.4.5 Generate a GBL Upgrade Image File for more information about GBL upgrade image file creation.

The user can use any terminal software that supports the XMODEM-CRC protocol for file transfer. This application note uses Tera Term as terminal software. The default serial port setting is 115200 bps 8-N-1.

1. Assume the UART XMODEM Bootloader and application firmware had already flashed to the radio board on WSTK.
2. Press the RESET and PB0 push buttons on the WSTK.
3. Release the RESET push button to run the UART XMODEM Bootloader.

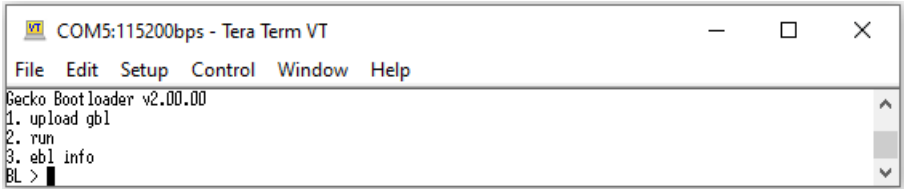


Figure 3.32. Options in UART XMODEM Bootloader

4. Release the PB0 push button. Press 1 (`upload gbl`) in Tera Term to upload a GBL upgrade image file.

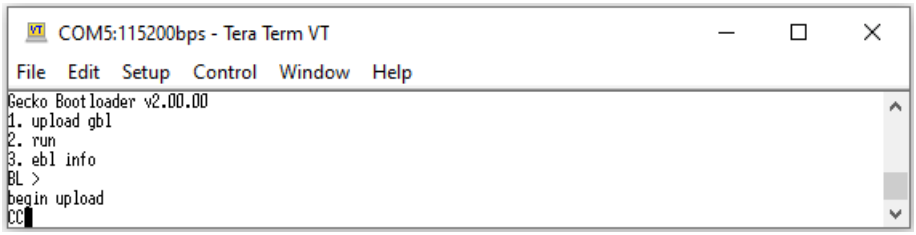


Figure 3.33. Upload GBL option

5. Transfer a file through XMODEM-CRC in Tera Term, navigate to **File** → **Transfer** → **XMODEM** → **Send...**.

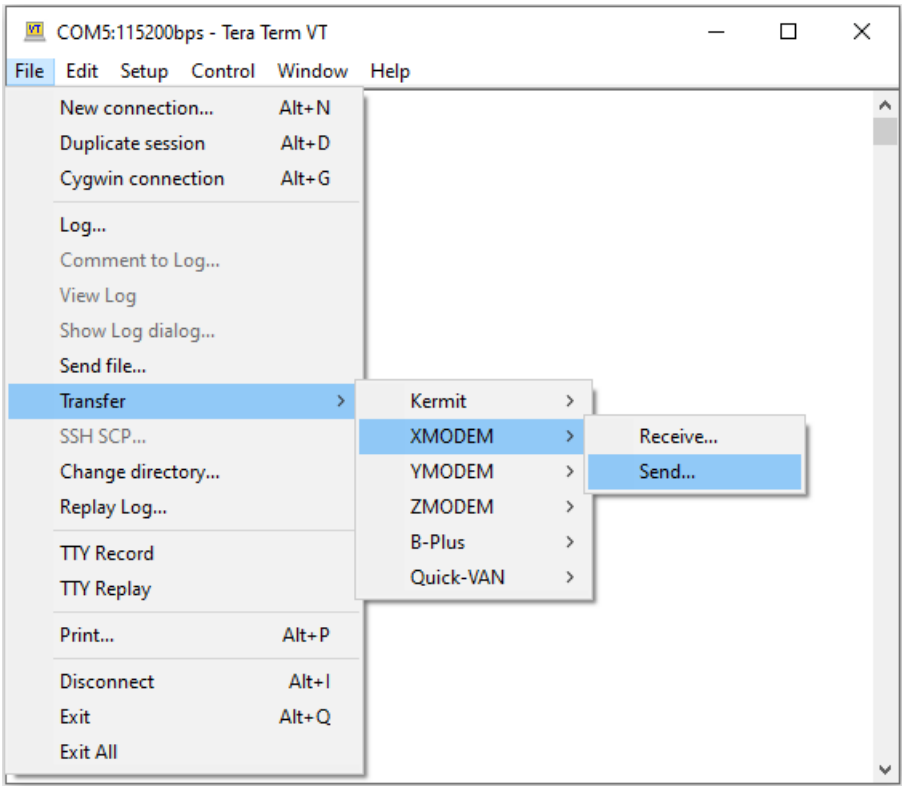


Figure 3.34. XMODEM Transfer in Tera Term

6. Select the target GBL upgrade image file. Click [**Open**] to upload.

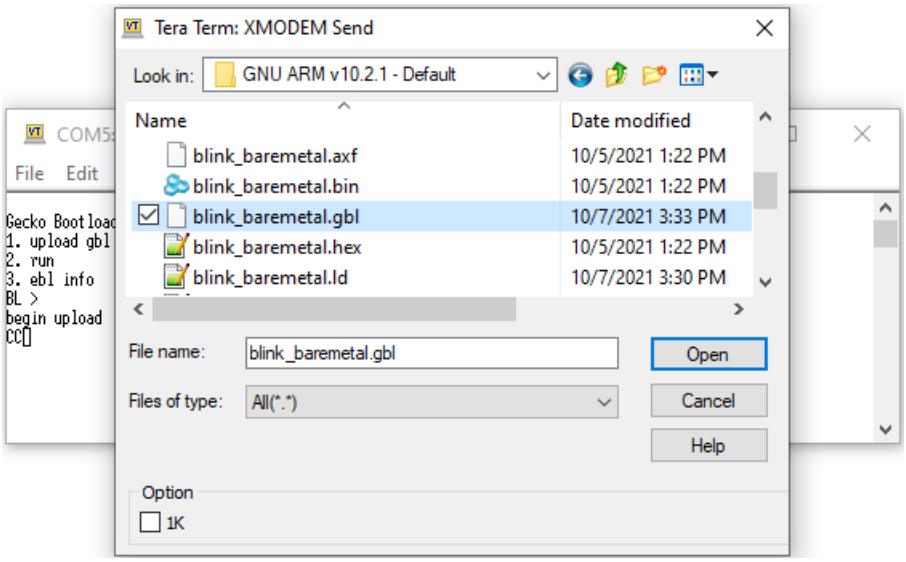


Figure 3.35. Target GBL file location

7. If no error occurs, press 2 (`run`) to start a firmware upgrade.

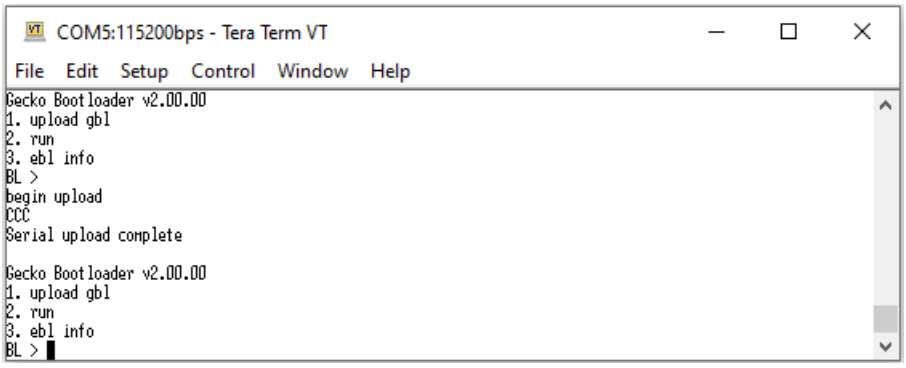


Figure 3.36. Run option

3.5 Upgrade to Certificate-Based Secure Boot

The user can upgrade the Series 2 devices deployed in the field from ECDSA-P256-SHA256 Secure Boot to certificate-based Secure Boot even the SECURE_BOOT_VERIFY_CERTIFICATE option in SE OTP is disabled.

```
commander security readconfig --serialno 440048205
```

```
MCU Flags
Secure Boot                    : Enabled
Secure Boot Verify Certificate : Disabled
Secure Boot Anti Rollback      : Enabled
Secure Boot Page Lock Narrow   : Disabled
Secure Boot Page Lock Full     : Disabled
```

The following procedures for the upgrade to certificate-based Secure Boot is an **IRREVERSIBLE** process.

1. Follow the procedures in 3.1.4 Generate Key and Signing to generate an ECDSA-P256 bootloader certificate key pair.
2. Follow the procedures in 3.4.4 Signing for Certificate-Based Secure Boot to generate the signed GBL image file with the bootloader certificate key pair in step 1. The bootloader certificate version (`--cert-version` in the `util gencert` command) in this signed GBL image file must be **equal to or higher than one (≥ 1)**.
3. Follow the procedures in 3.4.5 Generate a GBL Upgrade Image File to upgrade the bootloader to certificate-based Secure Boot. Use the Private Sign Key for ECDSA-P256-SHA256 Secure Boot to sign the bootloader GBL upgrade image file if required.

   SE will use the Public Bootloader Key to validate the bootloader image once SE identifies a bootloader certificate in the bootloader image. If the bootloader certificate version from step 2 is **higher than zero (> 0) and gets verified once**, SE will never again accept the ECDSA-P256-SHA256 Secure Boot signed bootloader image. Refer to the "*Secure Boot Procedure*" section in UG266/UG489 for more information.

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

   ```
   SE Firmware version : 1.2.9
   Serial number       : 00000000000000014b457fffe045a2d
   Debug lock          : Disabled
   Device erase        : Enabled
   Secure debug unlock : Disabled
   Tamper status       : Not OK
   Secure boot         : Enabled
   Boot status         : 0x18 - Failed: Secure Boot requires cert, but none found
   DONE
   ```

4. **(Standard Certificate-Based)** Follow the procedures in 3.4.4 Signing for Certificate-Based Secure Boot to generate the signed application image file with the Private Bootloader Key in step 1.
5. **(Advanced Certificate-Based)** Follow the procedures in 3.1.4 Generate Key and Signing to generate an ECDSA-P256 application certificate key pair.

   Follow the procedures in 3.4.4 Signing for Certificate-Based Secure Boot to generate the signed application image file with the application certificate key pair in this step and the Private Bootloader Key in step 1. The application certificate version (`--cert-version` in the `util gencert` command) in this signed application image file must be **equal to or higher** than the bootloader certificate version in step 2 (Table 3.7 Certificates and Key Pairs for Certificate-Based Secure Boot Examples on page 53).
6. Follow the procedures in 3.4.5 Generate a GBL Upgrade Image File to upgrade application with the signed image from step 4 or 5 for certificate-based Secure Boot. Use the Private Sign Key or Private Bootloader key in **step 1** for certificate-based Secure Boot to sign the application GBL upgrade image file if required.

3.6 Certificate Revocation

The certificate revocation is the act of invalidating a certificate when its private key shows signs of being compromised. The following procedures describe how to revoke the Series 2 devices' bootloader certificates deployed in the field.

1. Follow the procedures in 3.1.4 Generate Key and Signing to generate a new ECDSA-P256 bootloader certificate key pair.
2. Follow the procedures in 3.4.4 Signing for Certificate-Based Secure Boot to generate the signed GBL image file with the bootloader certificate key pair in step 1. The bootloader certificate version (`--cert-version` in the `util gencert` command) in this signed GBL image file must be **higher than** the certificate version in SE flash (Table 3.7 Certificates and Key Pairs for Certificate-Based Secure Boot Examples on page 53).
3. Follow the procedures in 3.4.5 Generate a GBL Upgrade Image File to upgrade the bootloader with the signed image from step 2. Use the Private Sign Key or **existing** Private Bootloader Key for certificate-based Secure Boot to sign the bootloader GBL upgrade image file if required.
4. **(Standard Certificate-Based)** Follow the procedures in 3.4.4 Signing for Certificate-Based Secure Boot to generate the signed application image file with the Private Bootloader Key in step 1.
5. **(Advanced Certificate-Based)** Follow the procedures in 3.4.4 Signing for Certificate-Based Secure Boot to generate the signed application image file with the Private Bootloader Key in step 1. The application certificate version (`--cert-version` in the `util gencert` command) in this signed application image file must be **equal to or higher** than the bootloader certificate version in step 2 (Table 3.7 Certificates and Key Pairs for Certificate-Based Secure Boot Examples on page 53).

   The user should generate a new ECDSA-P256 application certificate key pair if the Private Application Key for the application certificate is compromised.
6. Follow the procedures in 3.4.5 Generate a GBL Upgrade Image File to upgrade the application with the signed image from step 4 or 5. Use the Private Sign Key or Private Bootloader key in **step 1** for certificate-based Secure Boot to sign the application GBL upgrade image file if required.

3.7 Upgrade to Secure Boot with RTSL

The following procedures describe upgrading Series 2 devices deployed in the field without Secure Boot to Secure Boot with RTSL.

1. **(Recommended)** Upgrade SE firmware to the latest version if available. See the "*Gecko Bootloader Operation - Secure Engine Upgrade*" section in UG266/UG489.
2. Follow the procedures in 3.4.1.1 AppBuilder or 3.4.1.2 Bootloader-core Software Component to prepare an unsigned GBL image with the required Secure Boot configuration for the application firmware.
3. Follow the procedures in 3.1.4 Generate Key and Signing to generate the ECDSA-P256 Sign Key pair for Secure Boot. The key pairs for the bootloader certificate and application certificate (advanced) are required if using Certificate-Based Secure Boot.
4. Follow steps 1 to 2 in 3.3.2 SE Manager Key Provisioning Platform Example if this HSE GBL Decryption Key option is selected. Use the Public Sign Key in step 3 and follow steps 1 to 3 in 3.2.2 SE Manager Key Provisioning Platform Example to generate an **unsigned** image. Use this image to create an application GBL upgrade image file.
5. The original GBL (application Secure Boot is disabled) boots into the **unsigned** `SE Manager Key Provisioning Platform Example` after upgrading the application with the image file in step 4.
6. Follow steps 5 to 8 in SE Manager Key Provisioning Platform Example to install the Public Sign Key to SE OTP and GBL Decryption Key (optional) to HSE OTP. Press `SPACE` instead of `ENTER` in step 9 to **BYPASS** the programming of the Secure Boot configuration in SE OTP.

```
. Press ENTER to initialize SE OTP for secure boot configuration or press SPACE to skip.

. SE manager deinitialization... SL_STATUS_OK (cycles: 5 time: 0 us)
```

   Note:
   - Programming the Public Sign Key to the top page of the main flash (not included in this example) is required for the VSE device ECDSA-P256-SHA256 Secure Boot.
   - Programming the GBL Decryption Key to the top page of the main flash (not included in this example) is required if the default storage option for GBL Decryption Key is selected and the **Require encrypted firmware upgrade files** option is enabled in step 2.

7. Follow the signing procedures in 3.4.3 Signing for ECDSA-P256-SHA256 Secure Boot or 3.4.4 Signing for Certificate-Based Secure Boot (Bootloader Image File section, skip the Secure Boot configuration for the bootloader) with the required key(s) generated in step 3 to sign the unsigned GBL image generated from step 2. Use this signed image to create a bootloader GBL upgrade image file.
8. Follow the signing procedures in 3.4.3 Signing for ECDSA-P256-SHA256 Secure Boot or 3.4.4 Signing for Certificate-Based Secure Boot (Application Image File section) with the required key(s) generated in step 3 to sign the unsigned application image generated from step 4. Use this signed image to create an application GBL upgrade image file.

   Note: For the application bootloader with storage, the user can generate a single GBL upgrade image file for signed images from steps 7 and 8.

9. The Secure Boot in SE OTP is not yet enabled, so FSB does not verify the signature when upgrading to the signed GBL in step 7. The updated GBL (application Secure Boot enabled) verifies the signature when upgrading or booting to the signed `SE Manager Key Provisioning Platform Example` in step 8.
10. Follow steps 9 to 10 (use `SPACE` to skip previous steps for OTP key programming) in SE Manager Key Provisioning Platform Example to program the required Secure Boot configuration in SE OTP for signed GBL.
11. Update a **signed** custom application firmware to replace the signed `SE Manager Key Provisioning Platform Example` used for Secure Boot with RTSL upgrade.

Note:
- Refer to the "*Enabling Secure Boot RTSL on Series 2 Devices*" section (either Standalone Bootloaders or Application Bootloaders with Storage) in UG266/UG489 for details.
- The `SE Manager Key Provisioning Platform Example` used here is just for reference. The user can modify or write a new application to automate the processes for the Secure Boot with RTSL upgrade.
- If the Require signed firmware upgrade files option is enabled in step 2, the GBL upgrade image files from steps 8 and 11 must be signed.
- If the Require encrypted firmware upgrade files option is enabled in step 2, the GBL upgrade image files from steps 8 and 11 must be encrypted. And the GBL Decryption Key for the corresponding option in GBL must be in place.

3.8 Recover Devices when Secure Boot Fails

If a Secure Boot process fails (meaning firmware image at device starting address validation fails), the only way to recover is to flash a correctly signed image.

There are two scenarios to recover the device from a Secure Boot failure:

1. Debug Lock
2. BUSLOCK

3.8.1 DEBUG LOCK

The following table describes the different debug lock scenarios on recovering the Secure Boot failure device.

Table 3.9. Debug Lock Scenarios

Secure Debug	Device Erase	Debug Lock	State	Recover from Secure Boot Failure
Disabled	Enabled	Disabled	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: The error code in the **Boot status** of examples below depends on boot failure caused by the host image (GBL).

The following procedures describe how to recover the Secure Boot failure device from the lock states below.

- Unlocked
- Standard debug locked
- Secure debug locked

1. Follow the procedure in 3.4.3 Signing for ECDSA-P256-SHA256 Secure Boot or 3.4.4 Signing for Certificate-Based Secure Boot to generate a correctly signed GBL.
2. **(Unlocked)** Run the `security status` command to get the boot status.

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

```
SE Firmware version : 1.2.9
Serial number       : 00000000000000014b457fffe045afd
Debug lock          : Disabled
Device erase        : Enabled
Secure debug unlock : Disabled
Tamper status       : Not OK
Secure boot         : Enabled
Boot status         : 0x12 - Failed: Error while checking signature of host firmware
DONE
```

Run the `flash` command to flash the correctly signed image (like `bootloader-uart-xmodem.s37`). If a failed Secure Boot is detected, the device will be erased before flashing the new image.

```
commander flash bootloader-uart-xmodem.s37 --device EFR32MG21A010F1024 --serialno 440048205
```

```
WARNING: Failed secure boot detected. Issuing a mass erase before flashing to recover the device...
Parsing file bootloader-uart-xmodem.s37...
Writing 16384 bytes starting at address 0x00000000
Comparing range 0x00000000 - 0x00003FFF (16 KiB)
Programming range 0x00000000 - 0x00001FFF (8 KiB)
Programming range 0x00002000 - 0x00003FFF (8 KiB)
DONE
```

3. **(Standard debug locked)** Run the `security status` command to get the boot status.

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

```
SE Firmware version : 1.2.9
Serial number       : 000000000000000014b457fffe045afd
Debug lock          : Enabled
Device erase        : Enabled
Secure debug unlock : Disabled
Tamper status       : Not OK
Secure boot         : Enabled
Boot status         : 0x12 - Failed: Error while checking signature of host firmware
DONE
```

Run the `security erasedevice` command to unlock the 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.

Run the `flash` command to flash the correctly signed image (like `bootloader-uart-xmodem.s37`). If a failed Secure Boot is detected, the device will be erased before flashing the new image.

```
commander flash bootloader-uart-xmodem.s37 --device EFR32MG21A010F1024 --serialno 440048205
```

```
WARNING: Failed secure boot detected. Issuing a mass erase before flashing to recover the device...
Parsing file bootloader-uart-xmodem.s37...
Writing 16384 bytes starting at address 0x00000000
Comparing range 0x00000000 - 0x00003FFF (16 KiB)
Programming range 0x00000000 - 0x00001FFF (8 KiB)
Programming range 0x00002000 - 0x00003FFF (8 KiB)
DONE
```

4. **(Secure debug locked)** Run the `security status` command to get the boot status.

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

```
SE Firmware version : 1.2.9
Serial number       : 00000000000000000d6ffffe0a3a5f
Debug lock          : Enabled
Device erase        : Disabled
Secure debug unlock : Enabled
Tamper status       : Not OK
Secure boot         : Enabled
Boot status         : 0x12 - Failed: Error while checking signature of host firmware
DONE
```

Run the `security unlock` command to unlock the device with the debug unlock token.

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

```
Unlocking with unlock payload:
C:/Users/<username>/AppData/Local/SiliconLabs/commander/SecurityStore/
device_00000000000000000d6ffffe0a3a5f/challenge_020fc3cc9e492088d06d75d71b7aabfe/
unlock_payload_0000000000111110.bin
Secure debug successfully unlocked
DONE
```

Run the `flash` command with the **`--noreset`** option to flash the correctly signed image (like `bootloader-uart-xmodem.s37`).

```
commander flash --noreset bootloader-uart-xmodem.s37 --device EFR32MG21A010F1024 --serialno 440048205
```

```
Parsing file bootloader-uart-xmodem.s37...
Writing 16384 bytes starting at address 0x00000000
Comparing range 0x00000000 - 0x00003FFF (16 KiB)
Erasing range 0x00000000 - 0x00003FFF (2 sectors, 16 KiB)
Programming range 0x00000000 - 0x00001FFF (8 KiB)
Programming range 0x00002000 - 0x00003FFF (8 KiB)
DONE
```

Note: The `--noreset` option prevents the device from returning to the secure debug lock state before flashing.

5. Run the `security status` command to check the boot status. The example below is an unlocked device.

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

```
SE Firmware version : 1.2.9
Serial number       : 00000000000000014b457fffe045afd
Debug lock          : Disabled
Device erase        : Enabled
Secure debug unlock : Disabled
Tamper status       : OK
Secure boot         : Enabled
Boot status         : 0x20 - OK
DONE
```

3.8.2 BUSLOCK

When secure boot is enabled, the SE enforces the secure boot process through a hardware mechanism called BUSLOCK. This BUSLOCK will halt the host-side (M33) bus so that it will not start the executing code. The BUSLOCK is enabled out of reset when secure boot is enabled.

- **BUSLOCK status for secure boot success:** BUSLOCK is enabled out of reset and is released only when the Secure boot process is successful and returns 0x20 - OK status code.

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

```
SE Firmware version : 1.2.9
Serial number : 000000000000000014b457fffe045afd
Debug lock : Disabled
Device erase : Enabled
Secure debug unlock : Disabled
Tamper status : OK
Secure boot : Enabled
Boot status : 0x20 - OK
DONE
```

- **BUSLOCK status for secure boot failure:** BUSLOCK remains applied when the Secure Boot process fails and returns anything but 0x20 - OK status code. This keeps the BUSLOCK intact, and the host CPU is frozen, resulting in halting firmware execution on the M33 core. The device can be recovered from the BUSLOCK state by following these steps:
  1. Issue a ‘security erase’ command to the SE:

     ```
     commander security erasedevice
     ```

     ```
     Successfully erased device
     DONE
     ```
  2. After issuing the ‘security erase’ command to the SE, flash the correctly signed image to recover from a secure boot failure .

4. Debugging on Secure Boot Enabled Device

Assume a correctly signed GBL image has been programmed to the device. Follow the procedures in 3.4.2 Generate an Unsigned Application Image to generate an unsigned application image for the GBL.

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.

The following sections describe how to debug an application firmware with Simplicity IDE, or IAR on a Secure Boot enabled device.

4.1 Simplicity IDE

This application note uses Simplicity Studio v5.2.3.1. The procedures and pictures may be different for the other versions of Simplicity Studio 5.

1. The Simplicity IDE creates a folder below (`<NAME>` is the Windows User Name on PC) in Windows when building the unsigned application image.

   ```
   C:\Users\<NAME>\SimplicityStudio\v5_workspace\blink_baremetal\GNU ARM v10.2.1 - Default
   ```

2. Follow the procedures in 3.4.3 Signing for ECDSA-P256-SHA256 Secure Boot or 3.4.4 Signing for Certificate-Based Secure Boot to create a batch file (Windows) to sign the unsigned application image and then flash it to the device. This application note uses ECDSA-P256-SHA256 Secure Boot (Using Simplicity Commander) as an example to create a `secure_boot_debug.bat` file below.

   ```
   commander convert blink_baremetal.s37 --secureboot --keyfile sign_key.pem --verify sign_pubkey.pem
   --outfile blink_baremetal.s37
   commander flash blink_baremetal.s37
   ```

3. Copy the batch file in step 2 and files (`sign_key.pem` and `sign_pubkey.pem` in this example) specified in `secure_boot_debug.bat` to the folder in step 1.
4. Right-click the project in the **Project Explorer** window, and then click **Properties** to open the properties dialog.

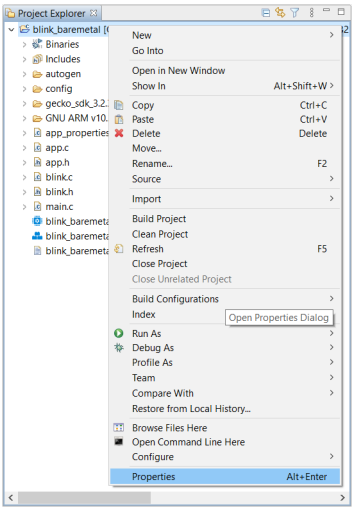


Figure 4.1. Project Explorer window

5. Select **C/C++ Build→Settings→Build Steps**. Enter the phrase below to the **Command:** box under the **Post-build steps** (enter text to **Description:** box is optional) to run the batch file as a post-build action. Click [**Apply and Close**] to exit.

```
cmd //c 'secure_boot_debug.bat'
```

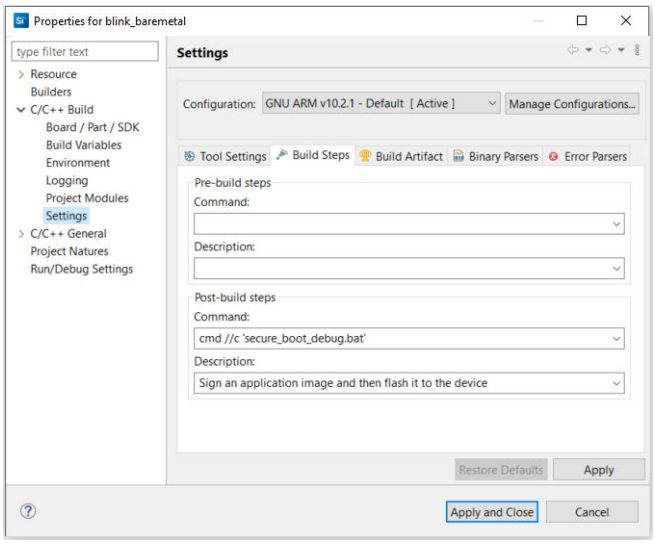


Figure 4.2. Properties dialog box

6. After building the project, the batch file in the **Post-build steps** overwrites the unsigned application image with the signed application image.

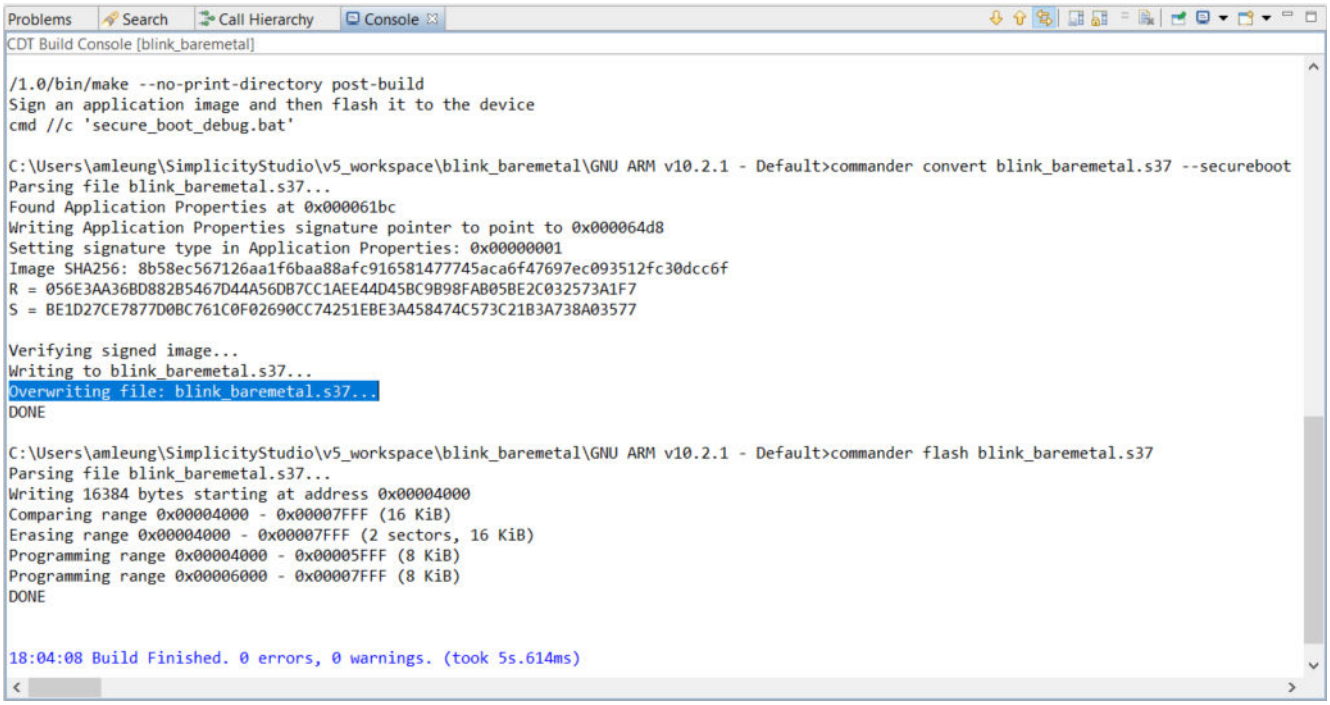


Figure 4.3. Overwriting in Post-build steps

Note: If the project is already up-to-date, it will not invoke the **Post-build steps** in step 5 to run the batch file. Use a dummy edit (add space or newline) on one of the source files in the project to trigger the build action.

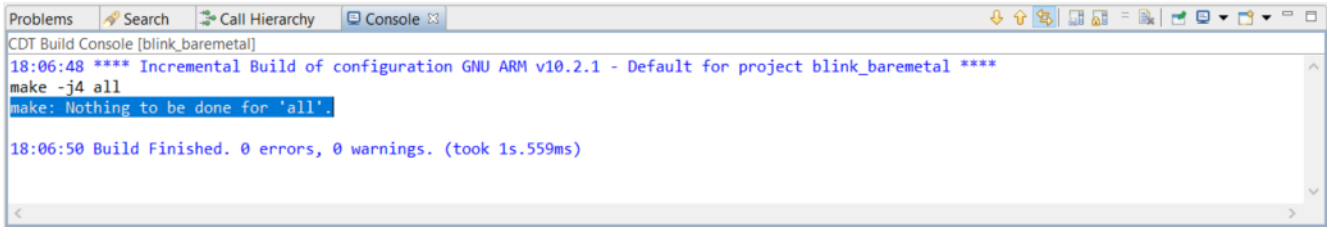


Figure 4.4. Console window

7. The application starts to run if no error in step 6.

8. Select the project in the **Project Explorer** window, click **Run→Attach to→1 Silicon Labs ARM Program** to attach to the running target for debugging on the signed application image.

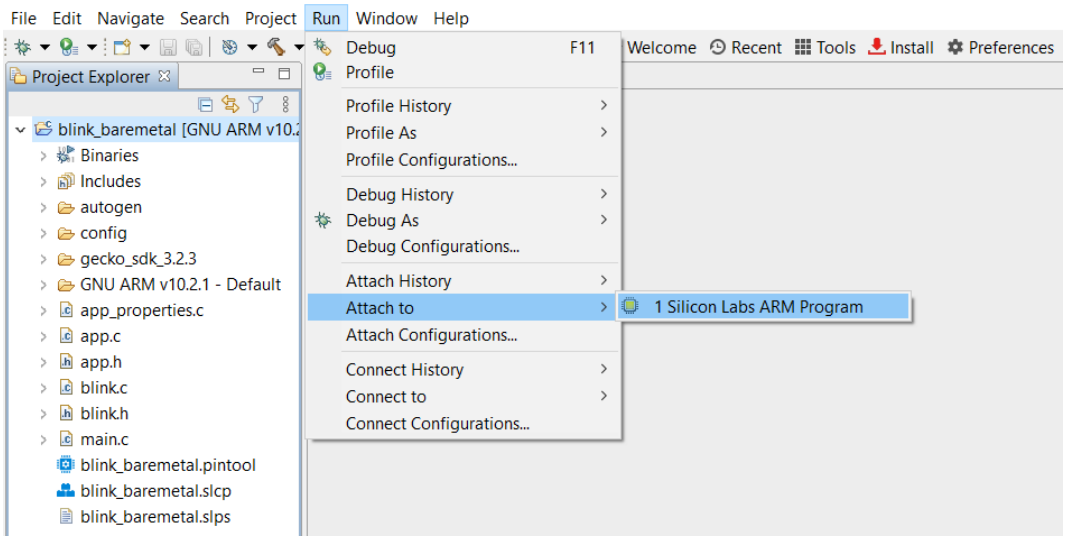


Figure 4.5. Run window

4.2 IAR

This section uses Simplicity Studio v5.4.2.0 and IAR v9.20.4. The procedures and pictures may be different for the other versions of Simplicity Studio 5 and IAR.

1. The **Overview** tab shows the **Target and Tool Settings** card on the left side. Scroll down if necessary and click [**Change Target/SDK/Generators**].

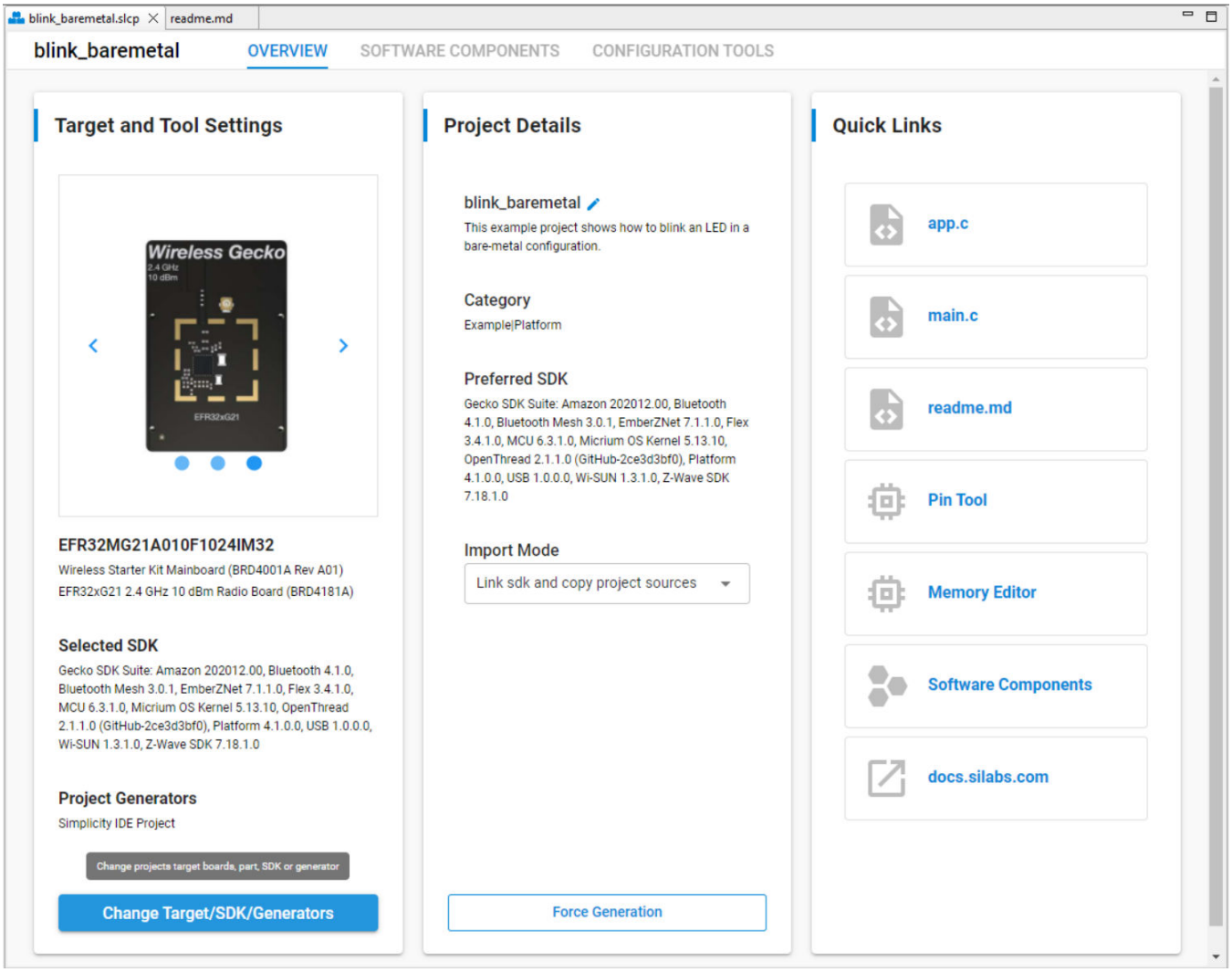


Figure 4.6. Overview tab in SCLP file

2. Drop down the **CHANGE PROJECT GENERATORS** list and select **IAR Embedded Workbench Project**. Click [**Save**] to generate an IAR project.

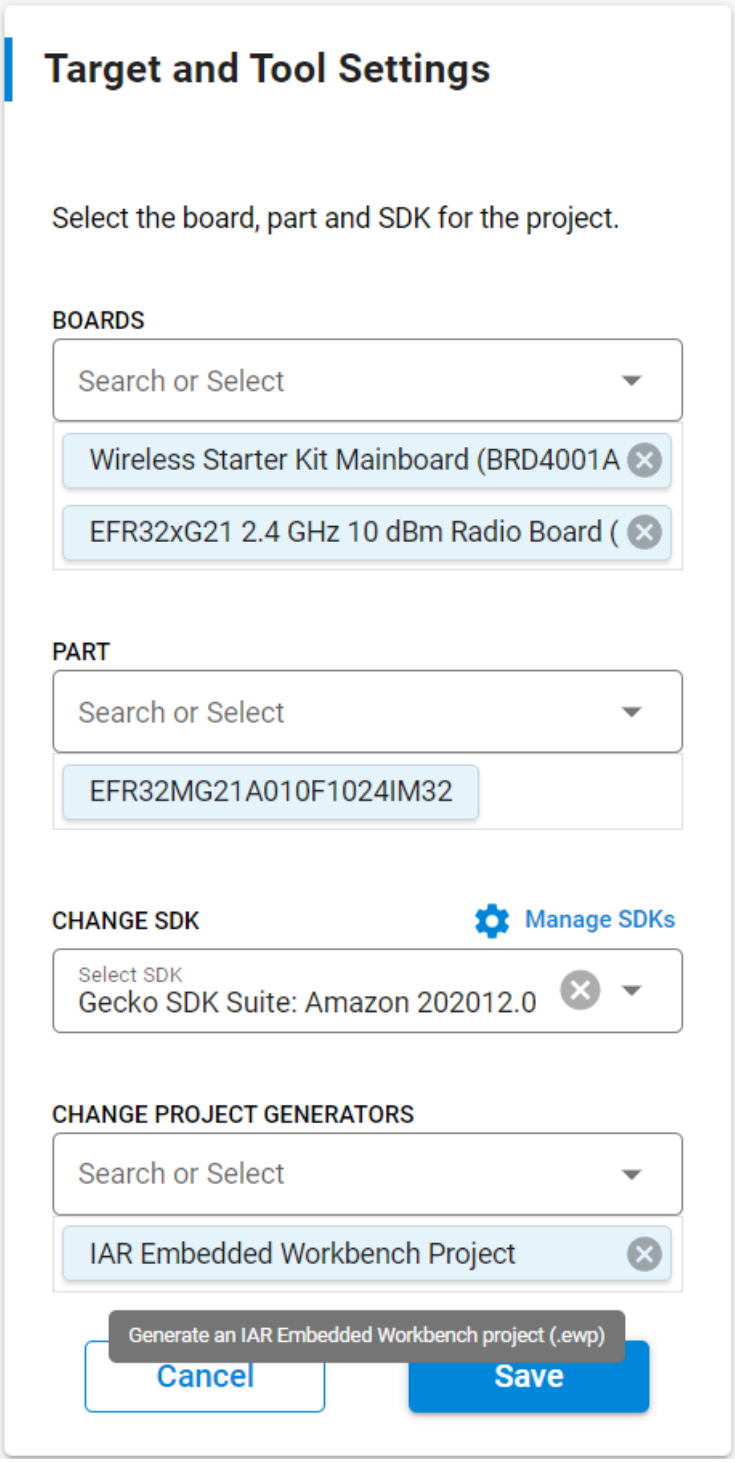


Figure 4.7. Target and tool settings dialog box

3. Double click the IAR workspace file (`blink_baremetal.eww`) in the **Project Explorer** window to open the IAR project. The IAR creates a folder below (`<NAME>` is the Windows User Name on PC) in Windows to store the compiled image.

   `C:\Users\<NAME>\SimplicityStudio\v5_workspace\blink_baremetal\ewarm-iar\exe`

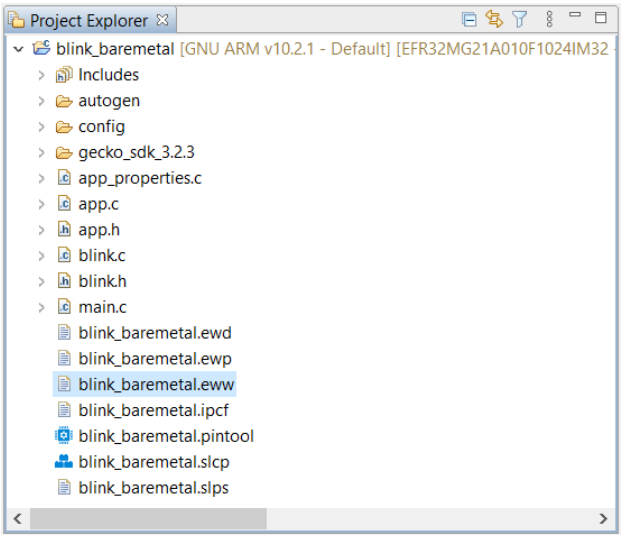


Figure 4.8. Project Explorer

4. Follow the procedures in 3.4.3 Signing for ECDSA-P256-SHA256 Secure Boot or 3.4.4 Signing for Certificate-Based Secure Boot to create a batch file (Windows) to sign the unsigned application image. This application note uses ECDSA-P256-SHA256 Secure Boot (Using Simplicity Commander) as an example to create a `secure_boot_debug.bat` file below.

```
cd C:\Users\<NAME>\SimplicityStudio\v5_workspace\blink_baremetal\ewarm-iar\exe
commander convert blink_baremetal.s37 --secureboot --keyfile sign_key.pem --verify sign_pubkey.pem
--outfile blink_baremetal.s37
```

5. Copy the batch file in step 4 and files (`sign_key.pem` and `sign_pubkey.pem` in this example) specified in `secure_boot_debug.bat` to the folder in step 3.

6. Right-click the project in the workspace, and then click **Options...**.

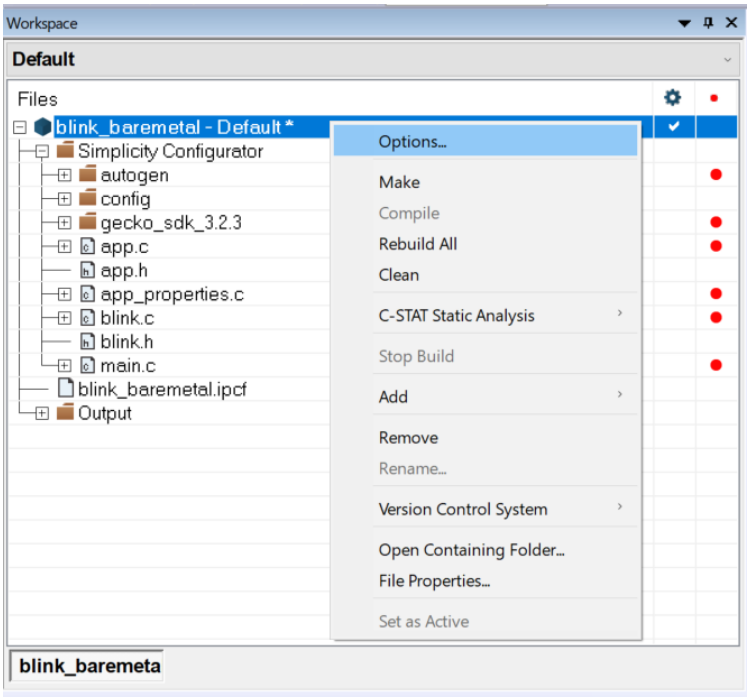


Figure 4.9. Workspace window

Note: For GSDK v3.2 and lower, the `app_properties.c` is manually added to the IAR project.

7. Click **Build Actions** to open the **Build Actions Configuration** dialog box. Enter the phrase below to the **Post-build command line:** box to run the batch file as a post-build action. Click [**OK**] to exit.

```
cmd /c "$PROJ_DIR$\ewarm-iar\exe\secure_boot_debug.bat > $PROJ_DIR$\log.txt 2>&1"
```

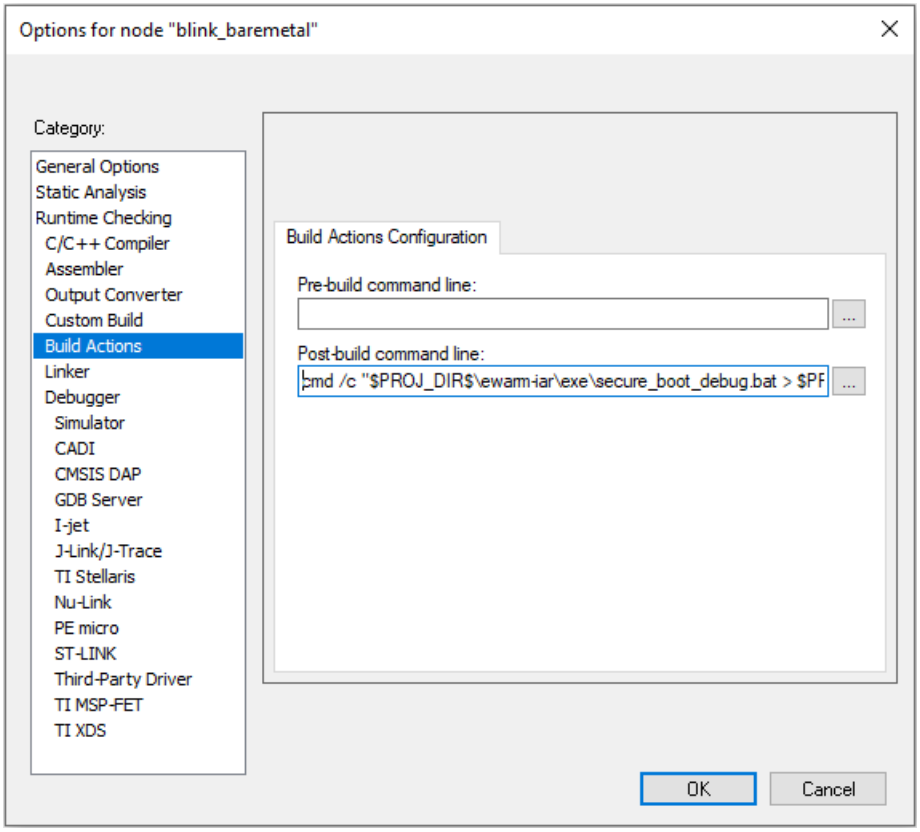


Figure 4.10. Build Actions Configuration dialog box

8. After building the project, the batch file in the **Post-build command** overwrites the unsigned application image with the signed application image.

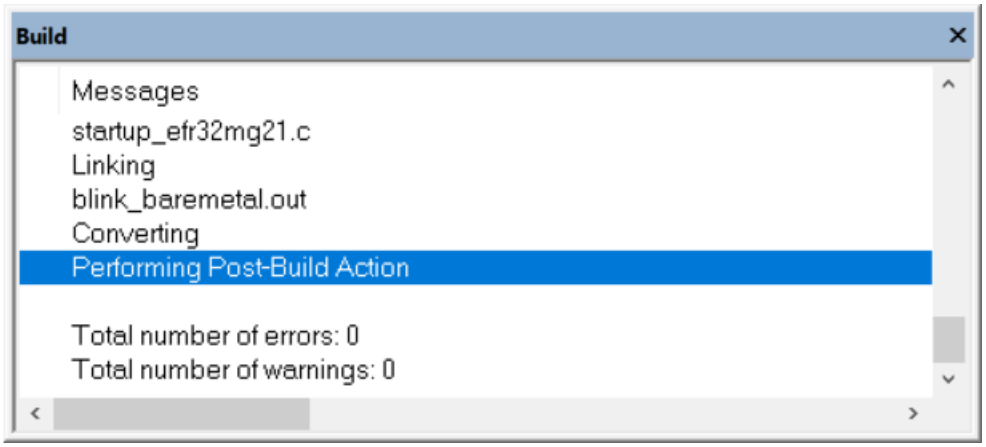


Figure 4.11. Build messages

Note: If the project is already up-to-date, it will not invoke the **Post-build command** in step 7 to run the batch file. Use a dummy edit (add space or newline) on one of the source files in the project to trigger the build action.

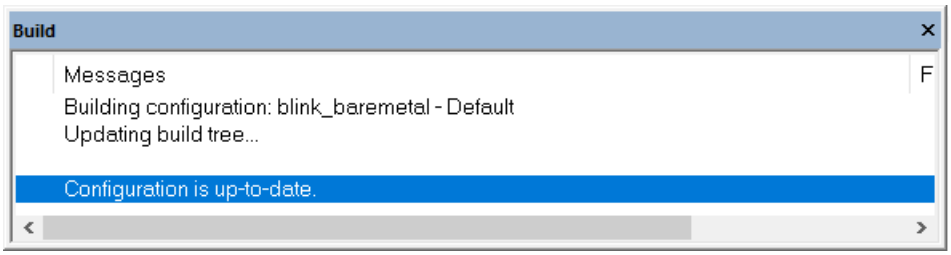


Figure 4.12. Build messages

9. The `> $PROJ_DIR$\log.txt 2>&1` redirects the batch file output to the `log.txt` file in the IAR project folder.

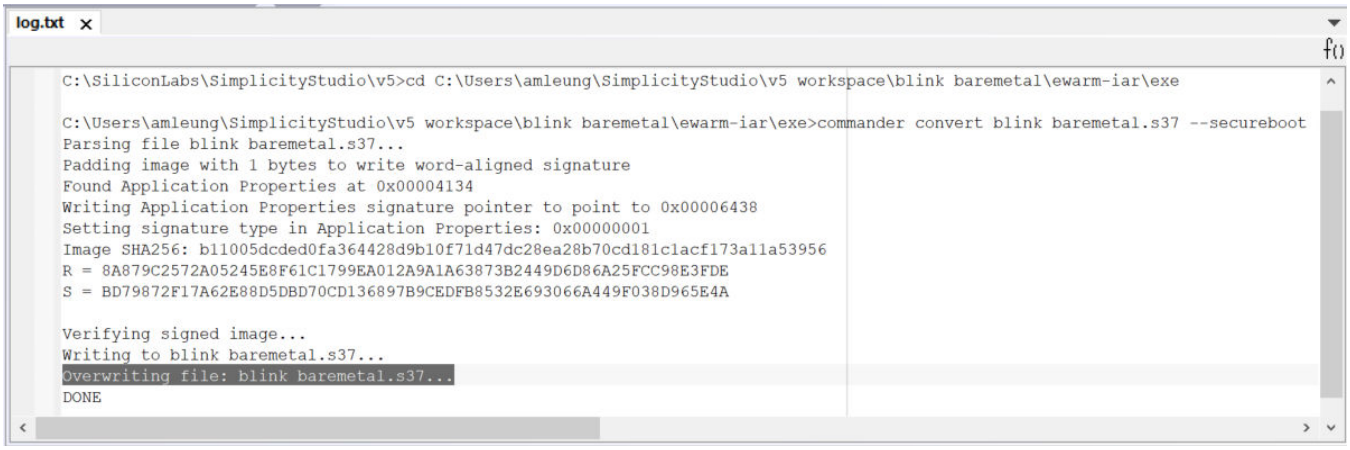


Figure 4.13. Log.txt file

10. If no error in step 8, click the  icon to start debugging on the signed application image.

5. Failure Analysis

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

Table 5.1. Scenarios for Returning Series 2 Devices 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.

Note:

1. Follow the procedures in AN1190 section "Secure Debug Unlock and Roll Challenge - 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.

6. Secure Boot Status Codes

Boot status codes can be used to know the status of the boot mechanism.

The `security status` command can be used to get the boot status.

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

SE Firmware version : 1.2.9 Serial number : 000000000000000014b457fffe045afd Debug lock : Disabled Device erase : Enabled Secure debug unlock : Disabled Tamper status : OK Secure boot : Enabled **Boot status : 0x20 - OK** DONE

The following table shows Status codes in Secure Boot mechanism and their description:

Table 6.1. Boot Status Codes

STATUS CODE	DESCRIPTION
0x00	Start PUF
0x01	Fetch OTP for bootloader
0x02	Tamper test
0x03	Self-tests
0x04	TRNG Initialization failed
0x05	NVM Initialization failed
0x0B	Jump to main loop
0x0C	Fetch OTP for host boot
0x0D	Fetch pointers host firmware
0x0E	Fetch header host firmware
0x0F	Fetch host firmware
0x10	Check version host firmware
0x11	Check signature on certificate for host firmware
0x12	Check signature host firmware
0x13	Failed to get data from internal NVM
0x14	Finding host application properties pointer
0x15	Validating host application properties structure
0x16	Validating host application signature pointer
0x17	Getting SecureBoot key
0x18	SecureBoot requires cert, but none found
0x19	Updating required certificate version failed
0x1A	Certificate is of an older version as the last cert we validated
0x1B	Certificate structure version is not supported by this firmware
0x1C	Certificate pointer is out of range
0x20	Main loop entered
0x80	PUF AC was somehow cleared
0x81	PUF failed to reconstruct after the longest delay

STATUS CODE	DESCRIPTION
0x90	ESEC aborted booting due to catching too many successive tamper resets
0xFF	Finished verifying host app

7. Revision History

Revision 1.0

April 2025

- Updated Introduction.
- Updated Security Options of Application Firmware figure in Application Firmware.
- Updated Secure Boot Examples table in Overview.
- Updated the OpenSSL version in Using an External Tool.
- Updated the recommended process for Secure Boot in Simplicity Commander.
- Updated the SE Manager Key Provisioning Example image in ERROR - unresolved reference (keys) Key Provisioning Platform Example.
- Updated options to select GBL Decryption Key for GBL upgrade image file in Provision ERROR - unresolved reference (keys).
- Updated application firmware configurations in step 4 in Bootloader-core Software Component.
- Added a note to the Secure Engine Upgrade section in Generate a GBL Upgrade Image File to clarify the usage of key pairs.
- Added BUSLOCK section.
- Added Secure Boot Status codes in Secure Boot Status Codes.

Revision 0.9

February 2023

- Removed keys/se_subsystem firmware recommendation (moved to SE Firmware) in Introduction.
- Added Provision keys/aes_key examples to Overview.
- Updated steps 5 and 8 in ERROR - unresolved reference (keys) Key Provisioning Platform Example.
- Added Provision ERROR - unresolved reference (keys).
- Updated Generate an Unsigned GBL Image.
- Added step 9 and note (after step 10) to Bootloader-core Software Component.
- Added note (after step 9) to Generate an Unsigned Application Image.
- Added a table to Signing for ECDSA-P256-SHA256 ERROR - unresolved reference (keys).
- Updated Signing for ECDSA-P256-SHA256 ERROR - unresolved reference (keys), Signing for Certificate-Based ERROR - unresolved reference (keys), and Generate a GBL Upgrade Image File to clarify two methods are used for signing.
- Updated Generate a GBL Upgrade Image File for Upgrade SE without using the staging area and Require encrypted firmware upgrade files options.
- Added a note to the Secure Engine Upgrade section in Generate a GBL Upgrade Image File.
- Updated Upgrade to ERROR - unresolved reference (keys) with RTSL for Require signed firmware upgrade files option.
- Updated Recover Devices when ERROR - unresolved reference (keys) Fails to describe the states for keys/secure_boot recovery.

Revision 0.8

August 2022

- Updated figure in Generate an Unsigned GBL Image for GSDK v4.1.
- Updated IAR file generation to match Simplicity Studio 5.4.2.0 interface.

Revision 0.7

June 2022

- Updated table and note in Series 2 Device Security Features.
- Replaced Device Compatibility with SE Firmware in Series 2 Device Security Features.
- Added Failure Analysis.

Revision 0.6

March 2022

- Added digit 4 to Note 3 in Series 2 Device Security Features.
- Updated Device Compatibility and moved it under Series 2 Device Security Features.

Revision 0.5

January 2022

- Corrected the Windows folder for GSDK v4.0 and higher in Introduction.
- Updated the web link for GSDK in Introduction.
- Added note to the table in Provision ERROR - unresolved reference (keys) and ERROR - unresolved reference (keys) Enabling.
- Added note to the table in Simplicity Commander step 5.
- Added step 9 to AppBuilder.
- Added step 9 to Bootloader-core Software Component.
- Corrected the Windows folder for GSDK v4.0 and higher in Generate an Unsigned Application Image steps 4 and 7.
- Updated note in Generate an Unsigned Application Image step 5.
- Inserted steps 5 and 6 to Bootloader Image File section in Signing for ECDSA-P256-SHA256 ERROR - unresolved reference (keys).
- Added step 5 to Application Image File section in Signing for ECDSA-P256-SHA256 ERROR - unresolved reference (keys).
- Corrected the Windows folder for GSDK v4.0 and higher in Certificate Structure.
- Inserted steps 6 and 7 to Bootloader Image File section in Signing for Certificate-Based ERROR - unresolved reference (keys).
- Added step 6 to Application Image File (Standard Certificate-Based) section in Signing for Certificate-Based ERROR - unresolved reference (keys).
- Added step 6 to Application Image File (Advanced Certificate-Based) section in Signing for Certificate-Based ERROR - unresolved reference (keys).

Revision 0.4

December 2021

- Formatting updates for source compatibility.
- Added Series 2 Device Security Features and use the terminology defined in this section throughout the document.
- Added Device Compatibility section.
- Removed terminology and Table 2.1 in Introduction.
- Added ERROR - unresolved reference (keys) Secure Loader Example to Secure Loader.
- Added ERROR - unresolved reference (keys) Time.
- Added ERROR - unresolved reference (keys) Configuration. Moved Sign Key and Secure Boot Enable Flag to this section.
- Added Using an External Tool, Using a Platform Example, and Generate Key and Signing to Overview.
- Added ERROR - unresolved reference (keys) Key Provisioning Platform Example to Provision ERROR - unresolved reference (keys) and ERROR - unresolved reference (keys) Enabling.
- Added Generate an Unsigned GBL Image to replace Overview section in keys/secure_boot.
- Added Generate an Unsigned Application Image to keys/secure_boot.
- Updated Signing for ECDSA-P256-SHA256 ERROR - unresolved reference (keys) and Signing for Certificate-Based ERROR - unresolved reference (keys) in keys/secure_boot.
- Added Generate a GBL Upgrade Image File to keys/secure_boot.
- Added Upload a GBL Upgrade Image File to keys/secure_boot.
- Added Upgrade to Certificate-Based ERROR - unresolved reference (keys) and Certificate Revocation to Examples.
- Updated Upgrade to ERROR - unresolved reference (keys) with RTSL and Recover Devices when ERROR - unresolved reference (keys) Fails in Examples.
- Added Debugging on ERROR - unresolved reference (keys) Enabled Device

Revision 0.3

July 2020

- Added keys/se_subsystem conventions to Introduction.
- Updated Figure 2.1 and Figure 2.2 to Simplicity Studio 5.
- Updated Simplicity Commander version to 1.9.2 in Using Simplicity Commander.
- Renamed Provision keys/sign_pub to Provision keys/sign_pub and keys/secure_boot Enabling, added note for keys/se_vault devices.
- Added ECDSA-P256-SHA256 keys/secure_boot and Certificate-Based keys/secure_boot to Examples.
- Added keys/se_subsystem Manager examples to Upgrade to Secure Boot with RTSL.
- Removed the Related Documents section in favor of web links in the text.

Revision 0.2

March 2020

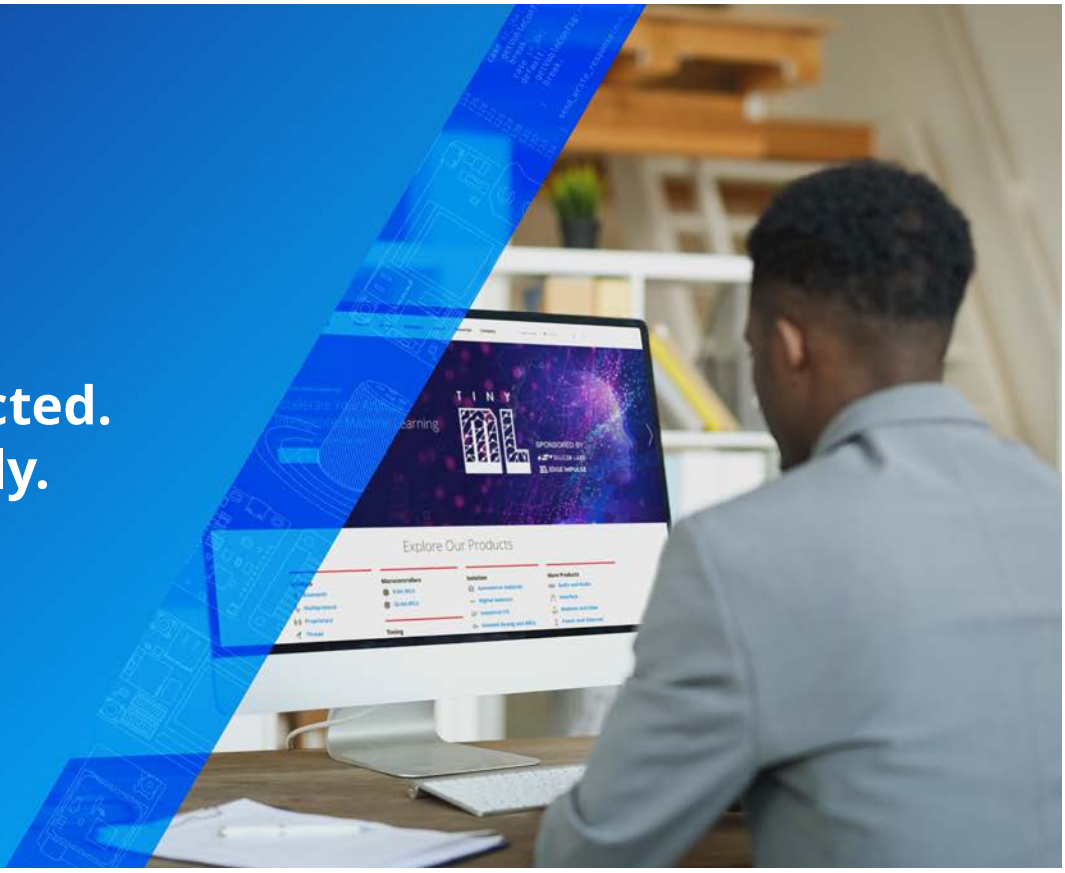
- Added figure to Secure Boot (ECDSA) in Series 1 Devices section.
- Added SE and VSE to Secure Boot (ECDSA) in Series 2 Devices section.
- Added figures to keys/secure_boot (ECDSA) in Series 2 Devices section.
- Added Secure Boot (Certificate) in Series 2 Devices section.
- Added Upgrade to keys/secure_boot with RTSL example.
- Combined all examples into one section and updated the content.
- Added Related Documents section.

Revision 0.1

August 2019

- Initial Revision.

Smart. Connected. Energy-Friendly.



IoT Portfolio
www.silabs.com/products

Quality
www.silabs.com/quality

Support & Community
www.silabs.com/community

SILICON LABS

Silicon Laboratories Inc.
400 West Cesar Chavez
Austin, TX 78701
USA

www.silabs.com