AN1218: Series 2 Secure Boot with RTSL

This application note describes the design of Secure Boot with RTSL (Root of Trust and Secure Loader), which was introduced with Wireless SoC Series 2. It also provides examples of how to implement the Secure Boot process.

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 a Series 2 device for the Secure Boot process
- Describes two methods to recover devices

1. Secure Boot Process

1.1 Introduction

The purpose of Secure Boot is to protect the integrity of the behavior of the system. Because the behavior of the system is defined by the firmware running on it, Secure Boot acts to ensure the authenticity and integrity of the firmware. 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 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 sign key. In Series 2 devices, the root of trust is in ROM, and the upgradable portion is checked both for integrity and authenticity.

On Series 2 devices, the Secure Boot is implemented by the Secure Element. The Secure Element may be hardware-based, or virtual (software). If hardware-based, the implementation may be either with or without Secure Vault. Throughout this document, the following conventions will be used.

- SE - Hardware Secure Element, either with or without Secure Vault if not specified
- VSE - Virtual Secure Element
- Secure Element - Either SE or VSE

The Secure Boot with RTSL is implemented by Root code executed by the SE Core or by the Cortex-M33 operating in VSE (Root mode). Table 1.1 Minimum Secure Element Firmware Version for Secure Boot with RTSL on page 2 indicates the minimum required Secure Element Root code versions that support Secure Boot with RTSL.

For more information about Secure Element, see section "*Secure Element Subsystem*" in AN1190: Series 2 Secure Debug.

Table 1.1. Minimum Secure Element Firmware Version for Secure Boot with RTSL

Device	Secure Element	Minimum Firmware Version for Secure Boot with RTSL
EFR32xG21A	SE without Secure Vault	Version 1.2.1
EFR32xG22	VSE	Version 1.2.1
Note: Silicon Labs strongly recommends installing the latest Secure Element firmware on Series 2 devices.		

1.2 Secure Boot (ECDSA-P256-SHA256) in Series 1 Devices

The Secure Boot process for Series 1 devices originates in flash, typically with the execution of the first stage of Gecko Bootloader. The first stage of Gecko Bootloader checks to see if an upgrade is pending for the second stage of Gecko Bootloader. 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 Gecko Bootloader checks the integrity and authenticity of the application image before executing it. If the integrity check fails, program control remains in the second stage bootloader. Figure 1.1 Series 1 Secure Boot (ECDSA-P256-SHA256) Process on page 3 illustrates the Secure Boot process on Series 1 devices.

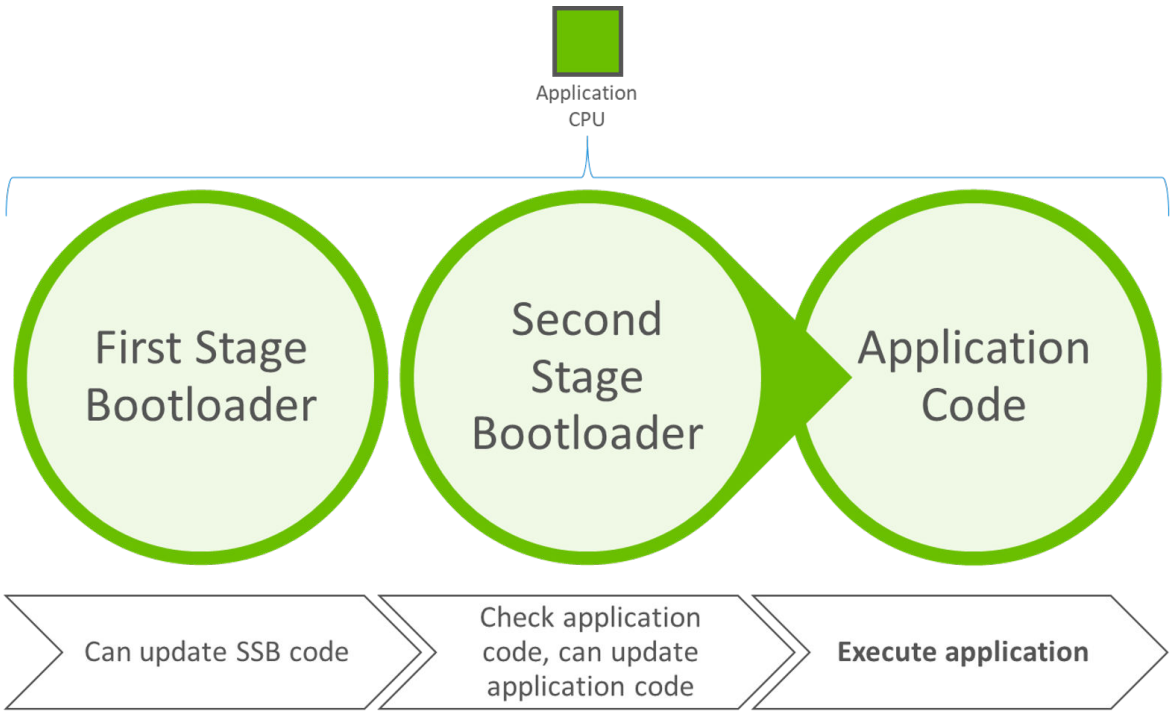


Figure 1.1. Series 1 Secure Boot (ECDSA-P256-SHA256) Process

UG266: Silicon Labs Gecko Bootloader User's Guide details the procedure for generating and downloading signed firmware images using Simplicity Commander.

1.3 Secure Boot (ECDSA-P256-SHA256) in Series 2 Devices

Details can be found in section "*Gecko Bootloader Security Features*" in UG266: Silicon Labs Gecko Bootloader User's Guide and the section 2.3.2 ECDSA-P256-SHA256 Secure Boot example.

1.3.1 SE

In Series 2 devices with SE Core, the Secure Boot process originates in ROM contained in the SE security co-processor. Figure 1.2 Series 2 SE Secure Boot (ECDSA-P256-SHA256) Process on page 4 and Figure 1.3 Series 2 SE Secure Boot (ECDSA-P256-SHA256) Flow on page 4 illustrate the Secure Boot process and flow on Series 2 SE devices.

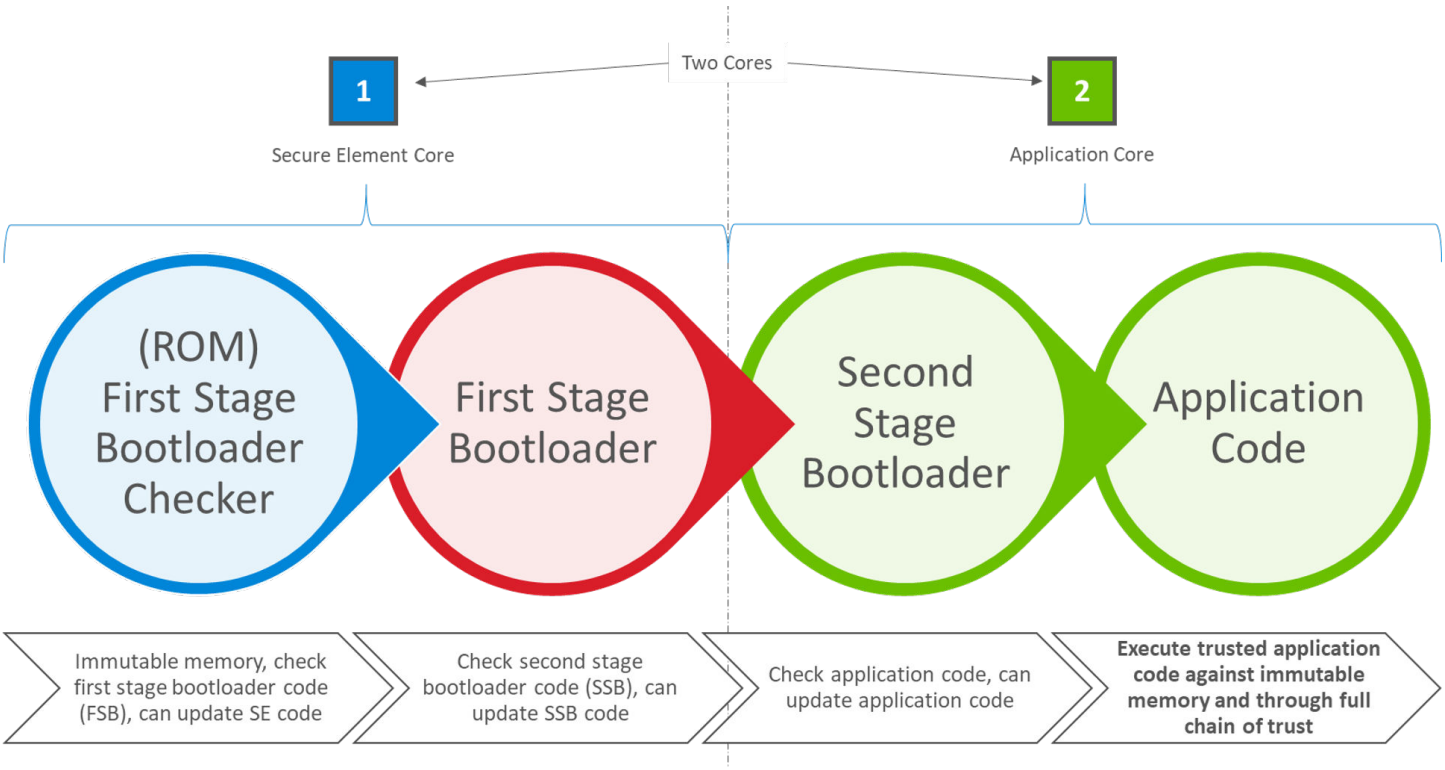


Figure 1.2. Series 2 SE Secure Boot (ECDSA-P256-SHA256) Process

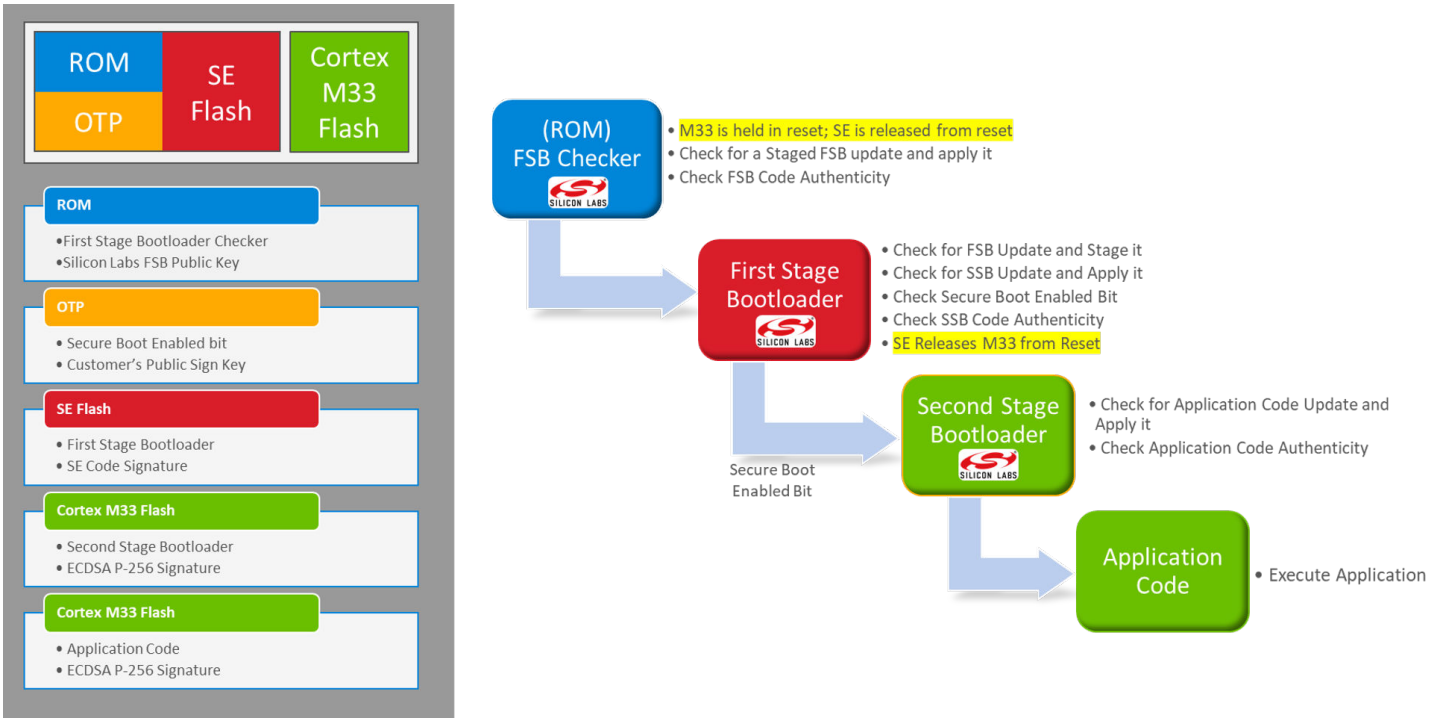


Figure 1.3. Series 2 SE Secure Boot (ECDSA-P256-SHA256) Flow

1.3.2 VSE

In Series 2 devices with VSE, the host MCU assumes an elevated security state out of reset and securely boots itself from code that originates in ROM. Figure 1.4 Series 2 VSE Secure Boot (ECDSA-P256-SHA256) Process on page 5 and Figure 1.5 Series 2 VSE Secure Boot (ECDSA-P256-SHA256) Flow on page 5 illustrate the Secure Boot process and flow on Series 2 VSE devices.

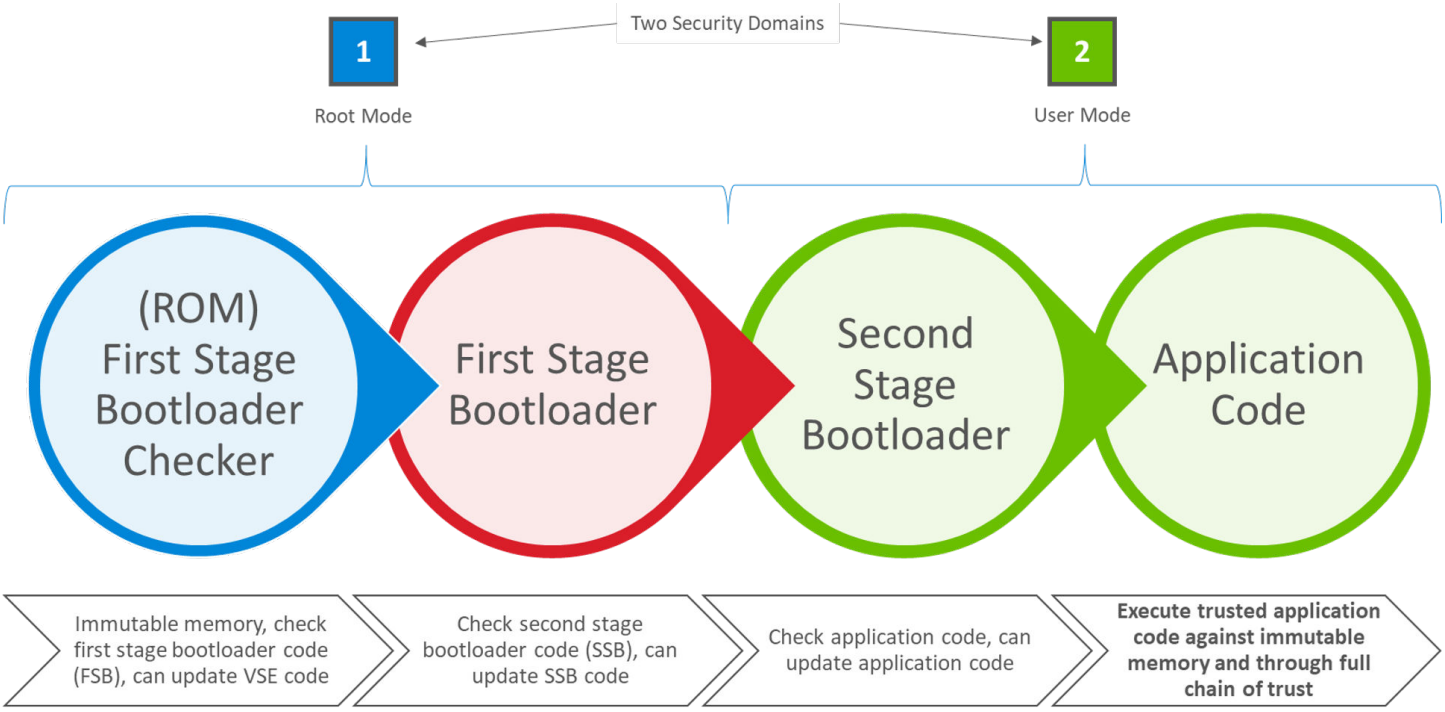


Figure 1.4. Series 2 VSE Secure Boot (ECDSA-P256-SHA256) Process

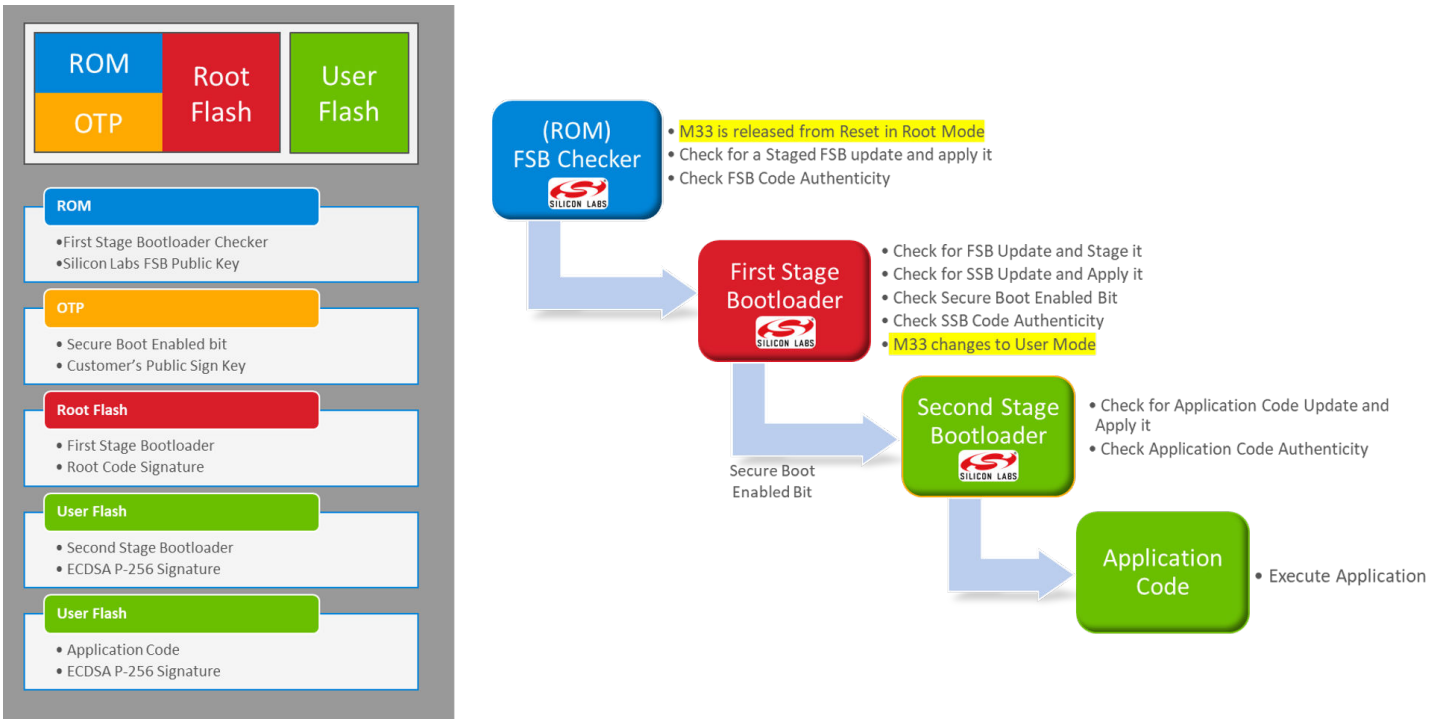


Figure 1.5. Series 2 VSE Secure Boot (ECDSA-P256-SHA256) Flow

1.4 Secure Boot (Certificate) in Series 2 Devices

On Series 2 devices, a certificate-based Secure Boot operation is supported. Details can be found in section "*Gecko Bootloader Security Features*" in UG266: Silicon Labs Gecko Bootloader User's Guide and the section 2.3.3 Certificate-Based Secure Boot example.

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

If the certificate's private key is leaked, all devices that have been programmed with that certificate can be updated with an image containing a certificate with a higher version (key revocation).

1.5 Sign Key and Secure Boot Enable Flag

In Series 2 devices, the Sign Key and the Secure Boot Enable flag are both located in immutable one-time programmable memory (OTP). This means that once either is programmed, its respective value cannot be changed. Once the 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 Sign Key used for Series 2 devices is the public portion of an ECDSA key pair over the NIST prime curve P-256. The 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 a common 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 certificate, should be tightly held, ideally secured in a hardware security module (HSM).

AN1222: Production Programming of Series 2 Devices details the procedure for Sign Key provisioning during production.

1.6 Secure Loader

In Series 2 devices, the Secure Loader is firmware pre-loaded into the devices. It is maintained by Silicon Labs, and is deployed through secure upgrade packages. It is the functional equivalent of the first-stage Gecko Bootloader on Series 1 devices (see UG266: Silicon Labs Gecko Bootloader User's Guide 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 Gecko Bootloader before calling the Secure Loader's `Upgrade` command.

2. Examples

2.1 Overview

The examples for Series 2 Secure Boot are described in Table 2.1 Secure Boot Examples on page 7.

Table 2.1. Secure Boot Examples

Example	Device	Radio Board	SE or VSE Firmware	Tool
Provision Public Sign Key and Secure Boot Enabling	EFR32MG21A010F1024IM32	BRD4181A	Version 1.2.1	Simplicity Studio
	EFR32MG22C224F512IM40	BRD4182A	Version 1.2.2	Simplicity Commander
ECDSA-P256-SHA256 Secure Boot	EFR32MG21A010F1024IM32	BRD4181A	Version 1.2.1	Simplicity Commander
Certificate-Based Secure Boot	EFR32MG21A010F1024IM32	BRD4181A	Version 1.2.1	Simplicity Commander
Recover devices when Secure Boot fails	EFR32MG21A010F1024IM32	BRD4181A	Version 1.2.1	Simplicity Commander (GUI)
	EFR32MG22C224F512IM40	BRD4182A	Version 1.2.2	Simplicity Commander (CLI)
Upgrade to Secure Boot with RTSL	EFR32MG21A010F1024IM32	BRD4181A	Version 1.2.1	Secure Element Manager

2.1.1 Using Simplicity Commander

1. The Command Line Interface (CLI) of Simplicity Commander is invoked by `commander.exe` in the Simplicity Commander folder. The location in Windows is `C:\SiliconLabs\SimplicityStudio\<version>\developer\adapter_packs\commander`.
2. Simplicity Commander Version 1.9.2 is used in this application note.

```
commander --version
```

```
Simplicity Commander 1v9p2b791

JLink DLL version: 6.70a
Qt 5.5.1 Copyright (C) 2017 The Qt Company Ltd.
EMDLL Version: 0v17p12b535
mbed TLS version: 2.6.1

Emulator found with SN=440068705 USBAddr=0

DONE
```

3. The target Wireless Starter Kit (WSTK) must be specified using the `--serialno <J-Link serial number>` option if more than one WSTK is connected via USB.
4. The target device must be specified using the `--device <device name>` option if the WSTK is in debug mode OUT.
5. Run the `security genkey` command to generate the Sign Key pair (`sign_key.pem` and `sign_pubkey.pem`) for Secure Boot examples.

```
commander security genkey --type ecc-p256 --privkey sign_key.pem --pubkey sign_pubkey.pem
```

```
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
DONE
```

6. Run the `gbl keyconvert` command to generate the Public Sign Key text file (`sign_pubkey.txt`) for Public Sign Key provisioning example.

```
commander gbl keyconvert sign_pubkey.pem -o sign_pubkey.txt
```

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

7. For more information about Simplicity Commander, see UG162: Simplicity Commander Reference Guide.

2.2 Provision Public Sign Key and Secure Boot Enabling

In order to enable Secure Boot, a Sign Key pair must be generated. The public portion of the Sign Key pair is used to verify the image or certificate during Secure Boot and must then be written to the Secure Element OTP. The private portion of the Sign Key pair is used to sign the application image or certificate for Secure Boot, and this private key must be protected, ideally stored in a Hardware Security Module (HSM) or equivalent key storage instrument.

For SE with Secure Vault devices, the tamper settings must be written together with secure boot settings, and are immutable after they are written.

Note: Simplicity Studio does currently not support tamper provisioning on SE with Secure Vault devices. The procedures in 2.2.1 Simplicity Studio are only for SE without Secure Vault and VSE devices.

2.2.1 Simplicity Studio

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

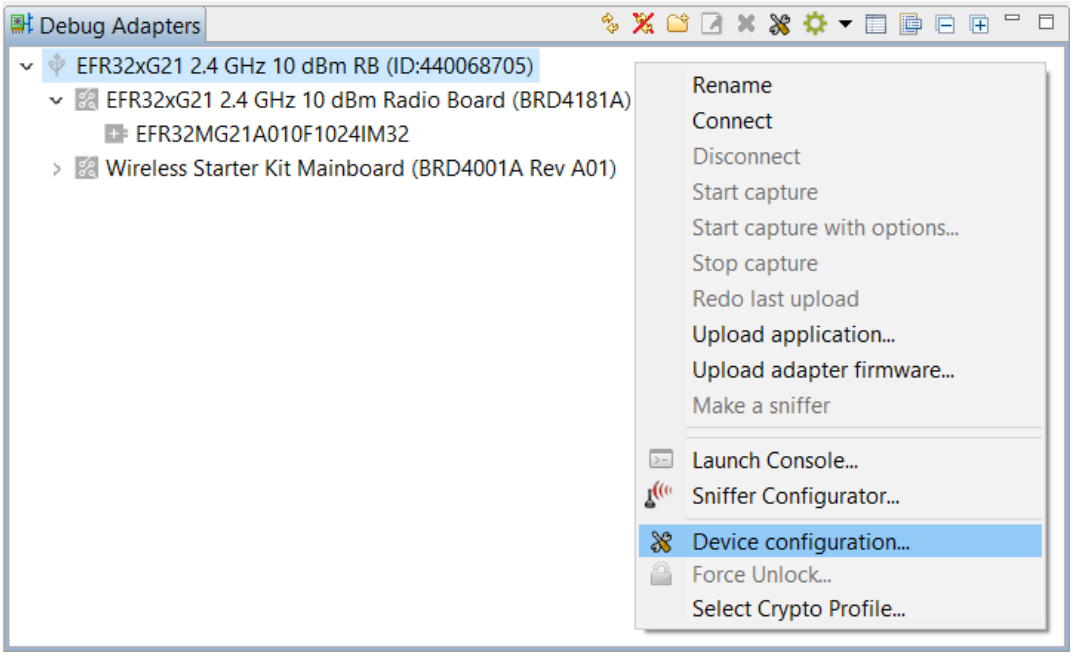


Figure 2.1. Debug Adapter Context Menu

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.

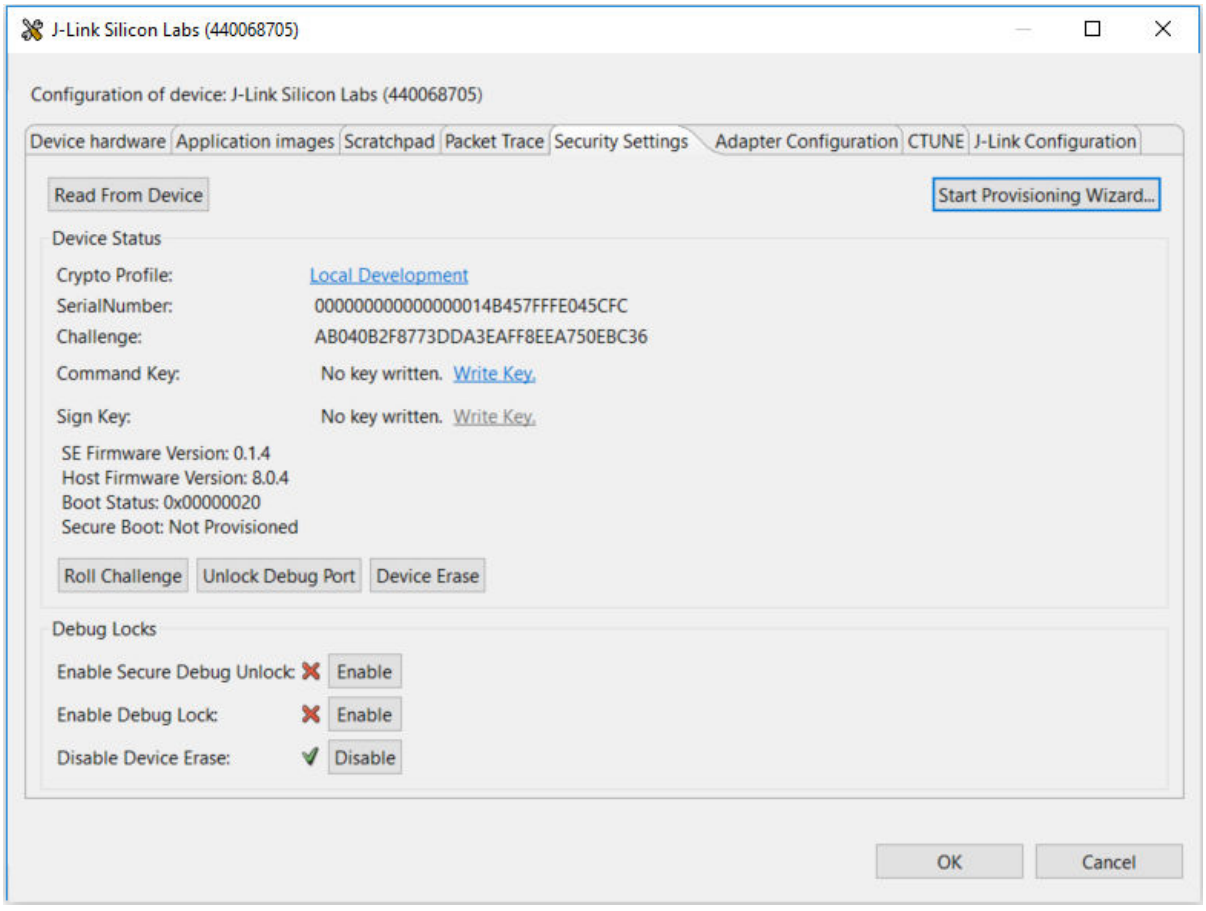


Figure 2.2. Configuration on Selected Device

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.

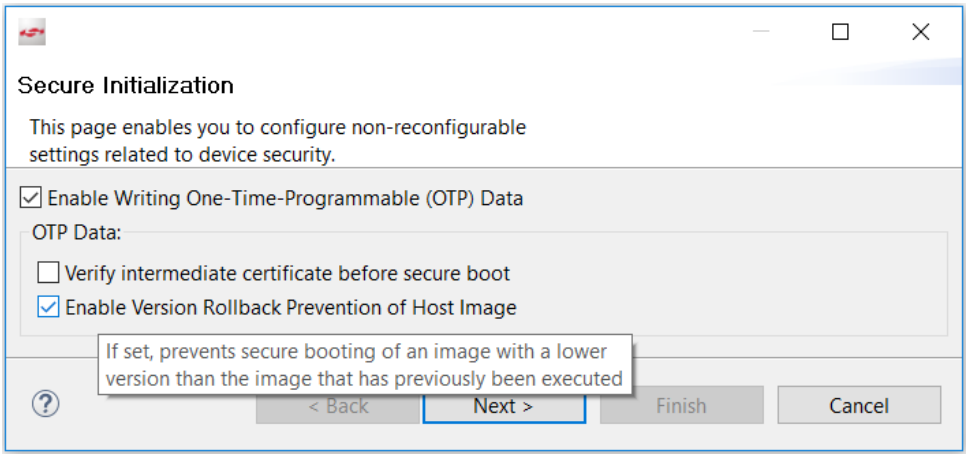


Figure 2.3. Secure Initialization Dialog Box

Note: The **Verify intermediate certificate before secure boot** option is for certificate-based Secure Boot as described in 1.4 Secure Boot (Certificate) in Series 2 Devices.

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

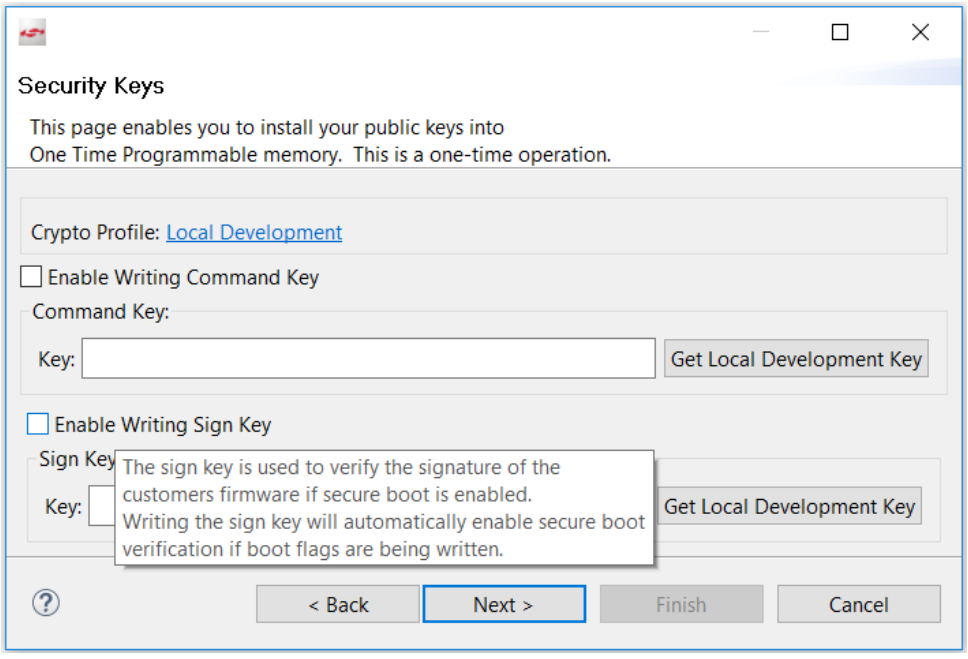


Figure 2.4. Security Keys Dialog Box

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

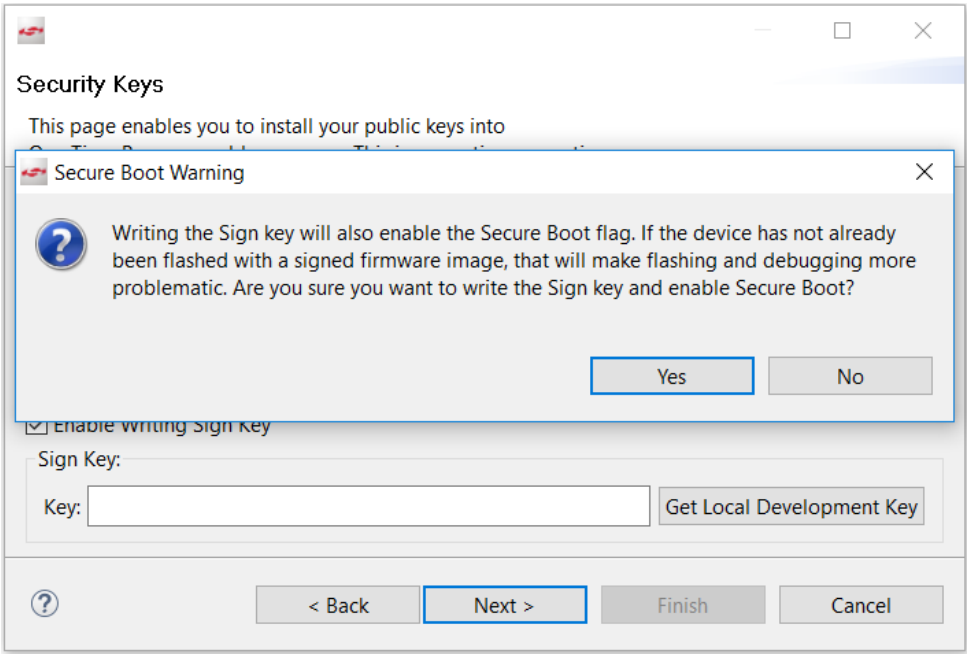


Figure 2.5. Secure Boot Warning

6. Open `sign_pubkey.txt` file generated in 2.1.1 Using Simplicity Commander step 6.

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

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

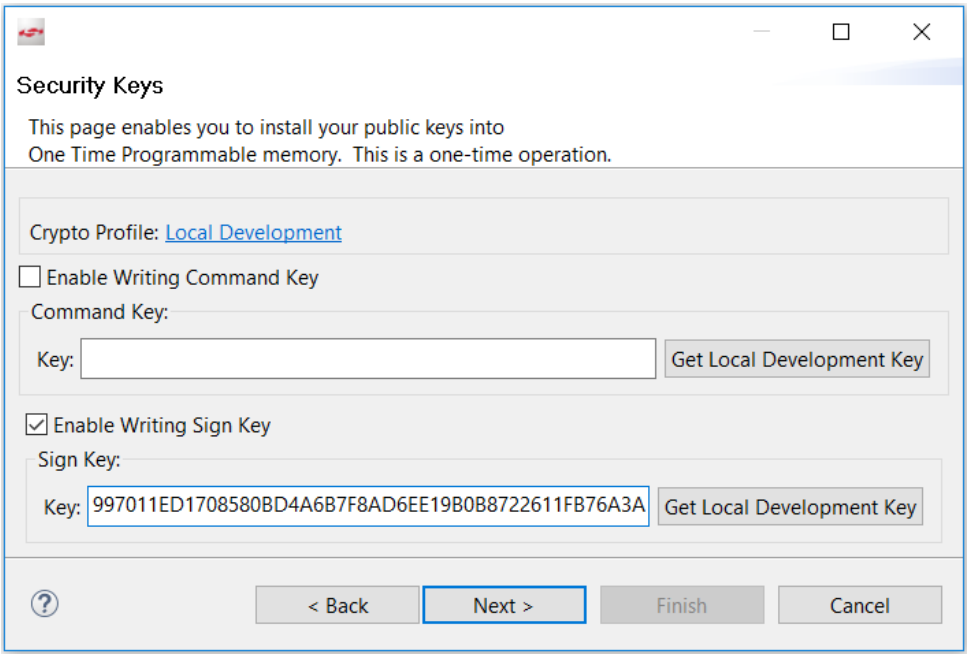


Figure 2.6. Enter Public Sign Key

8. Click [**Next >**]. The **Secure Locks** dialog box is displayed. When Secure Boot is enabled, the **Debug locks** are not set by default.

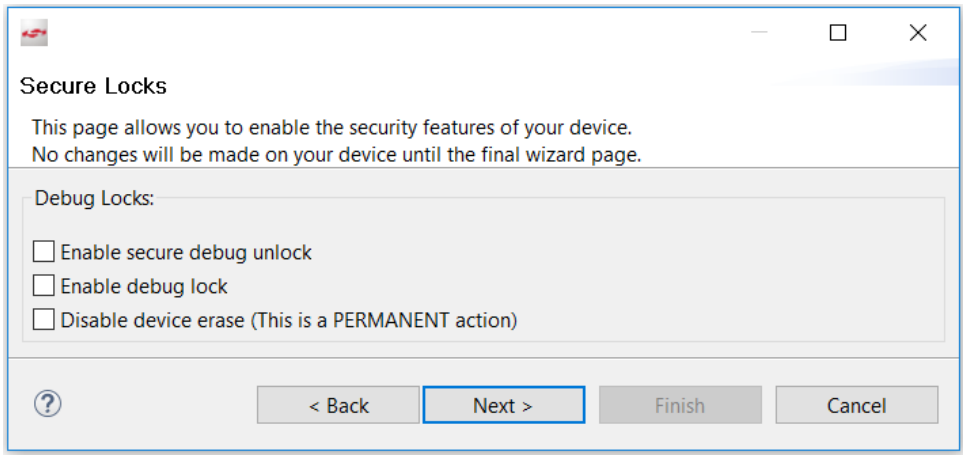


Figure 2.7. Security Locks Dialog Box

Note: See AN1190: Series 2 Secure Debug for more information about these locks

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

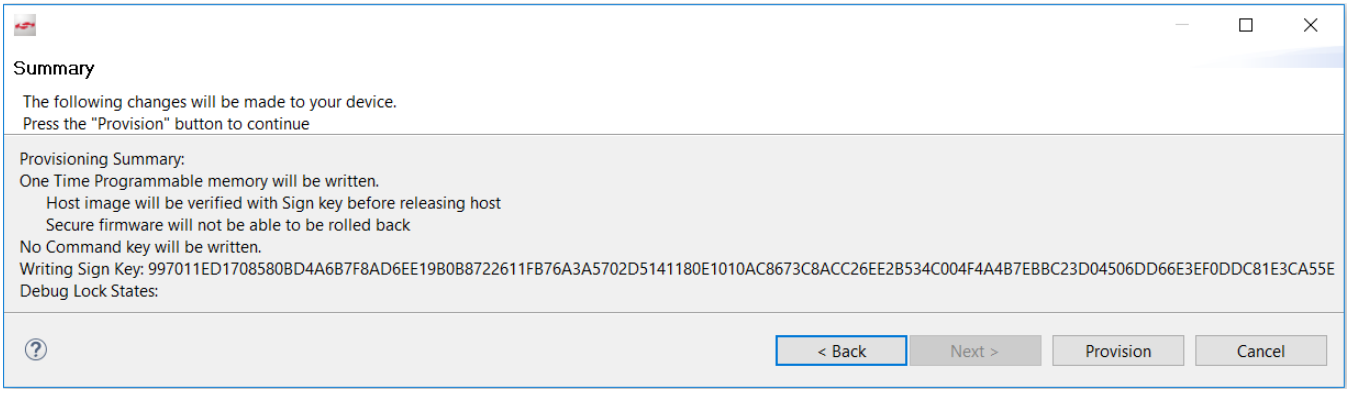


Figure 2.8. Summary Dialog Box

10. If the information displayed is correct, click [**Provision**]. Click [**Yes**] to confirm.

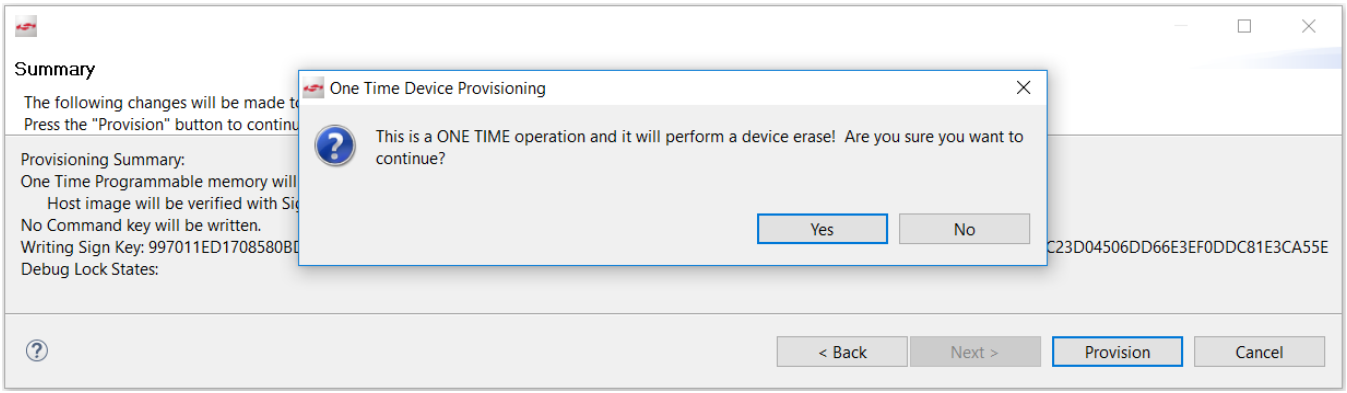


Figure 2.9. One Time Device Provisioning Window

Note: The Public Sign Key and Secure Boot enable cannot be changed once written.

11. The **Provisioning Status** is displayed in the **Summary** dialog box.

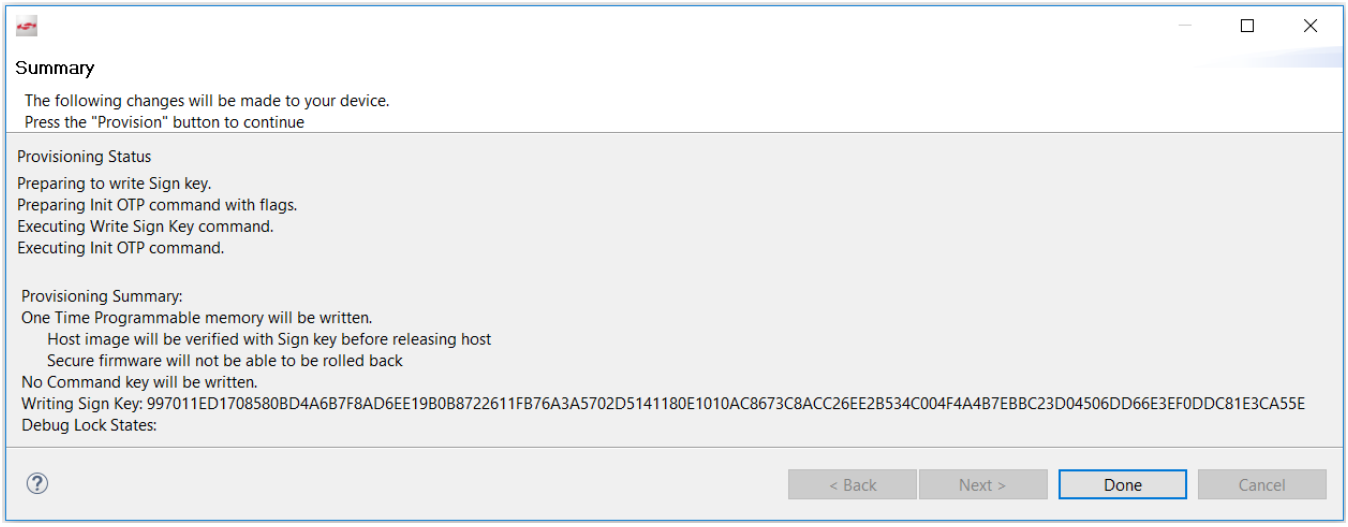


Figure 2.10. Provisioning Status

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

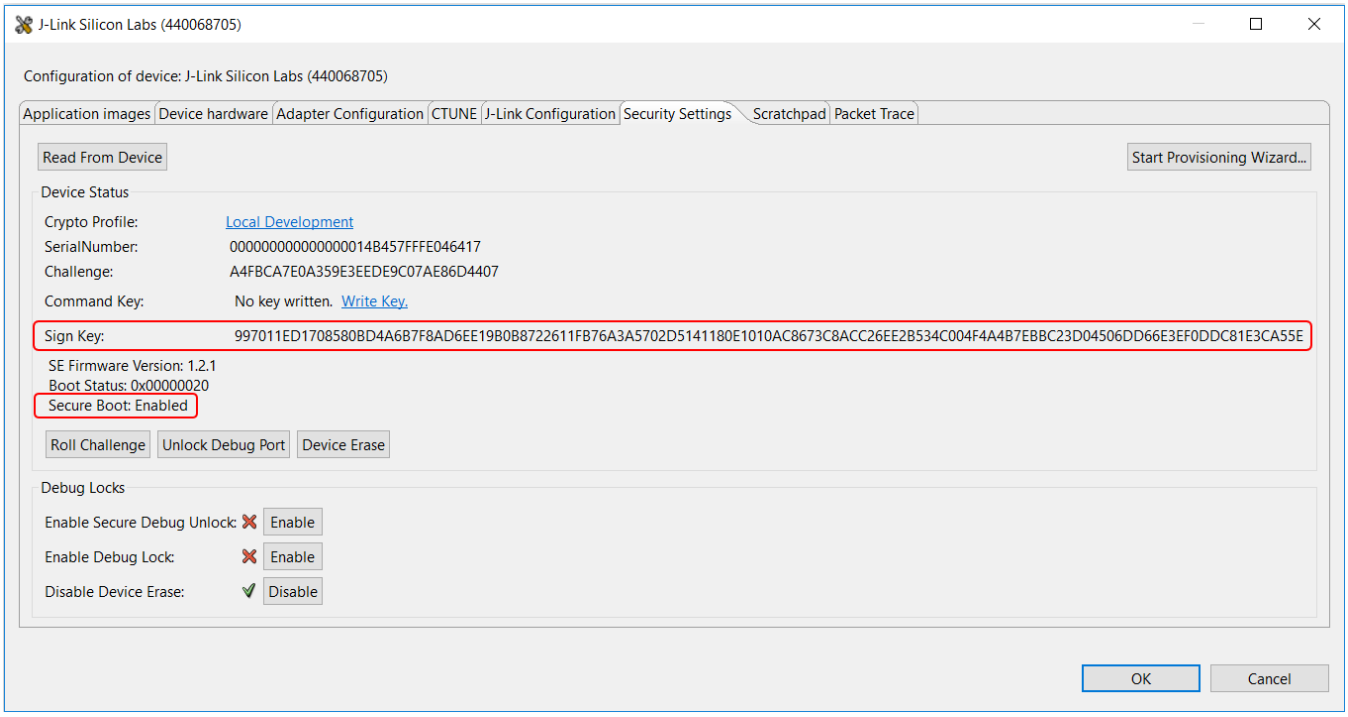


Figure 2.11. Device Configuration after Provisioning

2.2.2 Simplicity Commander

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

```
commander security status --device EFR32MG22C224F512 --serialno 440068705
```

```
SE Firmware version : 1.2.1
Serial number       : 000000000000000014b457fffed50d1e
Debug lock          : Disabled
Device erase        : Enabled
Secure debug unlock : Disabled
Secure boot         : Disabled
Boot status         : 0x20 - OK
DONE
```

2. Run the `security writekey` command to provision the Public Sign Key with `sign_pubkey.pem` file generated in 2.1.1 Using Simplicity Commander step 5.

```
commander security writekey --sign sign_pubkey.pem --device EFR32MG22C224F512 --serialno 440068705
```

```
Device has serial number 000000000000000014b457fffed50d1e

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

Note: The Public Sign Key cannot be changed once written.

3. Run the `security readkey` command to verify the Public Sign Key with `sign_pubkey.txt` generated in 2.1.1 Using Simplicity Commander step 6.

```
commander security readkey --sign --device EFR32MG22C224F512 --serialno 440068705
```

```
997011ED1708580BD4A6B7F8AD6EE19B0B8722611FB76A3A5702D5141180E101
0AC8673C8ACC26EE2B534C004F4A4B7EBBC23D04506DD66E3EF0DDC81E3CA55E
DONE
```

4. Instructions on how to enable the Secure Boot and tamper provisioning (SE with Secure Vault devices only) can be found in section "*Secure Boot Enabling*" in AN1222: Production Programming of Series 2 Devices.

2.3 Secure Boot

2.3.1 Overview

1. The Sign Key pair (`sign_key.pem` and `sign_pubkey.pem`) in 2.1.1 Using Simplicity Commander step 5 is used to sign and verify the image file or certificate in the following examples.
2. A Gecko Bootloader called `bootloader-uart-xmodem` (v1.10.3) is used in the following examples. It can be built using the Application Builder in Simplicity Studio. For Series 2 devices, only the main bootloader hex file `bootloader-uart-xmodem.s37` is generated. This file should be copied to the Simplicity Commander folder.

   Note: For more information on how to build a Gecko Bootloader example, see section "*Getting Started with the Gecko Bootloader*" in UG266: Silicon Labs Gecko Bootloader User's Guide.

3. For Series 2 devices, the bootloader is placed at address `0x00000000`. Therefore the start address of the application image should be set to the next main flash page after the bootloader.

   Note: For more information on how to set up the start address of the application image, see section "*Creating Applications for Use with the Bootloader*" in AN0042: USB/UART Bootloader.

4. Assume an unsigned application image file `blink.s37` at `0x00004000` (SE devices) or `0x00006000` (VSE devices) was generated and copied to the Simplicity Commander folder.

   Note: For more information about application start address, see section "*Memory Space For Bootloading*" in UG103.6: Bootloader Fundamentals.

5. The application image should contain an `ApplicationProperties_t` structure declaring the application version, capabilities, and other metadata. The definition of `ApplicationProperties_t` can be found in `application_properties.h` (the location in Windows is `C:\SiliconLabs\SimplicityStudio\<version>\developer\sdks\gecko_sdk_suite\<version>\platform\bootloader\api`).

   Note: In protocol stacks from Silicon Labs, the `ApplicationProperties_t` structure is already present in an `application_properties.c` file.

6. Run the `util appinfo` command to get all available information about `ApplicationProperties_t` structure in an application image. If **Enable Version Rollback Prevention of Host Image** in Figure 2.3 Secure Initialization Dialog Box on page 10 is checked, application images with a lower `App version` than the image currently stored in flash will not run on the device.

```
commander util appinfo blink.s37
```

```
Parsing file blink.s37...
Found application properties in image.
Application properties info:
Application properties location : 0x0000109c
Signature location              : Not set (0x00000000)
Signature type                  : No signature
Long token section address      : Not set (0x00000000)

Application data info:
If rollback prevention is enabled, 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                     : 0x00000000
Product ID                      : Not set (0x00000000000000000000000000000000)

No certificate found in image.
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 more information about the `ApplicationProperties_t` structure, see section "*Application Properties*" in UG266: Silicon Labs Gecko Bootloader User's Guide.

2.3.2 ECDSA-P256-SHA256 Secure Boot

Signing and verification for ECDSA-P256-SHA256 Secure Boot are described in Figure 2.12 ECDSA-P256-SHA256 Sign and Verify on page 16.

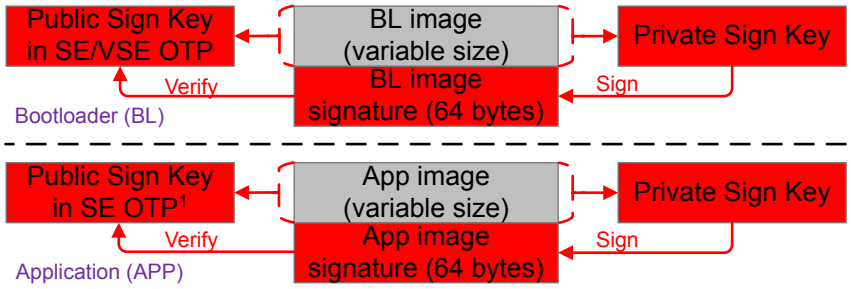


Figure 2.12. ECDSA-P256-SHA256 Sign and Verify

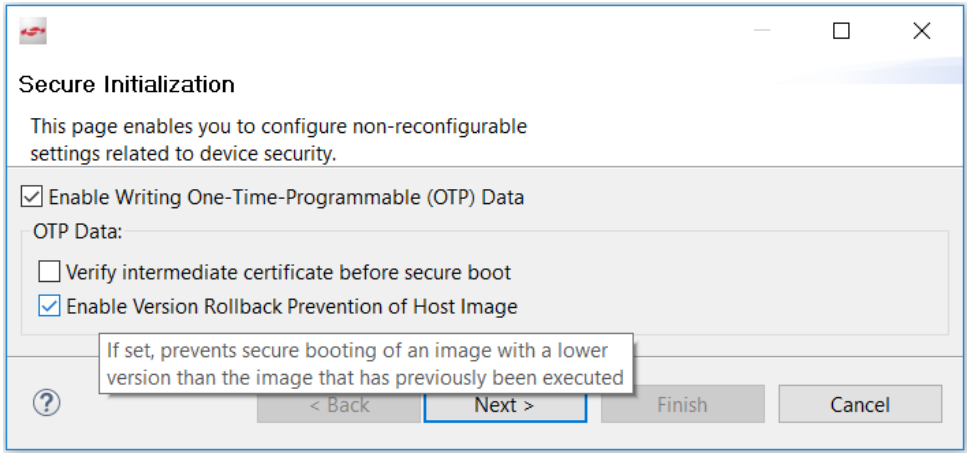
Note: The bootloader cannot access the Public Sign Key in VSE OTP to verify the application image. Therefore VSE devices need to store a Public Sign Key copy in the top page of the main flash (see section "*Key Storage*" in UG266: Silicon Labs Gecko Bootloader User's Guide).

Run the `flash` command to write the file (`sign_pubkey.txt` generated in 2.1.1 Using Simplicity Commander step 6) containing the Public Sign Key as a manufacturing token to the VSE device.

```
commander flash --tokengroup znet --tokenfile sign_pubkey.txt
```

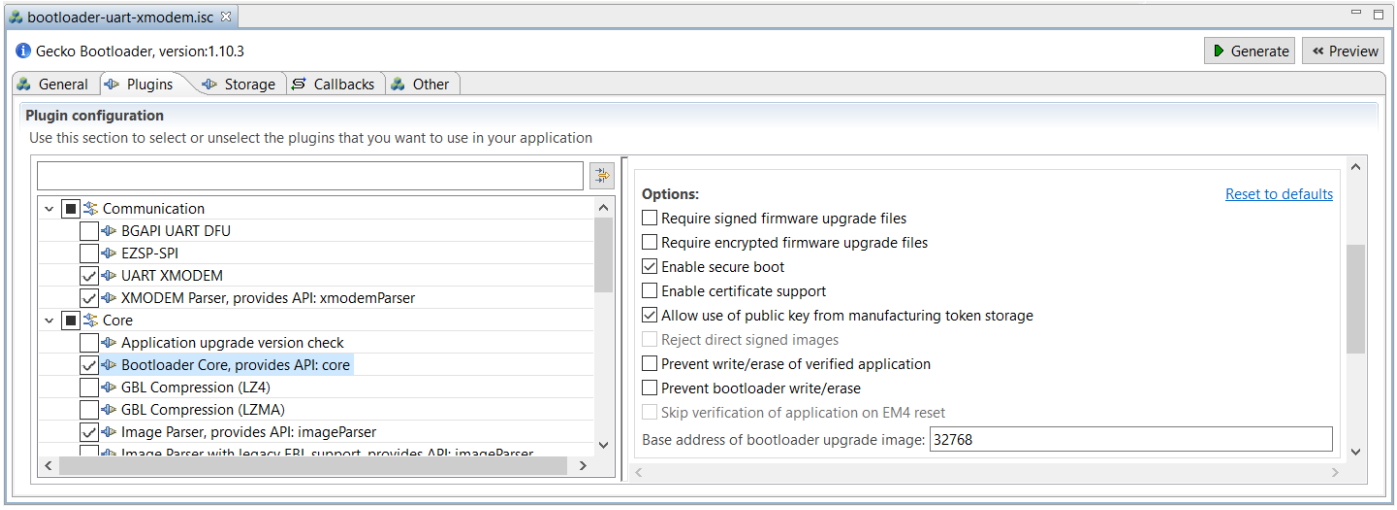
Secure Boot settings:

Use the Secure Boot settings in Figure 2.3 Secure Initialization Dialog Box on page 10 to provision the Public Sign Key for ECDSA-P256-SHA256 Secure Boot.



Bootloader image file:

1. Open the `bootloader-uart-xmodem.isc` file, and check the **Enable secure boot** option in AppBuilder's **Core** Plugin to enable Secure Boot in the application image. Click [**Generate**] to save the change, then build the project to generate the `bootloader-uart-xmodem.s37` file.



2. Run the `convert` command with **Private Sign Key** to generate the signed bootloader image file (`bootloader-uart-xmodem-signed.s37`).

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

```
Parsing file bootloader-uart-xmodem.s37...
Image SHA256: c4edf30cd3fef23790141720402e9b0bf670ed06901b10bf16ff4764a2bb738b
R = 4885A746098A895849238D80C5FF0B5EDCA2C74B0D781B44198D549A5C831AD8
S = 2D2524721147F0FA7464EA6E62858C4733295C4913889581FDDEE066D6395E17
Writing to bootloader-uart-xmodem-signed.s37...
DONE
```

3. Run the `util verifysign` command with **Public Sign Key** to verify that the file was correctly signed (optional).

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

```
Parsing file bootloader-uart-xmodem-signed.s37...
Found application properties at 0x00002960
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.
Found signature at 0x00002b88
Successfully verified application signature.
DONE
```

4. Run the `flash` command to program the `bootloader-uart-xmodem-signed.s37` file to the device.

```
commander flash bootloader-uart-xmodem-signed.s37 --device EFR32MG21A010F1024 --serialno 440068705
```

```
ERROR: DCI command failed due to: Internal error
WARNING: DCI communication failed, trying again after reset...
WARNING: Failed secure boot detected. Issuing a mass erase before flashing to recover the device...
Parsing file bootloader-uart-xmodem-signed.s37...
Writing 16384 bytes starting at address 0x00000000
Comparing range 0x00000000 - 0x00003FFF (16 KB)
Programming range 0x00000000 - 0x00001FFF (8 KB)
Programming range 0x00002000 - 0x00003FFF (8 KB)
Verifying range 0x00000000 - 0x00003FFF (16 KB)
DONE
```

Application image file:

1. The application image should contain an `application_properties.c` file as shown below for ECDSA-P256-SHA256 Secure Boot. The `cert` is set to `NULL` to disable the application certificate option. The `signatureType` and `signatureLocation` are filled by Simplicity Commander when signing the application image using the `convert` command.

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

// Application version number (uint32_t) for anti-rollback
#define APP_PROPERTIES_VERSION  (0UL)

// Application properties for secure boot
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 = APP_PROPERTIES_VERSION,
    .capabilities = 0UL,
    .productId = { 0U },
  },
  .cert = NULL,
  .longTokenSectionAddress = NULL,
};
```

2. Run the `convert` command with **Private Sign Key** to generate the signed application image file (`blink-signed.s37`).

```
commander convert blink.s37 --secureboot --keyfile sign_key.pem --outfile blink-signed.s37
```

```
Parsing file blink.s37...
Image SHA256: a2d5af113a6b65c7d815e68b518335b17e9cc4196b7e4b294cb7e2f4531ea926
R = B79D7AD20629F617E32F08C207867541965945181FD282F0AFE2F98EAFDECE94
S = 3A0E0CDEF3F48D24D8161E4FF5AEC7C685EF871B62B38F8532329323290215AE
Writing to blink-signed.s37...
DONE
```

3. Run the `util verifysign` command with **Public Sign Key** to verify that the file was correctly signed (optional).

```
commander util verifysign blink-signed.s37 --verify sign_pubkey.pem
```

```
Parsing file blink-signed.s37...
Found application properties at 0x0000109c
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.
```

GBL upgrade file:

1. Run the `gbl create` command to generate the bootloader GBL upgrade file (`bootloader-uart-xmodem.gbl`).

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

```
Parsing file bootloader-uart-xmodem-signed.s37...
Initializing GBL file...
Adding bootloader to GBL...
Writing GBL file bootloader-uart-xmodem.gbl...
DONE
```

2. Run the `gbl create` command to generate the application GBL upgrade file (`blink.gbl`).

```
commander gbl create blink.gbl --app blink-signed.s37
```

```
Parsing file blink-signed.s37...
Initializing GBL file...
Adding application to GBL...
Writing GBL file blink.gbl...
DONE
```

2.3.3 Certificate-Based Secure Boot

Signing and verification for certificate-based Secure Boot are described in Figure 2.13 Certificate-Based Sign and Verify on page 20.

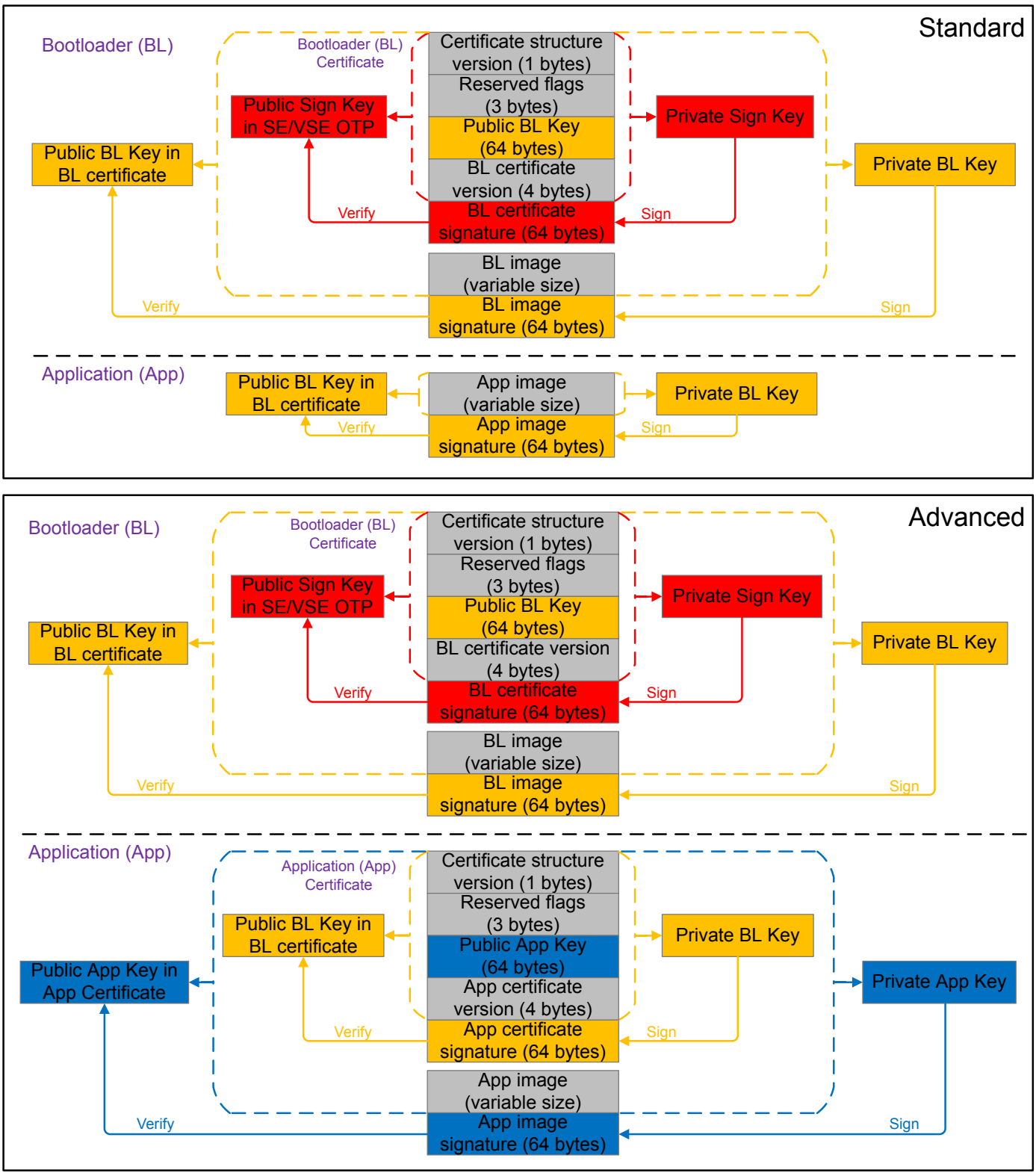


Figure 2.13. Certificate-Based Sign and Verify

Certificate:

The elements of a certificate are described in Table 2.2 Certificate Structure on page 21.

Table 2.2. Certificate Structure

Element	Description
Certificate structure version	The version of the certificate structure.
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.

Private/Public Key pair:

For certificate-based Secure Boot, three Private/Public Key pairs are used in different certificates (Table 2.3 Certificates and Key Pairs for Certificate-Based Secure Boot Examples on page 21). These key pairs can be generated by Simplicity Commander as in 2.1.1 Using Simplicity Commander step 5.

Table 2.3. Certificates and Key Pairs for Certificate-Based Secure Boot Examples

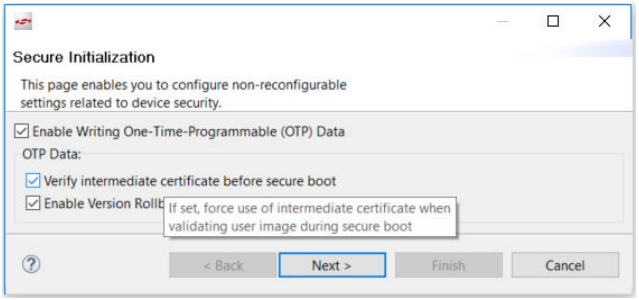
Certificate	Private Key	Public Key	Description
Bootloader (`bloader_cert.bin`)[1]	bloadercert_key.pem	bloadercert_pubkey.pem	The bootloader certificate is signed by the **Private Sign Key** corresponding to the Public Sign Key in Secure Element OTP.
Application (`app_cert.bin`)[2]	appcert_key.pem	appcert_pubkey.pem	The application certificate is signed by the **Private Bootloader Key** in the bootloader certificate.
GBL (`gbl_cert.bin`)[2]	gblcert_key.pem	gblcert_pubkey.pem	The GBL certificate is signed by the **Private Bootloader Key** in the bootloader certificate.

Note:

1. a. Certificate version in the bootloader certificate < certificate version in Secure Element - the certificate is rejected.
   b. Certificate version in the bootloader certificate = certificate version in Secure Element - the certificate is accepted.
   c. Certificate version in the bootloader certificate > certificate version in Secure Element - the certificate is accepted and the certificate version in Secure Element is updated to match (revocation mechanism).
2. The certificate version in both the Application and the GBL certificate is compared with the certificate version in the bootloader certificate. The Application or GBL certificate is accepted if its version is equal to or higher than certificate version in the bootloader certificate.

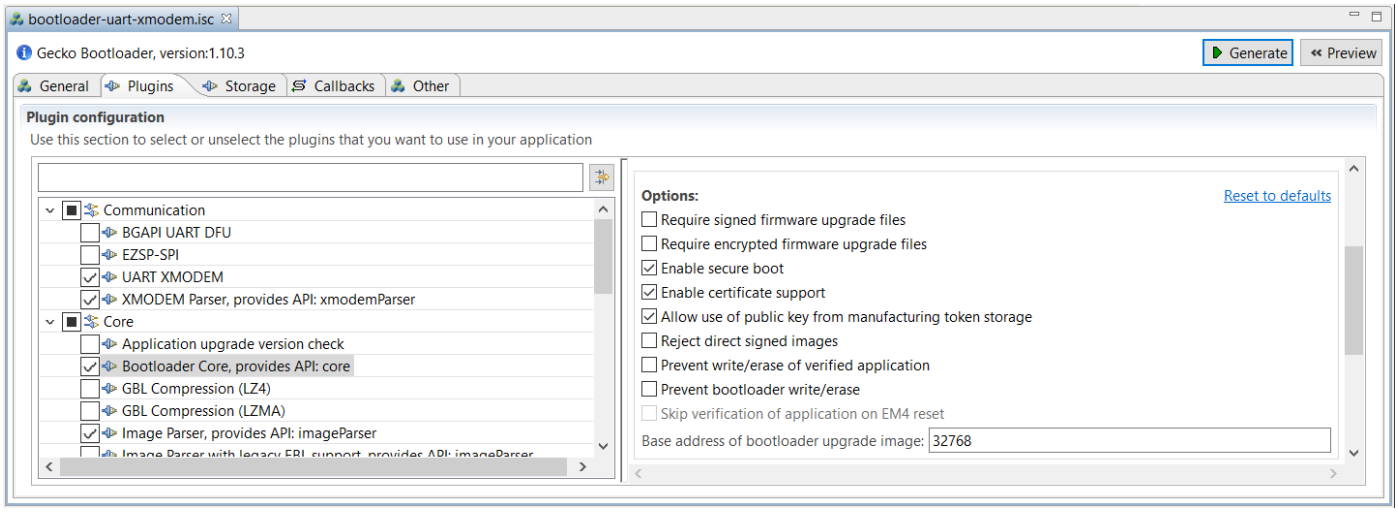
Secure Boot settings:

Check the **Verify intermediate certificate before secure boot** option in the Figure 2.3 Secure Initialization Dialog Box on page 10 to provision the Public Sign Key to enable certificate-based Secure Boot.



Bootloader image file:

1. Open the `bootloader-uart-xmodem.isc` file, and check the **Enable secure boot** and **Enable certificate support** options in App-Builder's **Core** Plugin to enable certificate-based Secure Boot in the application image. Click [**Generate**] to save the changes, then build the project to generate the `bootloader-uart-xmodem.s37` file.



2. Run the `util gencert` command with **Public Bootloader Key** and **Private Sign Key** to generate the bootloader certificate (`bloader_cert.bin`).

```
commander util gencert --cert-type secureboot --cert-version 0 --cert-pubkey bloadercert_pubkey.pem --sign
sign_key.pem --outfile bloader_cert.bin
```

```
Successfully signed certificate
DONE
```

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

```
commander convert bootloader-uart-xmodem.s37 --secureboot --certificate bloader_cert.bin --keyfile
bloadercert_key.pem --outfile bootloader-uart-xmodem-signed.s37
```

```
Parsing file bootloader-uart-xmodem.s37...
Writing certificate to location 0x00002960
Private key matches public key in certificate.
Image SHA256: 0d007d1c53a0c89da83f8c9dbf91a96f3a80e5a98b1edd4d2fa4e694f86488c5
R = 979E36B083C61180DBF9BB60AC6C418D3D49B0582E96796515410EA7EDBB1F85
S = 0948149CB0C22B100BAA4CFBEF4D6FD5A50626BA2DF0ED183A432D8201EC3570

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

4. Run the `util verifysign` command with **Public Sign Key** to verify that the **Bootloader Certificate** was correctly signed (optional).

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

```
Parsing file bootloader-uart-xmodem-signed.s37...
Found application properties at 0x000029e8
Found certificate at 0x00002960
Successfully verified certificate signature with verification key.
Using certificate key to verify application signature.
Found signature at 0x00002c10
Successfully verified application signature.
DONE
```

5. Run the `flash` command to program the `bootloader-uart-xmodem-signed.s37` file to the device.

```
commander flash bootloader-uart-xmodem-signed.s37 --device EFR32MG21A010F1024 --serialno 440068705
```

```
ERROR: DCI command failed due to: Internal error
WARNING: DCI communication failed, trying again after reset...
WARNING: Failed secure boot detected. Issuing a mass erase before flashing to recover the device...
Parsing file bootloader-uart-xmodem-signed.s37...
Writing 16384 bytes starting at address 0x00000000
Comparing range 0x00000000 - 0x00003FFF (16 KB)
Programming range 0x00000000 - 0x00001FFF (8 KB)
Programming range 0x00002000 - 0x00003FFF (8 KB)
Verifying range 0x00000000 - 0x00003FFF (16 KB)
DONE
```

Application image file (standard certificate-based):

1. The application image should contain an `application_properties.c` file, as shown below, for standard certificate-based Secure Boot. The `cert` is set to `NULL` to disable the application certificate option and the • Reject direct signed images option in AppBuilder's **Core** plugin should be unchecked. The `signatureType` and `signatureLocation` are filled by Simplicity Commander when signing the application image using the `convert` command.

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

// Application version number (uint32_t) for anti-rollback
#define APP_PROPERTIES_VERSION  (0UL)

// Application properties for secure boot
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 = APP_PROPERTIES_VERSION,
    .capabilities = 0UL,
    .productId = { 0U },
  },
  .cert = NULL,
  .longTokenSectionAddress = NULL,
};
```

2. Run the `convert` command with **Private Bootloader Key** to generate the signed application image file (`blink-signed.s37`).

```
commander convert blink.s37 --secureboot --keyfile bloadercert_key.pem --outfile blink-signed.s37
```

```
Parsing file blink.s37...
Image SHA256: a2d5af113a6b65c7d815e68b518335b17e9cc4196b7e4b294cb7e2f4531ea926
R = 3646A44F91DF15B440C59051AD458BEA4F0E9209A364397467EFEA408A7D1458
S = 2F049F04ACCCD94F32DDBAE9FFB57A5B93B6127A8F2D4825B86F5213285F399A
Writing to blink-signed.s37...
DONE
```

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

```
commander util verifysign blink-signed.s37 --verify bloadercert_pubkey.pem
```

```
Parsing file blink-signed.s37...
Found application properties at 0x0000109c
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.
```

Application image file (advanced certificate-based):

1. The application image should contain an `application_properties.c` file, as shown below, for advanced certificate-based Secure Boot. The `cert` is pointed to a valid `ApplicationCertificate_t` structure to enable the application certificate option. The `key`, `version` and `signature` in `sl_app_certificate` are filled by Simplicity Commander when signing the application certificate using `util gencert` command.

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

// Application version number (uint32_t) for anti-rollback
#define APP_PROPERTIES_VERSION  (0UL)

// Application properties for secure boot
const ApplicationCertificate_t sl_app_certificate = {
  .structVersion = APPLICATION_CERTIFICATE_VERSION,
  .flags = { 0U },
  .key = { 0U },
  .version = 0,
  .signature = { 0U },
};

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 = APP_PROPERTIES_VERSION,
    .capabilities = 0UL,
    .productId = { 0U },
  },
  .cert = (ApplicationCertificate_t *)&sl_app_certificate,
  .longTokenSectionAddress = NULL,
};
```

2. Run the `util gencert` command with **Public Application Key** and **Private Bootloader Key** to generate the application certificate (`app_cert.bin`).

```
commander util gencert --cert-type secureboot --cert-version 0 --cert-pubkey appcert_pubkey.pem --sign
bloadercert_key.pem --outfile app_cert.bin
```

```
Successfully signed certificate
DONE
```

3. Run the `convert` command with **Application Certificate** and **Private Application Key** to generate the signed application image file (`blink-signed.s37`).

```
commander convert blink.s37 --secureboot --certificate app_cert.bin --keyfile appcert_key.pem --outfile
blink-signed.s37
```

```
Parsing file blink.s37...
Writing certificate to location 0x00004f70
Private key matches public key in certificate.
Image SHA256: cde34993c86d642fe1a5d3ea2c77c6ca0ddad5d8882c765ec6fe68cbdb8904a0
R = 3B843DE62958D785A27B900500E9BDEA7D2A9A78B2719540D01E48B4E1CF8956
S = 16D73AD5A2F4E9C7AE9BD5BE4D9F52CCCCBE489C8F9CEFCF7B29CC988F740EBB

Verifying signed image...
Writing to blink-signed.s37...
DONE
```

4. Run the `util verifysign` command with **Public Bootloader Key** to verify that the **Application Certificate** was correctly signed (optional).

```
commander util verifysign blink-signed.s37 --verify bloadercert_pubkey.pem
```

```
Parsing file blink-signed.s37...
Found application properties at 0x00001124
Found certificate at 0x00004f70
Successfully verified certificate signature with verification key.
Using certificate key to verify application signature.
```

GBL upgrade file using a bootloader certificate:

1. The application GBL upgrade file can be signed by the Private Bootloader Key as in Figure 2.14 Application GBL Upgrade File Using a Bootloader Certificate on page 25. A similar operation can apply to bootloader and secure element GBL upgrade files.

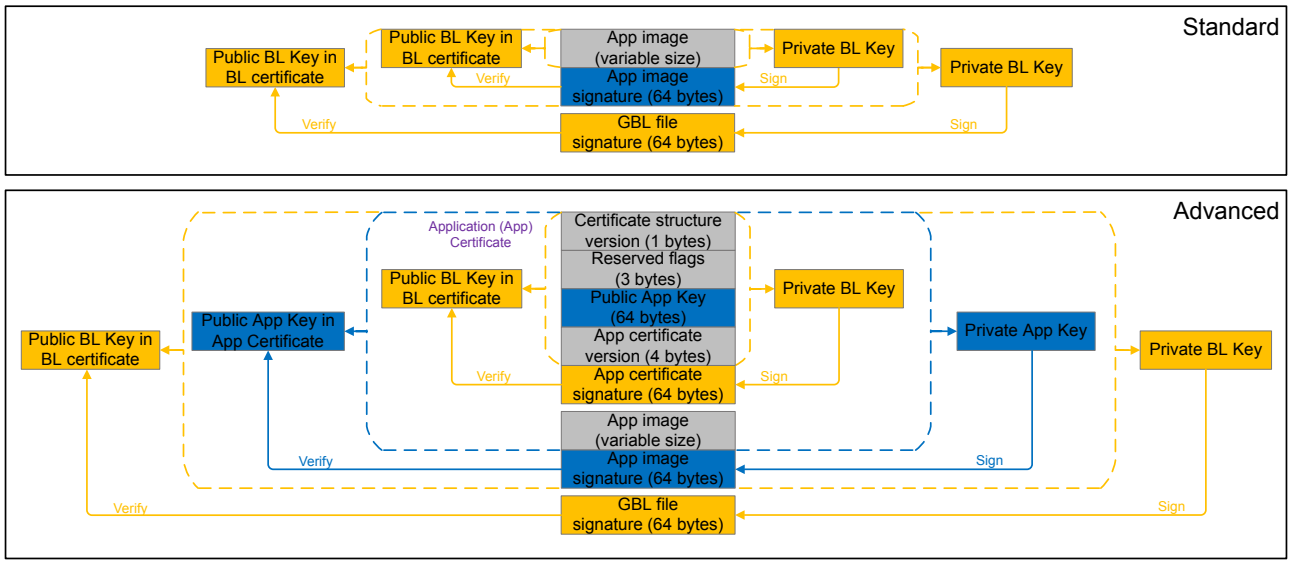


Figure 2.14. Application GBL Upgrade File Using a Bootloader Certificate

2. Run the `gbl create` command with **Private Bootloader Key** to generate the application GBL upgrade file (`blink-sgined.gbl`).

```
commander gbl create blink-signed.gbl --app blink-signed.s37 --sign bloadercert_key.pem
```

```
Parsing file blink-signed.s37...
Initializing GBL file...
Adding application to GBL...
Signing GBL...
Image SHA256: 08d541cf7b7f3b43a5c0dd637593071da731867b5da641b72188536314ce3889
R = C816EA97714654D86B91F64B85616CAC8DDCF12B885FAA7847D2CEA38940CA88
S = 32EDB05CDBCCDE6C1D3A7B2973B7F9F10D487CA00F15414684AE8E1758F5394D
Writing GBL file blink-signed.gbl...
DONE
```

GBL upgrade file using a GBL certificate:

1. The application GBL upgrade file can also be signed by the Private GBL key as in Figure 2.14 Application GBL Upgrade File Using a Bootloader Certificate on page 25. A similar operation can apply to bootloader and secure element GBL upgrade files.

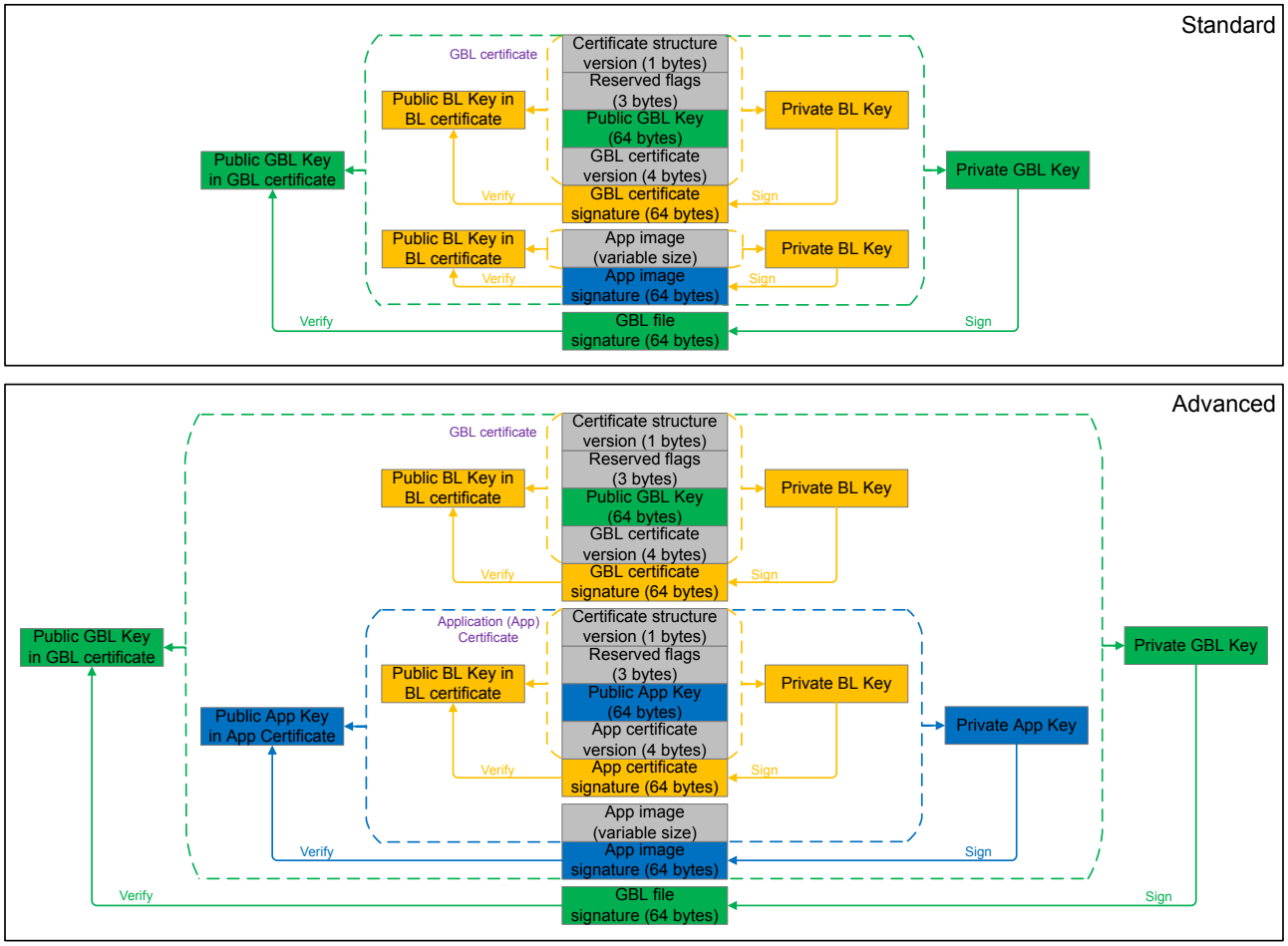


Figure 2.15. Application GBL Upgrade File Using a GBL Certificate

2. Run the `util gencert` command with **Public GBL Key** and **Private Bootloader Key** to generate the GBL certificate (`gbl_cert.bin`).

```
commander util gencert --cert-type gbl --cert-version 0 --cert-pubkey gblcert_pubkey.pem --sign
bloadercert_key.pem --outfile gbl_cert.bin
```

```
Successfully signed certificate
DONE
```

Note: The `--cert-type` is `gbl` instead of `secureboot`.

3. Run the `gbl create` command with **GBL Certificate** and **Private GBL key** to generate the application GBL upgrade file (`blink-signed-cert.gbl`).

```
commander gbl create blink-signed-cert.gbl --app blink-signed.s37 --certificate gbl_cert.bin --sign gblcert_key.pem
```

```
Parsing file blink-signed.s37...
Initializing GBL file...
Adding application to GBL...
Adding certificate to GBL...
Signing GBL...
Image SHA256: f034b5eea97a4f726e5aea97eacde8103f7d48575f1bb6e889487c8c72accd5d
R = 550AE0537EC7240D2A1566AA23808C355A0A01ECFF6621D6676DFD9EB4DCB4DB
S = 548AF527B1DC961E0600038646D660E2AF3955EBB79EEE535B5C839AD60AA1B5
Writing GBL file blink-signed-cert.gbl...
DONE
```

2.4 Recover Devices when Secure Boot Fails

If a Secure Boot process fails (meaning firmware image validation fails), the only way to recover is to flash a correctly-signed image. This section describes two methods by which to flash a correctly-signed image.

2.4.1 Simplicity Commander (GUI)

1. Run `commander` to open the Simplicity Commander GUI.

```
commander
```

2. Connect Simplicity Commander to a Wireless Starter Kit (WSTK) and click [**Flash**].

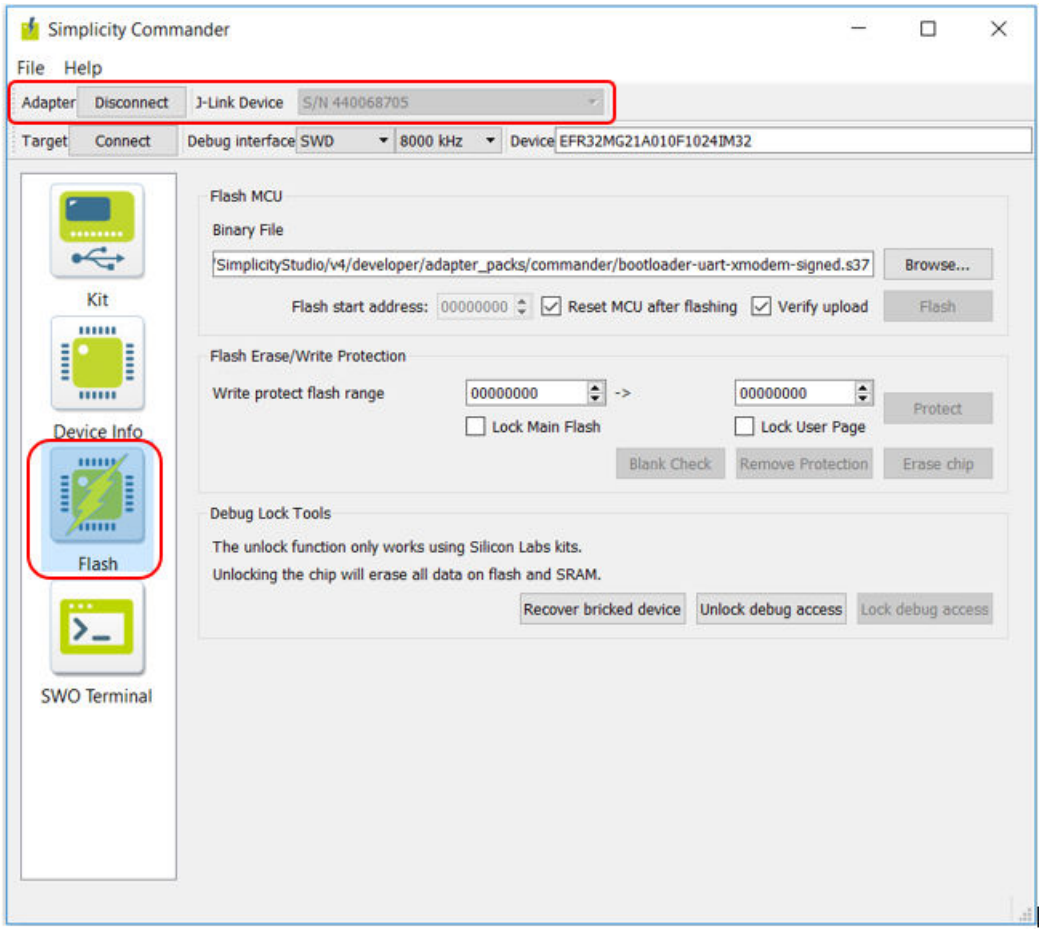


Figure 2.16. Connect Simplicity Commander to a WSTK

3. Click [**Browse…**] to select the correctly-signed image (for example `bootloader-uart-xmodem-signed.s37`) from the file system. Click [**Connect**] next to **Target**, then click [**OK**] to exit.

4. Click [**Flash**] to flash the correctly-signed image to the device. If a failed Secure Boot is detected, the device will be erased and unlocked before flashing the new image.

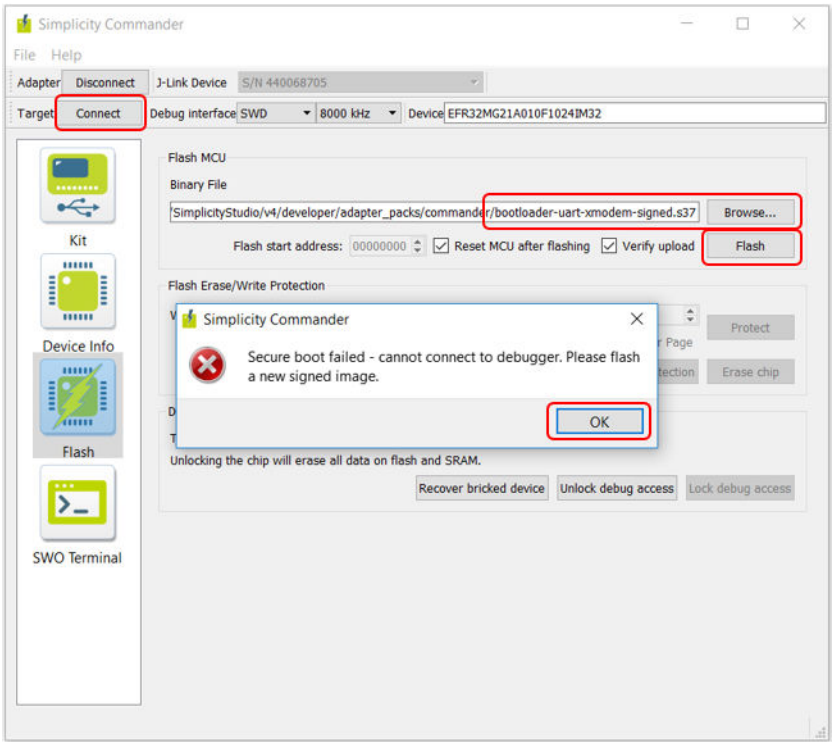


Figure 2.17. Flash Correctly-Signed Image

5. Click [**Connect**] next to **Target**, then click [**Device Info**] to verify the device is recovered.

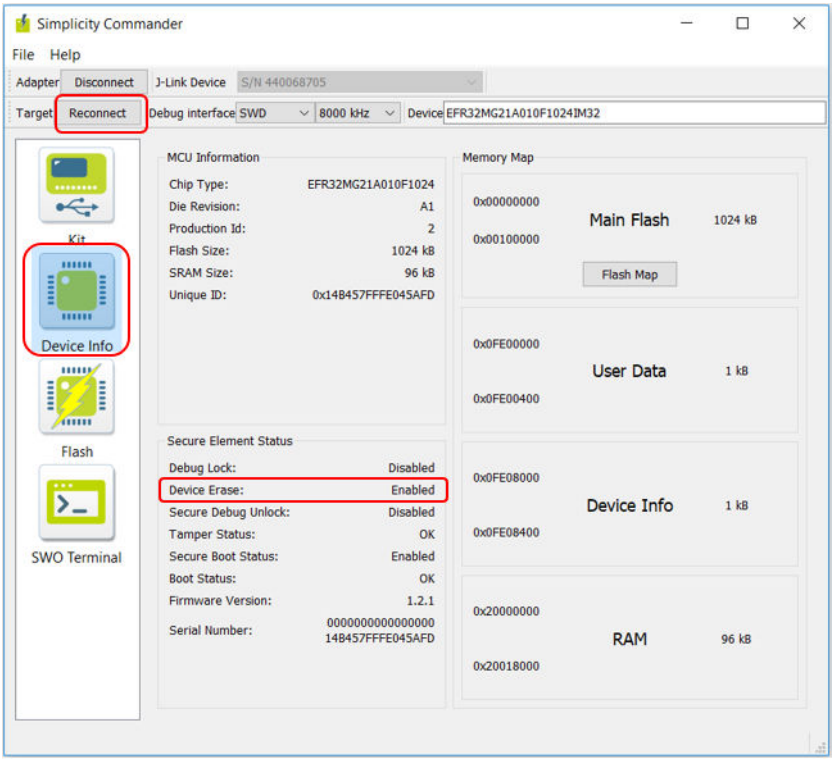


Figure 2.18. Device Information on Recovered Device

Note: The device cannot recover if **Device Erase** has been disabled.

2.4.2 Simplicity Commander (CLI)

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

```
commander flash bootloader-uart-xmodem-signed.s37 --device EFR32MG22C224F512 --serialno 440068705
```

```
WARNING: Failed secure boot detected. Issuing a mass erase before flashing to recover the device...
Parsing file bootloader-uart-xmodem-signed.s37...
Flashing 16384 bytes to address 0x00000000
Uploading flash loader...
Waiting for flashloader to become ready...
Erasing flash...
Flashing...
Verifying written data...
Resetting...
Finished!
DONE
```

Note: The device cannot recover if **Device Erase** has been disabled.

2.5 Upgrade to Secure Boot with RTSL

This section describes how to upgrade Series 2 devices already deployed in the field to Secure Boot with RTSL. The Public Sign Key here is copied from `sign_pubkey.txt` generated in 2.1.1 Using Simplicity Commander step 6.

General:

1. Upgrade Secure Element firmware to the latest version. See section "*Gecko Bootloader Operation - Secure Element Upgrade*" in UG266: Silicon Labs Gecko Bootloader User's Guide.
2. Follow the procedure in section "*Enabling Secure Boot RTSL on Series 2 Devices*" in UG266: Silicon Labs Gecko Bootloader User's Guide. The applications mentioned in procedure 3 (install Public Sign Key) and 5 (enable Secure Boot) are described in the following sections.
3. `emlib` provides an abstraction of the mailbox interface, allowing the application to construct messages and set up DMA transfers.
4. On top of `emlib`, Secure Element Manager provides an abstraction of the Secure Element's command set. The Secure Element Manager also provides APIs for cryptographic operations and thread synchronization. The Secure Element Manager is available in **GSDK v3.0** or later.
5. The code in ● Application to enable Secure Boot using `emlib` (SE without Secure Vault only): on page 32 and ● Application to enable Secure Boot using Secure Element Manager: on page 34 is based on ECDSA-P256-SHA256 Secure Boot. The `otpConfig` structure contains the desired Secure Boot settings described in section "*Secure Boot Enabling*" in AN1222: Production Programming of Series 2 Devices.

Note: The `emlib` functions used here are deprecated in **GSDK v3.0** and will be removed in a future version of `emlib`. All high-level functionality has been moved to the Secure Element Manager.

Application to install Public Sign Key using `emlib` (SE only):

```
#include "em_chip.h"
#include "em_common.h"
#include "em_se.h"
#include <string.h>

SL_ALIGN(4) static uint8_t keyBuffer[64] =      // Public Sign Key
  {
    0x99, 0x70, 0x11, 0xED, 0x17, 0x08, 0x58, 0x0B,
    0xD4, 0xA6, 0xB7, 0xF8, 0xAD, 0x6E, 0xE1, 0x9B,
    0x0B, 0x87, 0x22, 0x61, 0x1F, 0xB7, 0x6A, 0x3A,
    0x57, 0x02, 0xD5, 0x14, 0x11, 0x80, 0xE1, 0x01,
    0x0A, 0xC8, 0x67, 0x3C, 0x8A, 0xCC, 0x26, 0xEE,
    0x2B, 0x53, 0x4C, 0x00, 0x4F, 0x4A, 0x4B, 0x7E,
    0xBB, 0xC2, 0x3D, 0x04, 0x50, 0x6D, 0xD6, 0x6E,
    0x3E, 0xF0, 0xDD, 0xC8, 0x1E, 0x3C, 0xA5, 0x5E
  };
SL_ALIGN(4) static uint8_t memBuffer[64];       // Buffer for key verification

/***************************************************************************//**
 * @brief  Main function
 ******************************************************************************/
int main(void)
{
  // Chip errata
  CHIP_Init();

  // Main loop
  if (SE_initPubkey(SE_KEY_TYPE_BOOT, keyBuffer, 64, false) != SE_RESPONSE_OK) {
    while (1) ;         // Public Sign Key write error
  } else {
    if (SE_readPubkey(SE_KEY_TYPE_BOOT, memBuffer, 64, false) == SE_RESPONSE_OK) {
      if (memcmp(memBuffer, keyBuffer, 64) != 0) {
        while (1) ;     // Public Sign Key verification fail
      }
    } else {
      while (1) ;       // Public Sign Key read error
    }
  }

  while (1) ;           // Public Sign Key provisioning done
}
```

Application to enable Secure Boot using `emlib` (SE without Secure Vault only):

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

// Application version number (uint32_t) for anti-rollback
#define APP_PROPERTIES_VERSION  (0UL)

// Application properties for secure boot
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 = APP_PROPERTIES_VERSION,
    .capabilities = 0UL,
    .productId = { 0U },
  },
  .cert = NULL,
  .longTokenSectionAddress = NULL,
};

static SE_OTPInit_t otpConfig =
  {
    .enableSecureBoot = true,                   // Enable secure boot
    .verifySecureBootCertificate = false,       // No certificate
    .enableAntiRollback = true,                 // Enable anti-rollback
    .secureBootPageLockNarrow = false,          // No lock
    .secureBootPageLockFull = false             // No lock
  };
static SE_Status_t status;

/***************************************************************************//**
 * @brief  Main function
 ******************************************************************************/
int main(void)
{
  // Chip errata
  CHIP_Init();

  if (SE_initOTP(&otpConfig) != SE_RESPONSE_OK) {
    while (1) ;         // Secure boot enable write error
  } else {
    if (SE_getStatus(&status) == SE_RESPONSE_OK) {
      if (status.secureBootEnabled == false) {
        while (1) ;     // Secure boot enable verification fail
      }
    } else {
      while (1);        // Secure boot enable read error
    }
  }

  while (1) ;           // Secure boot enable done
}
```

Application to install Public Sign Key using Secure Element Manager:

```
#include "em_chip.h"
#include "em_common.h"
#include "sl_se_manager.h"
#include "sl_se_manager_util.h"
#include <string.h>

SL_ALIGN(4) static uint8_t keyBuffer[64] =      // Public Sign Key
  {
    0x99, 0x70, 0x11, 0xED, 0x17, 0x08, 0x58, 0x0B,
    0xD4, 0xA6, 0xB7, 0xF8, 0xAD, 0x6E, 0xE1, 0x9B,
    0x0B, 0x87, 0x22, 0x61, 0x1F, 0xB7, 0x6A, 0x3A,
    0x57, 0x02, 0xD5, 0x14, 0x11, 0x80, 0xE1, 0x01,
    0x0A, 0xC8, 0x67, 0x3C, 0x8A, 0xCC, 0x26, 0xEE,
    0x2B, 0x53, 0x4C, 0x00, 0x4F, 0x4A, 0x4B, 0x7E,
    0xBB, 0xC2, 0x3D, 0x04, 0x50, 0x6D, 0xD6, 0x6E,
    0x3E, 0xF0, 0xDD, 0xC8, 0x1E, 0x3C, 0xA5, 0x5E
  };
SL_ALIGN(4) static uint8_t memBuffer[64];       // Buffer for key verification

/***************************************************************************//**
 * @brief  Main function
 ******************************************************************************/
int main(void)
{
  sl_se_command_context_t cmdCtx;

  // Chip errata
  CHIP_Init();

  // Initialize the SE manager
  sl_se_init();

  // Main loop
  if (sl_se_init_otp_key(&cmdCtx, SL_SE_KEY_TYPE_IMMUTABLE_BOOT, keyBuffer, 64) != SL_STATUS_OK) {
    while (1) ;         // Public Sign Key write error
  } else {
    if (sl_se_read_pubkey(&cmdCtx, SL_SE_KEY_TYPE_IMMUTABLE_BOOT, memBuffer, 64) == SL_STATUS_OK) {
      if (memcmp(memBuffer, keyBuffer, 64) != 0) {
        while (1) ;     // Public Sign Key verification fail
      }
    } else {
      while (1) ;       // Public Sign Key read error
    }
  }

  // De-initialize the SE manager
  sl_se_deinit_command_context(&cmdCtx);
  sl_se_deinit();
  while (1) ;           // Public Sign Key provisioning done
}
```

Application to enable Secure Boot using Secure Element Manager:

```
#include "em_chip.h"
#include "sl_se_manager.h"
#include "sl_se_manager_util.h"
#include "application_properties.h"

// Application version number (uint32_t) for anti-rollback
#define APP_PROPERTIES_VERSION  (0UL)

// Application properties for secure boot
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 = APP_PROPERTIES_VERSION,
    .capabilities = 0UL,
    .productId = { 0U },
  },
  .cert = NULL,
  .longTokenSectionAddress = NULL,
};

static sl_se_otp_init_t otpConfig =
  {
    .enable_secure_boot = true,                 // Enable secure boot
    .verify_secure_boot_certificate = false,    // No certificate
    .enable_anti_rollback = true,               // Enable anti-rollback
    .secure_boot_page_lock_narrow = false,      // No lock
    .secure_boot_page_lock_full = false,        // No lock

    #if defined(SEMAILBOX_PRESENT) && (_SILICON_LABS_SECURITY_FEATURE == _SILICON_LABS_SECURITY_FEATURE_VAULT)
    // Tamper settings for SE with Secure Vault
    .tamper_levels = { 0, 1, 4, 0, 4, 4, 0, 4, 4, 0, 1, 0, 4, 0, 4, 1,
                       1, 1, 2, 2, 4, 4, 7, 7, 4, 2, 2, 2, 4, 0, 2, 4 },
    .tamper_filter_period = SL_SE_TAMPER_FILTER_PERIOD_2MIN,
    .tamper_filter_threshold = SL_SE_TAMPER_FILTER_THRESHOLD_4,
    .tamper_flags = 0,
    .tamper_reset_threshold = 5
    #endif
  };
static sl_se_otp_init_t status;

/*******************************************************************************//**
 * @brief  Main function
 ******************************************************************************/
int main(void)
{
  sl_se_command_context_t cmdCtx;

  // Chip errata
  CHIP_Init();

  // Initialize the SE manager
  sl_se_init();

  // Main loop
  if (sl_se_init_otp(&cmdCtx, &otpConfig) != SL_STATUS_OK) {
    while (1) ;         // Secure boot enable write error
  } else {
    if (sl_se_read_otp(&cmdCtx, &status) == SL_STATUS_OK) {
      if (status.enable_secure_boot == false) {
        while (1) ;     // Secure boot enable verification fail
      }
    } else {
      while (1);        // Secure boot enable read error
    }
  }

  // De-initialize the SE manager
  sl_se_deinit_command_context(&cmdCtx);
  sl_se_deinit();
  while (1) ;           // Secure boot enable done
}
```

3. Revision History

Revision 0.3

July 2020

- Added Secure Element conventions to 1.1 Introduction.
- Updated Figure 2.1 Debug Adapter Context Menu on page 9 and Figure 2.2 Configuration on Selected Device on page 9 to Simplicity Studio v5.
- Updated Simplicity Commander version to 1.9.2 in 2.1.1 Using Simplicity Commander.
- Renamed Provision Public Sign Key to 2.2 Provision Public Sign Key and Secure Boot Enabling, added note for SE with Secure Vault devices.
- Added 2.3.2 ECDSA-P256-SHA256 Secure Boot and 2.3.3 Certificate-Based Secure Boot to 2. Examples.
- Added Secure Element Manager examples to 2.5 Upgrade to Secure Boot with RTSL.
- Removed the Related Documents section in favor of web links in the text.

Revision 0.2

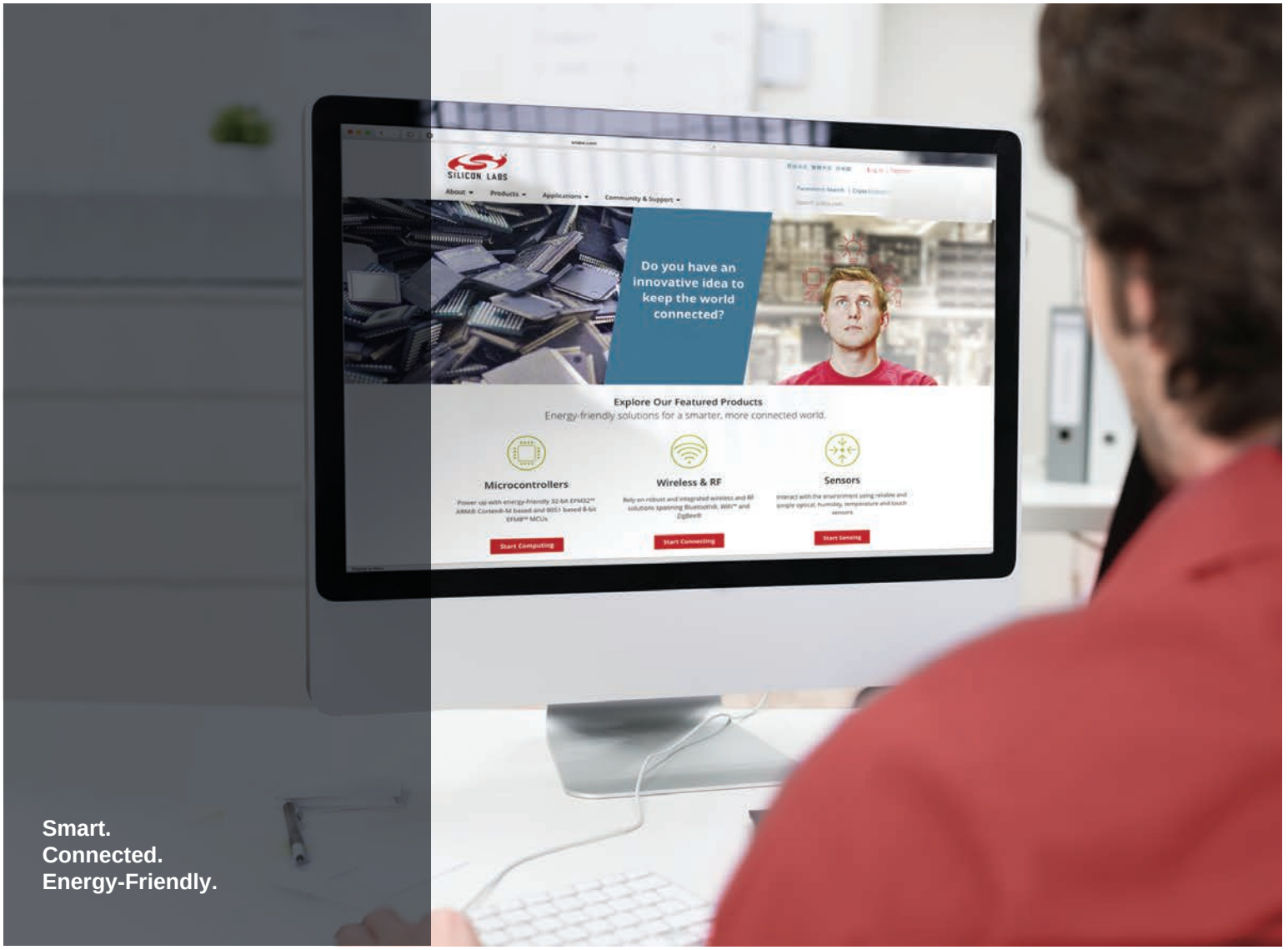
March 2020

- 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 Secure Boot (ECDSA) in Series 2 Devices section.
- Added Secure Boot (Certificate) in Series 2 Devices section.
- Added Upgrade to 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.



Products
www.silabs.com/products

Quality
www.silabs.com/quality

Support and Community
community.silabs.com

SILICON LABS

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

http://www.silabs.com