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.

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

For more information about Secure Element Subsystem (SE and VSE), see section "*Secure Element Subsystem*" in *AN1190: Series 2 Secure Debug*.

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

Device	Secure Element Subsystem	Minimum Firmware Version for Secure Boot with RTSL
EFR32xG21A	Secure Element (SE)	Version 1.1.2
EFR32xG22	Virtual Secure Element (VSE)	Version 1.2.1
Note: Silicon Labs strongly recommends installing the latest Secure Element Subsystem firmware on Series 2 devices.		

1.2 Secure Boot (ECDSA) 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) Process on page 3 illustrates the Secure Boot process on Series 1 devices.

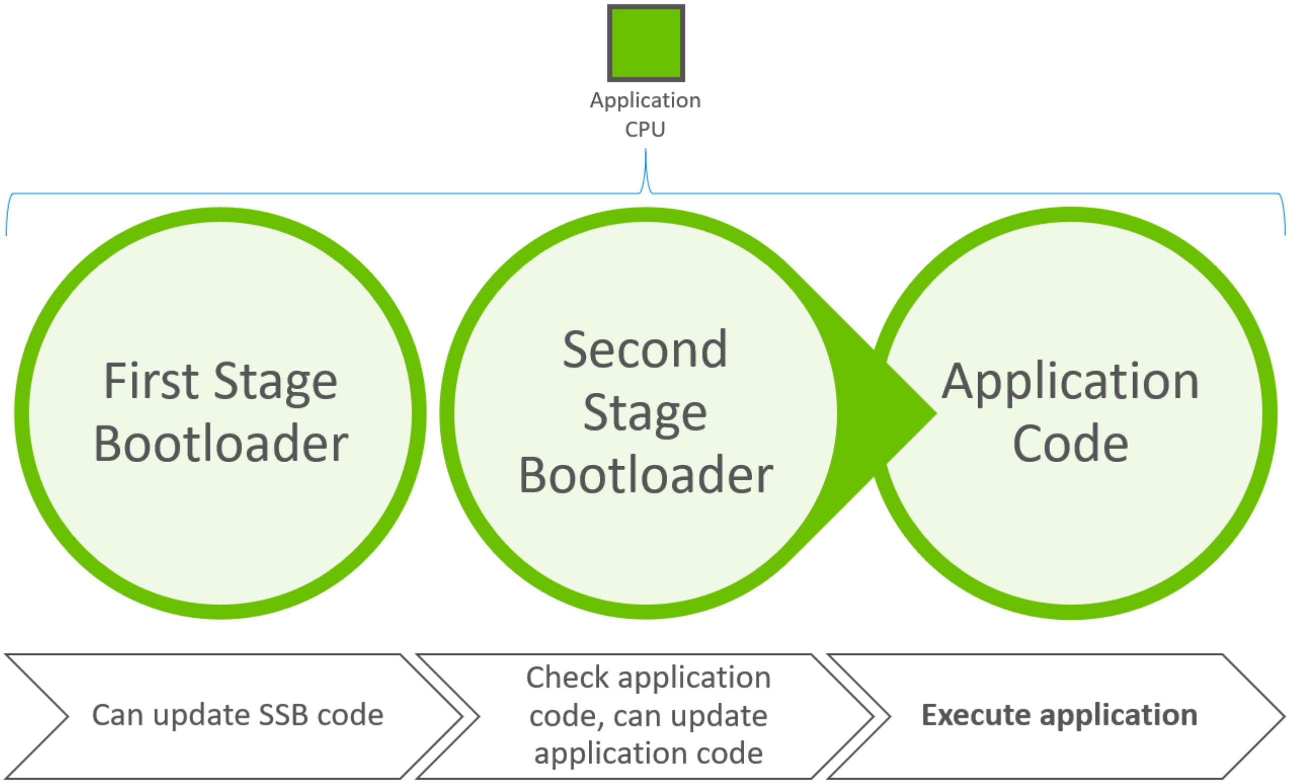


Figure 1.1. Series 1 Secure Boot (ECDSA) Process

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

1.3 Secure Boot (ECDSA) in Series 2 Devices

1.3.1 Secure Element (SE)

In Series 2 devices with Secure Element (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) Process on page 4 and Figure 1.3 Series 2 SE Secure Boot (ECDSA) Flow on page 4 illustrate the Secure Boot process and flow on Series 2 SE devices.

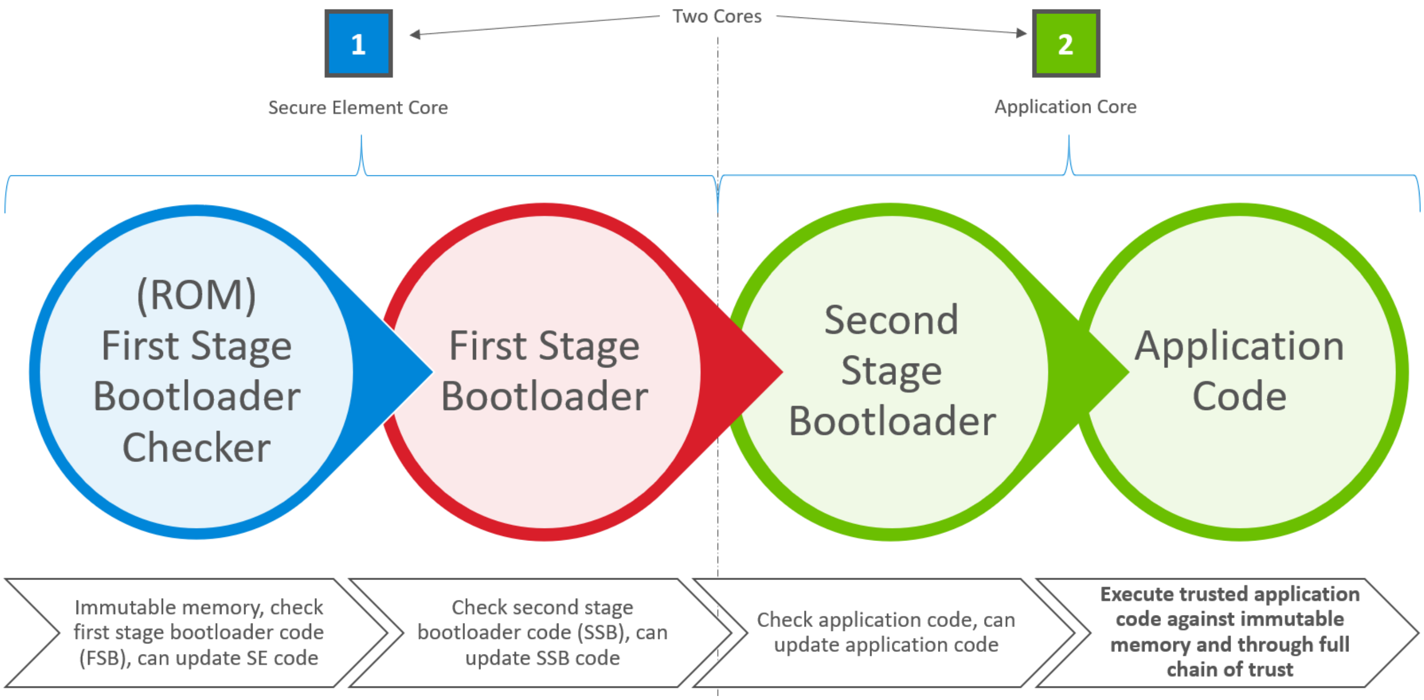


Figure 1.2. Series 2 SE Secure Boot (ECDSA) Process

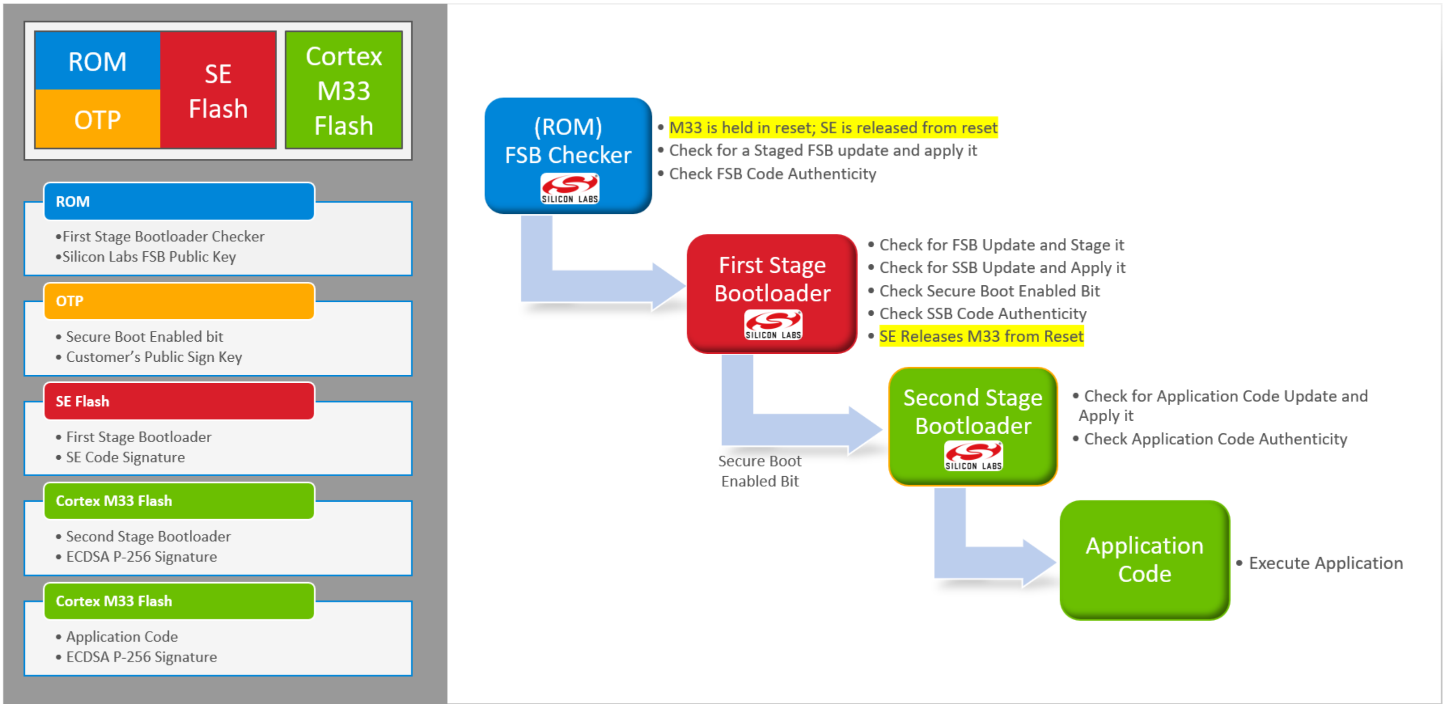


Figure 1.3. Series 2 SE Secure Boot (ECDSA) Flow

1.3.2 Virtual Secure Element (VSE)

In Series 2 devices with Virtual Secure Element (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) Process on page 5 and Figure 1.5 Series 2 VSE Secure Boot (ECDSA) Flow on page 5 illustrate the Secure Boot process and flow on Series 2 VSE devices.

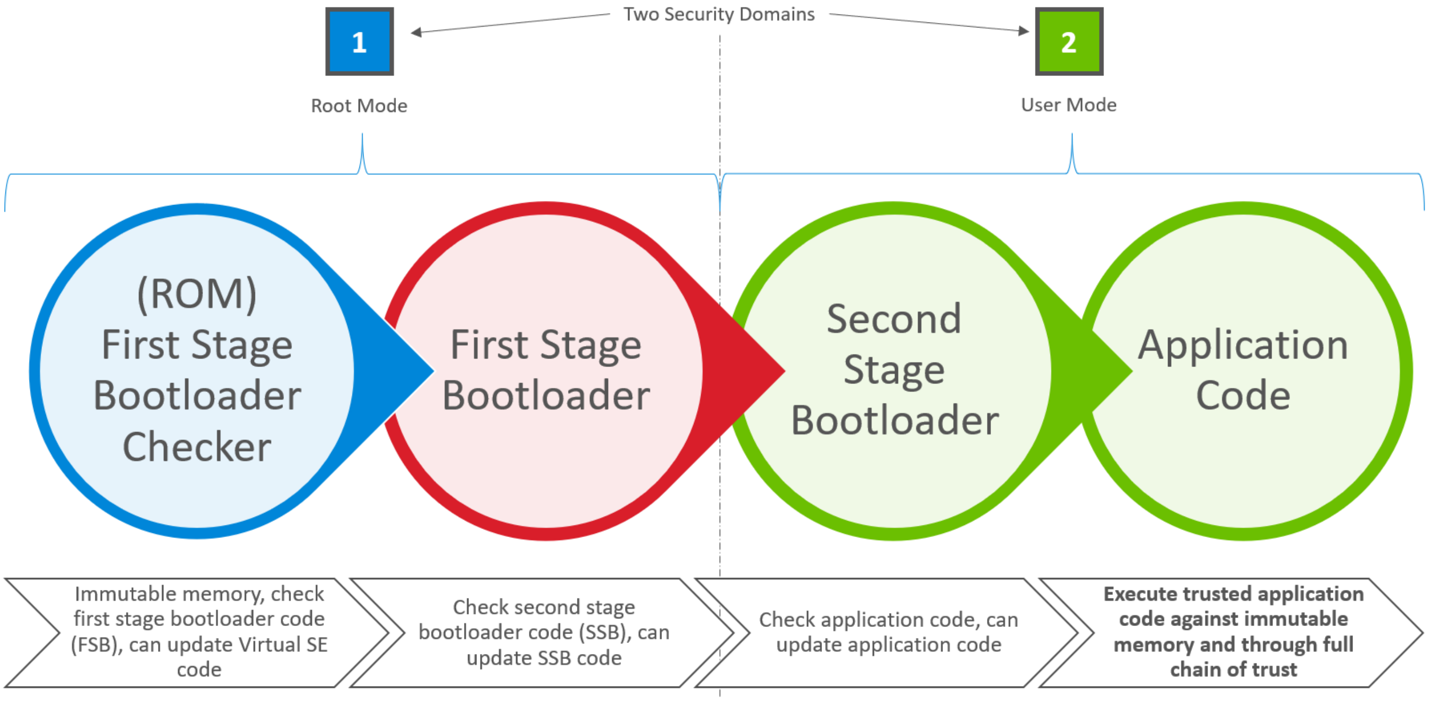


Figure 1.4. Series 2 VSE Secure Boot (ECDSA) Process

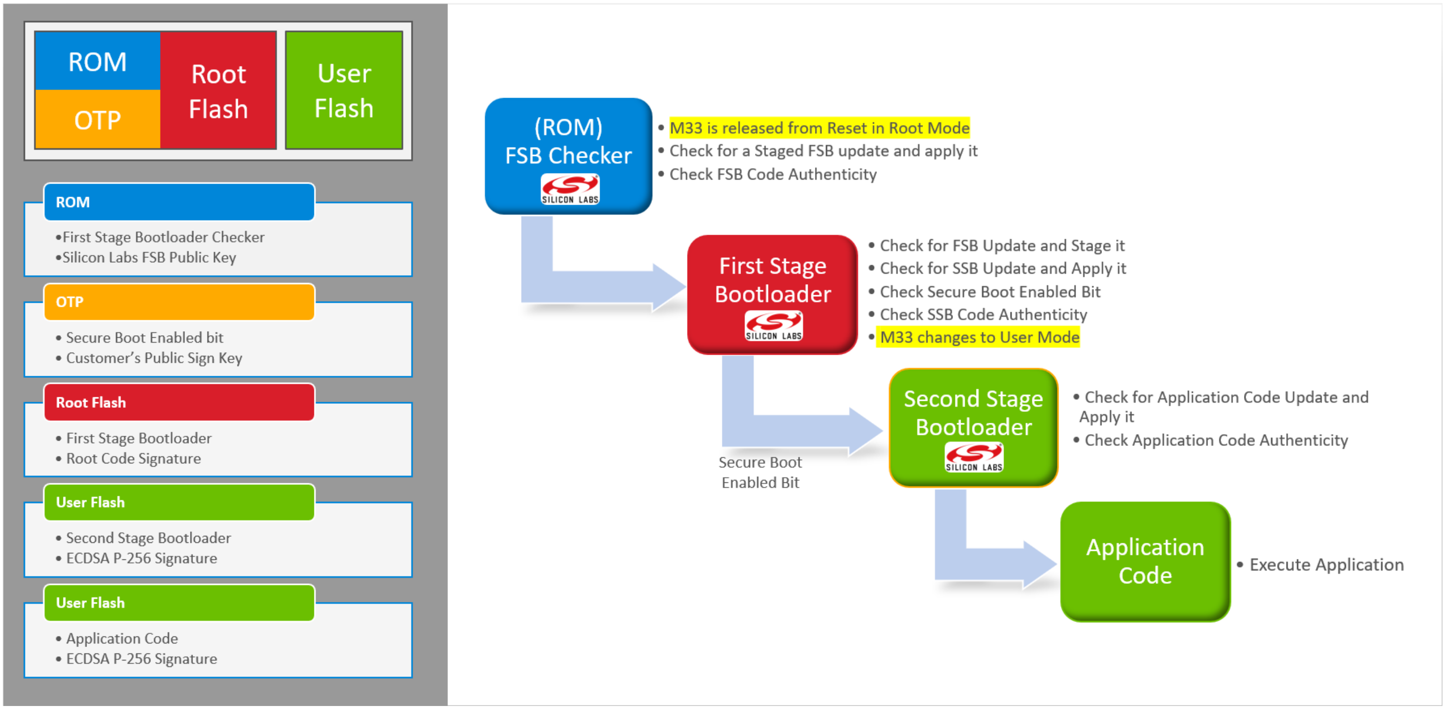


Figure 1.5. Series 2 VSE Secure Boot (ECDSA) Flow

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: Gecko Bootloader User's Guide*.

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 keypair 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, 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: 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	EFR32MG21A010F1024IM32	BRD4181A	Version 1.2.1	Simplicity Studio
	EFR32MG22C224F512IM40	BRD4182A	Version 1.2.1	Simplicity Commander
Recover devices when Secure Boot fails	EFR32MG21A010F1024IM32	BRD4181A	Version 1.2.1	Simplicity Commander (GUI)
	EFR32MG22C224F512IM40	BRD4182A	Version 1.2.1	Simplicity Commander (CLI)
Upgrade to Secure Boot with RTSL	EFR32MG21A010F1024IM32	BRD4181A	Version 1.2.1	Simplicity Commander

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 on Windows is `C:\SiliconLabs\SimplicityStudio\v4\developer\adapter_packs\commander`.
2. Simplicity Commander Version 1.8.2 is used in this application note.

```
commander --version
```

```
Simplicity Commander 1v8p2b708

JLink DLL version: 6.56a
Qt 5.12.1 Copyright (C) 2017 The Qt Company Ltd.
EMDLL Version: 0v17p10b530
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 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

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

2.2.1 Simplicity Studio

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

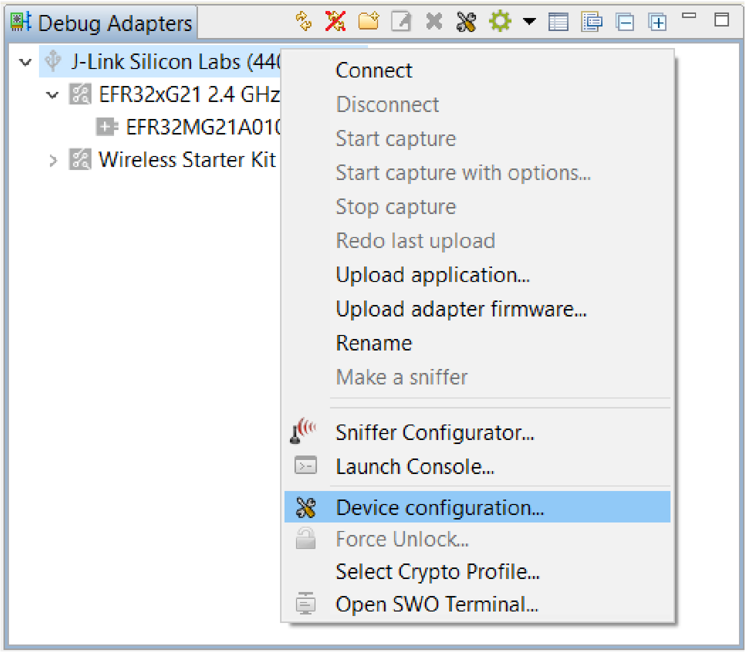


Figure 2.1. Debug Adapter Context Menu

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.

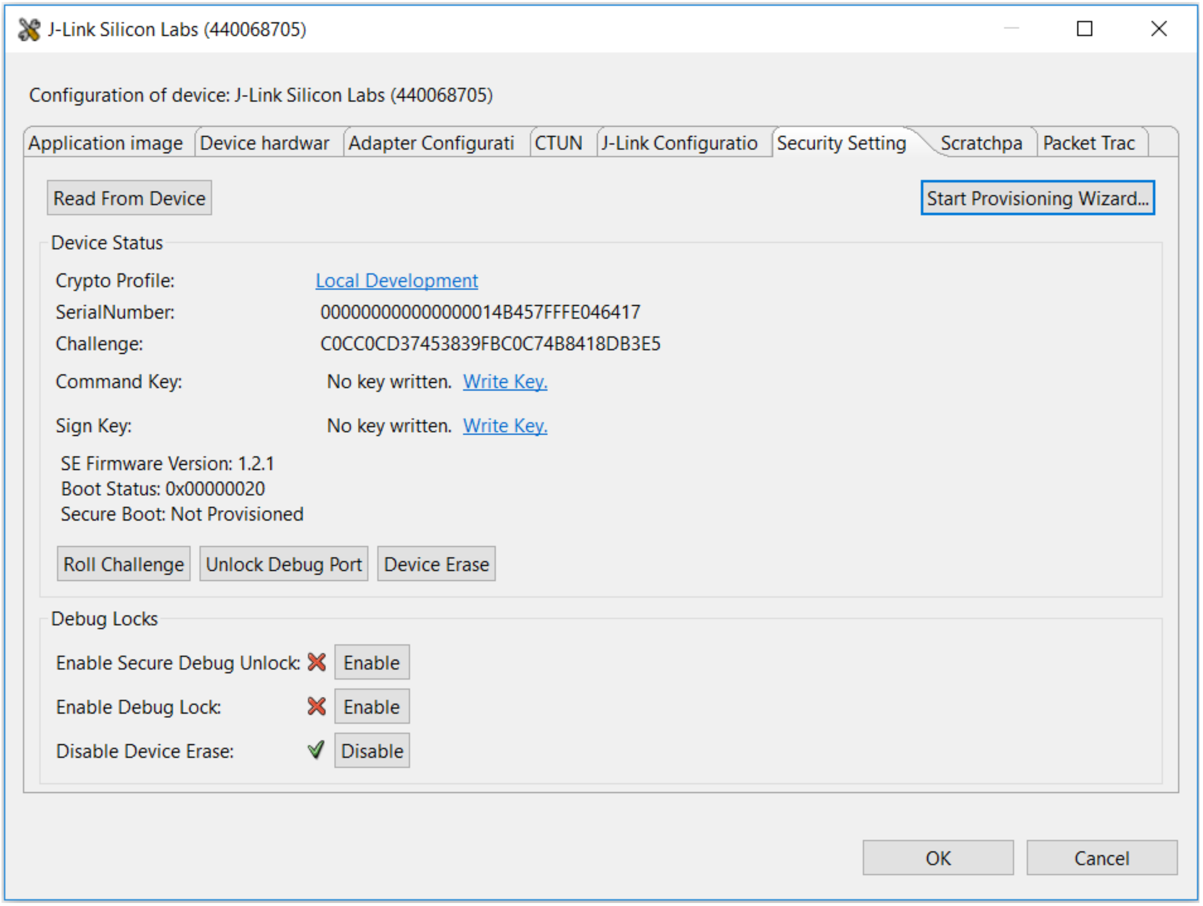


Figure 2.2. Configuration on Selected Device

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.

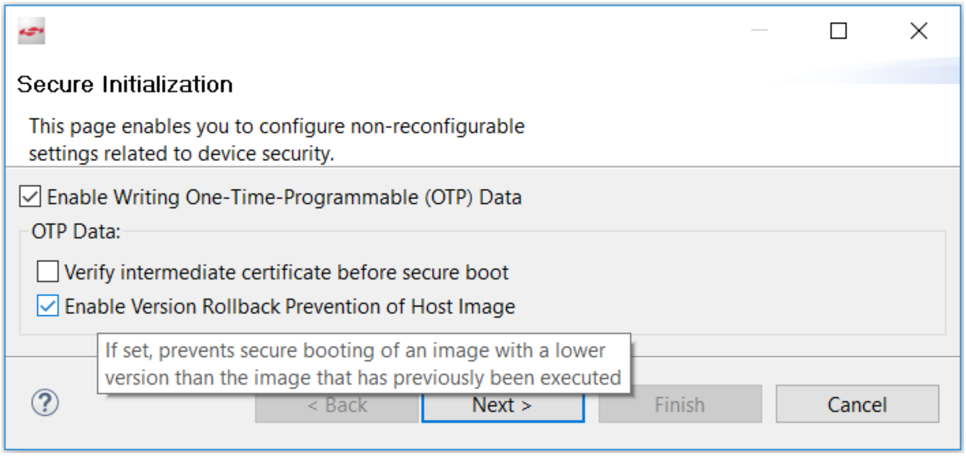


Figure 2.3. Secure Initialization Dialog Box

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.

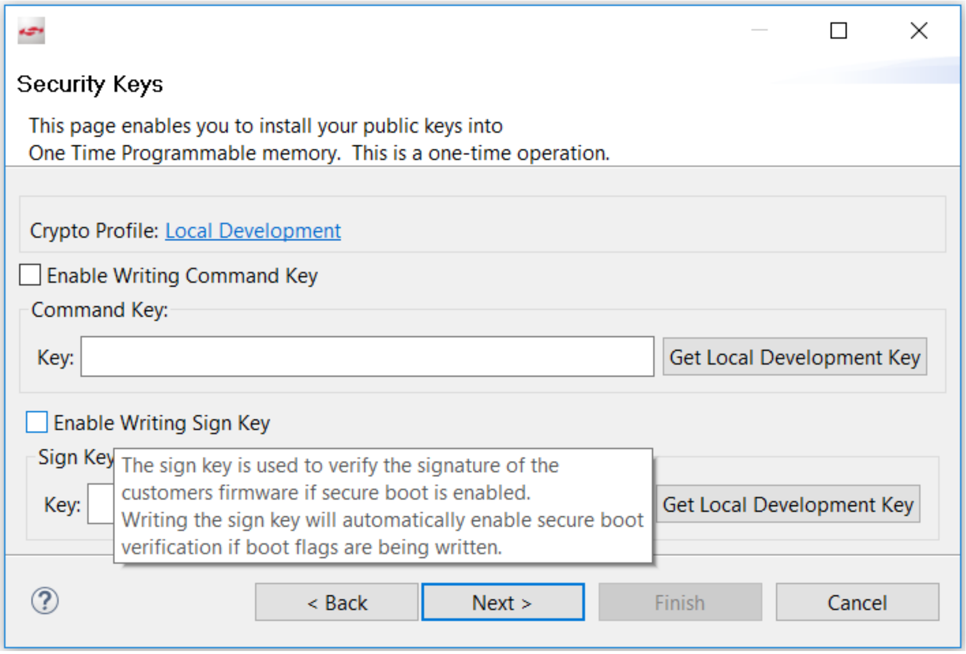


Figure 2.4. Security Keys Dialog Box

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

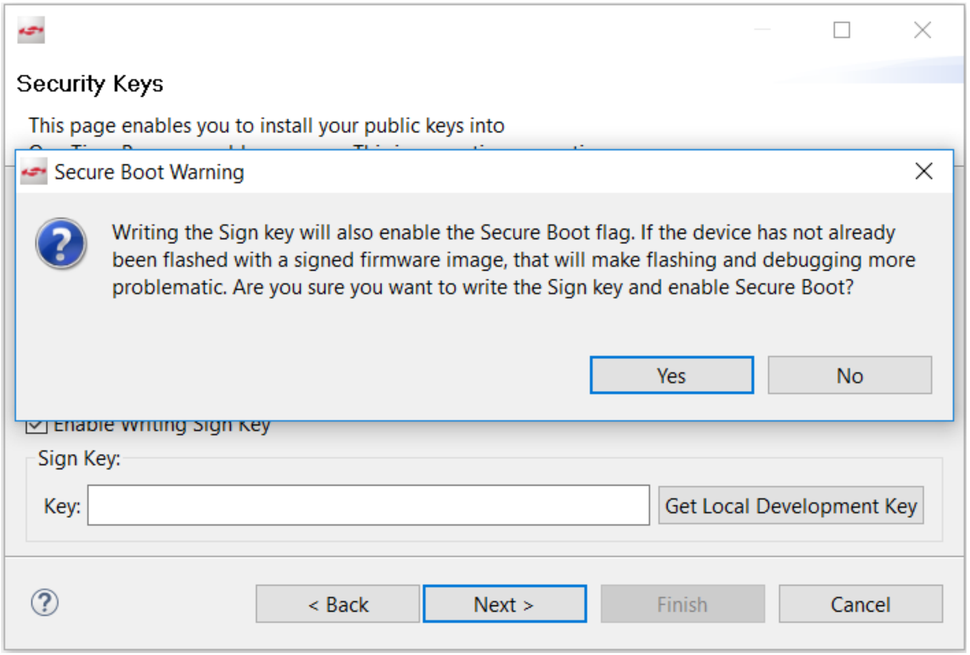


Figure 2.5. Secure Boot Warning

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:**.

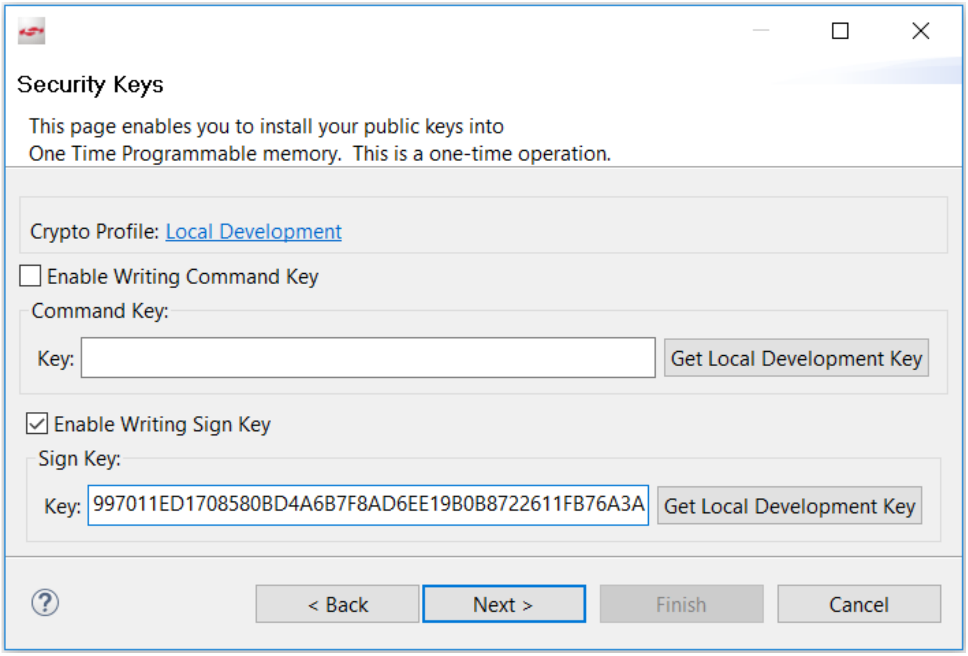


Figure 2.6. Enter Public Sign Key

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

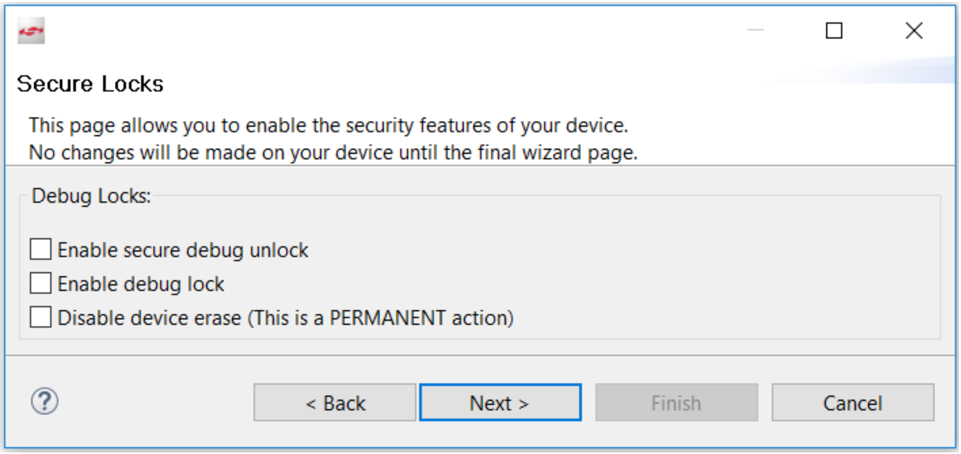


Figure 2.7. Security Locks Dialog Box

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

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

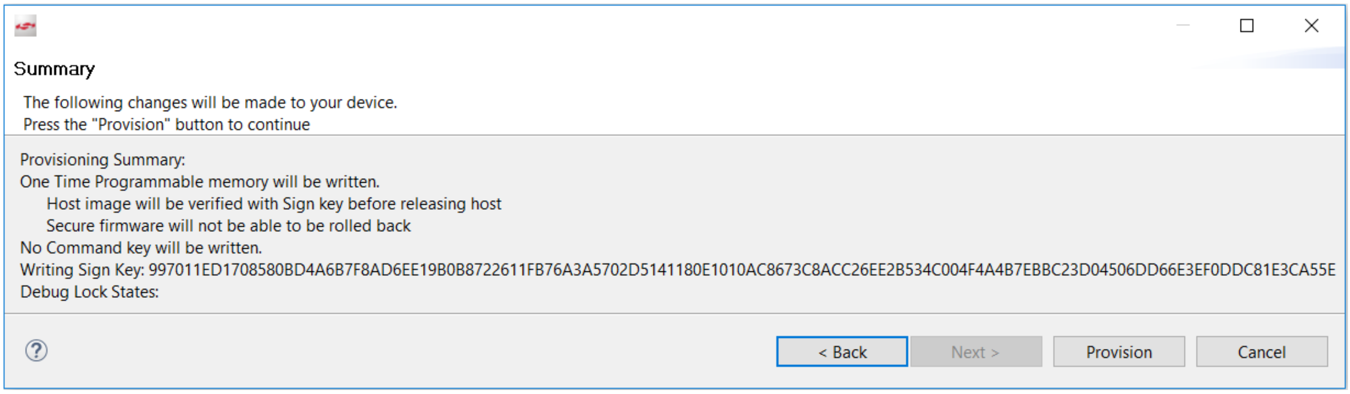


Figure 2.8. Summary Dialog Box

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

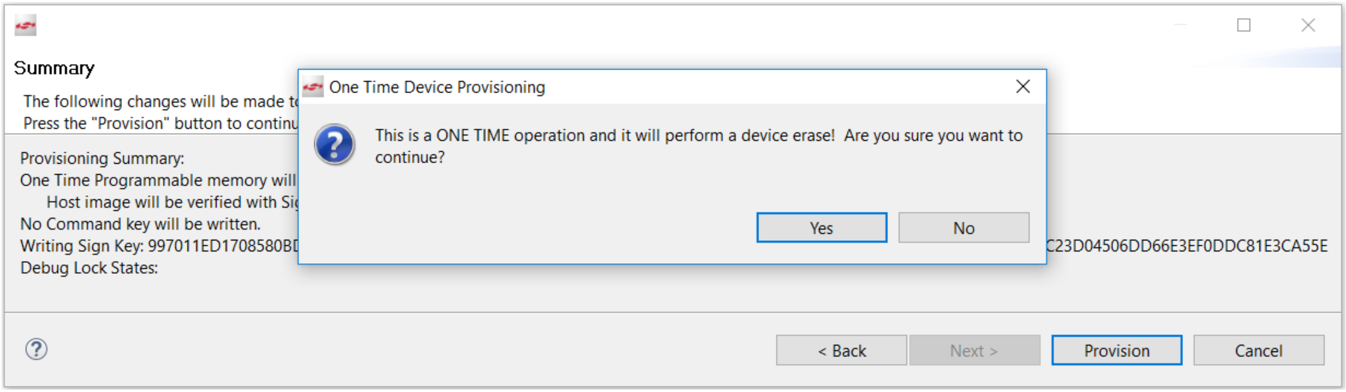
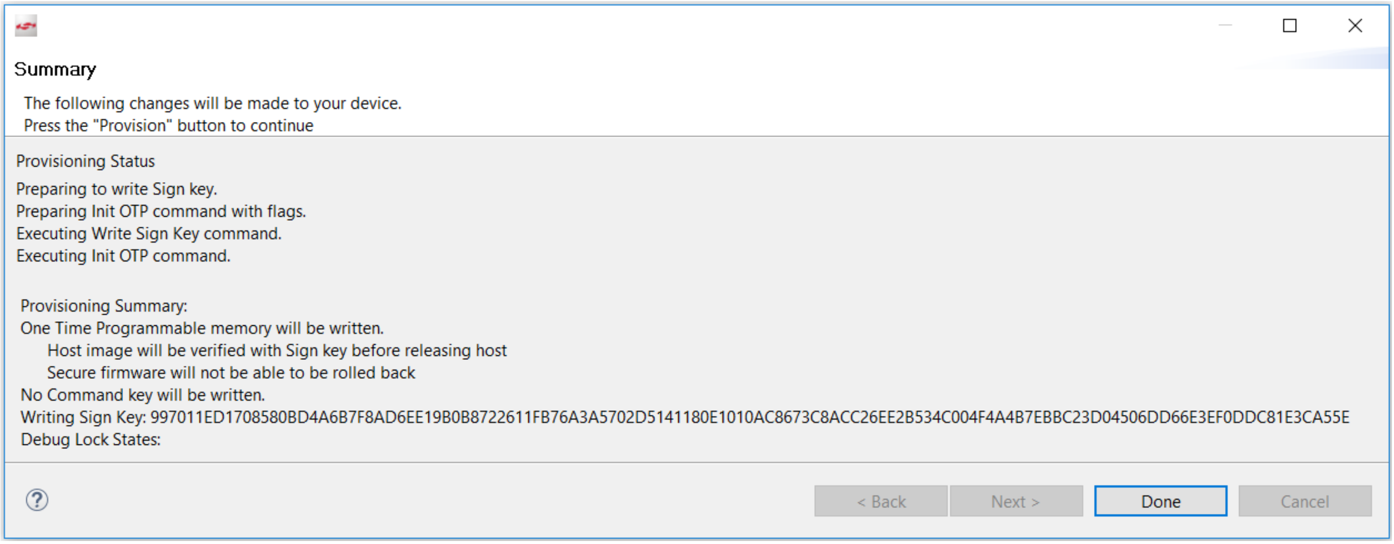


Figure 2.9. One Time Device Provisioning Window

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.

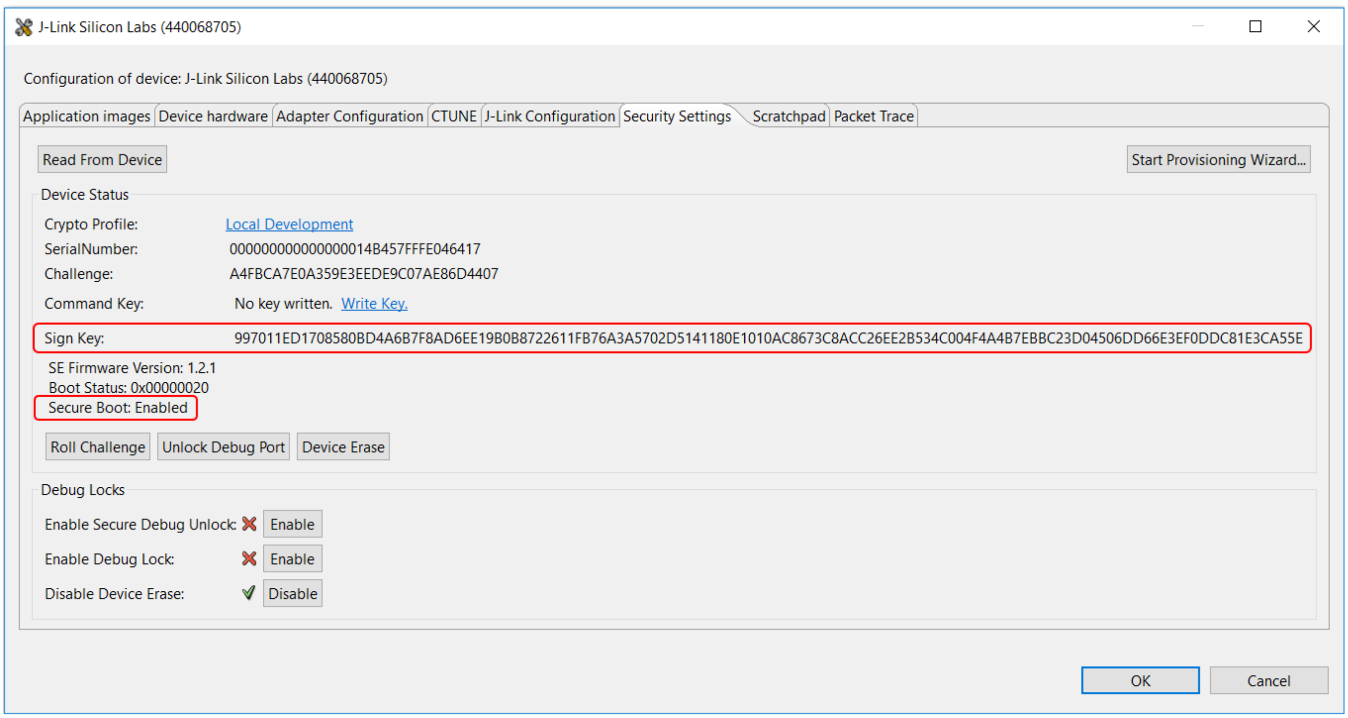


Figure 2.11. Device Configuration after Provisioning

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       : 0000000000000000014b457fffed50d1e
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 0000000000000000014b457fffed50d1e

================================================================================
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 can be found in section "*Secure Boot Enabling*" in *AN1222: Production Programming of Series 2 Devices*.

2.3 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.

2.3.1 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**].

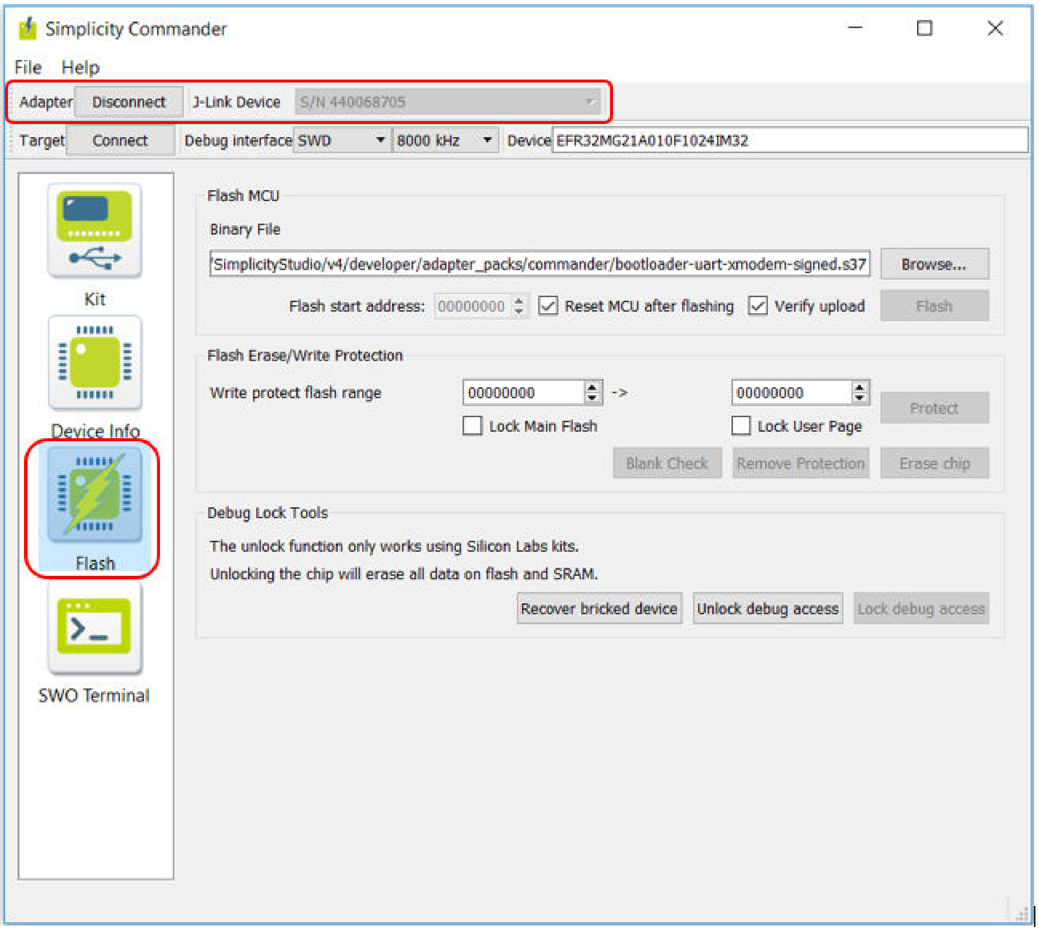


Figure 2.12. Connect Simplicity Commander to a WSTK

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.

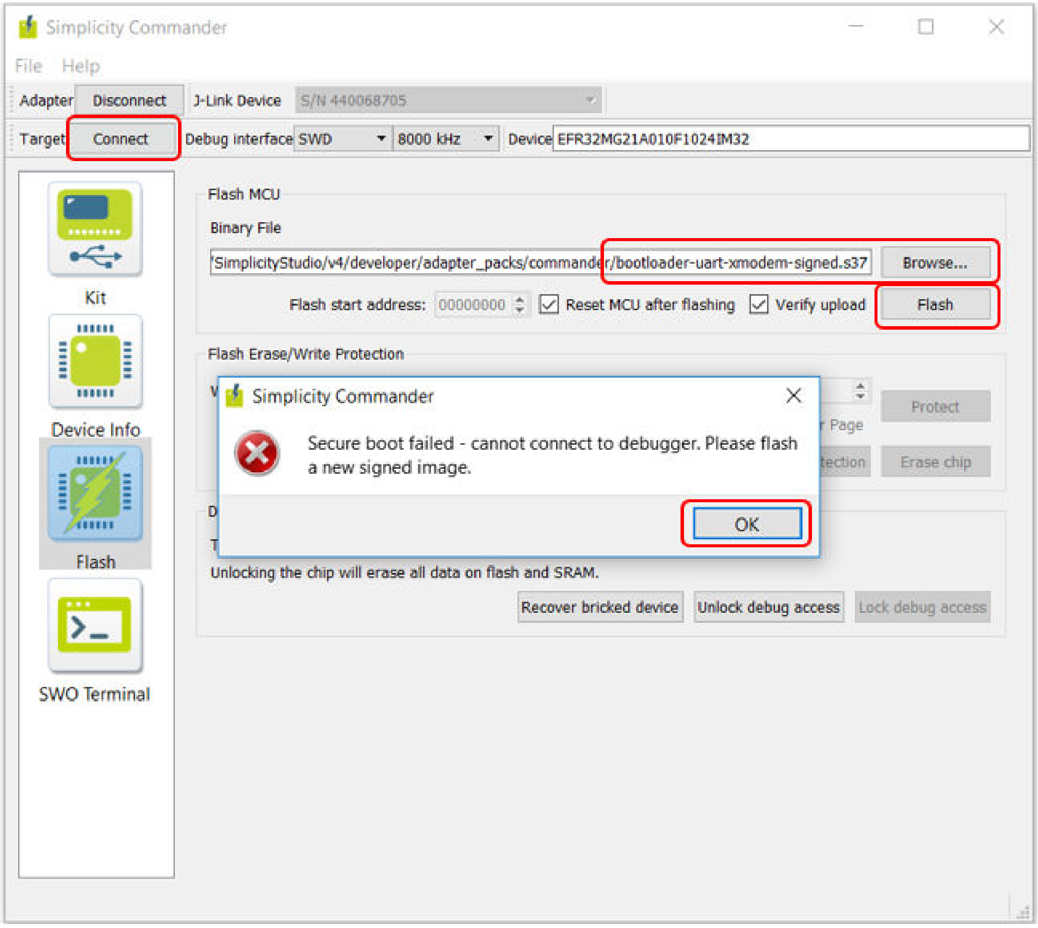


Figure 2.13. Flash Correctly-Signed Image

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

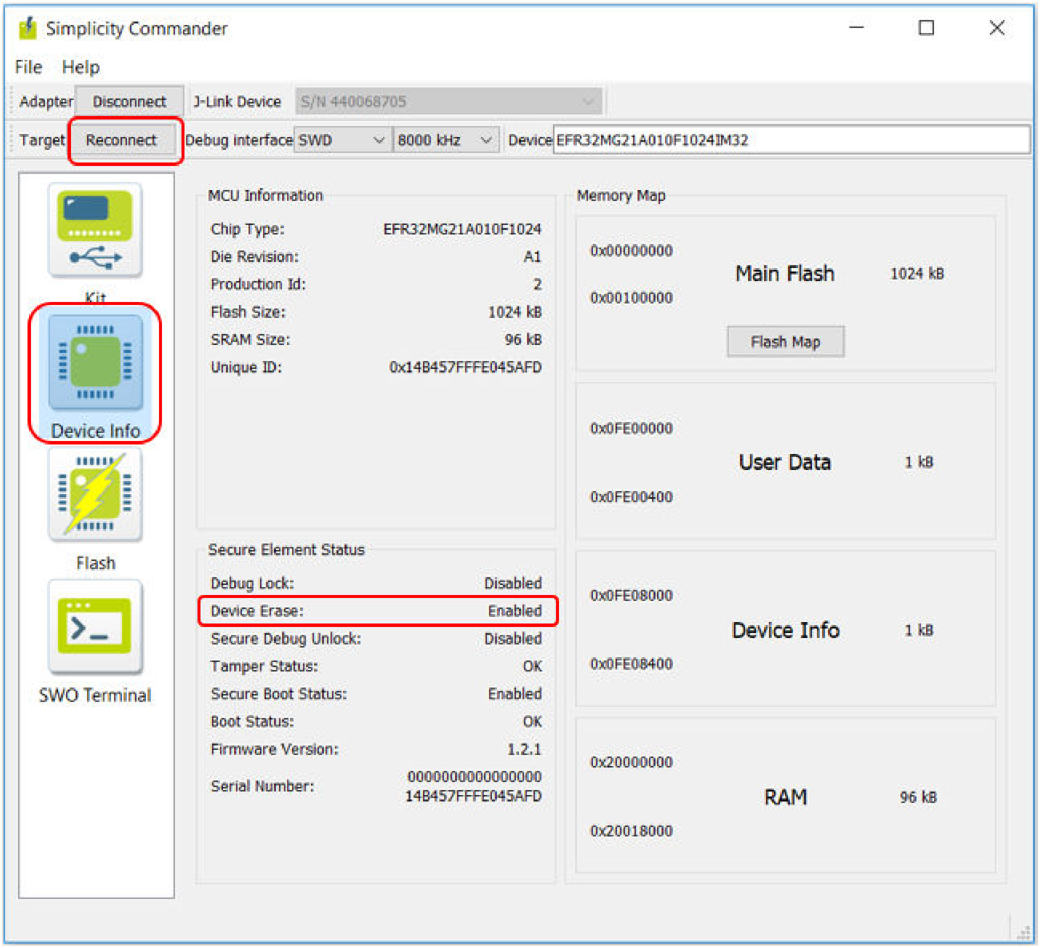


Figure 2.14. Device Information on Recovered Device

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

2.3.2 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.

2.4 Upgrade to Secure Boot with RTSL

This section describes how to upgrade devices already deployed in the field to Secure Boot with RTSL.

1. Upgrade SE firmware to the latest version, see section "*Gecko Bootloader Operation - Secure Element Upgrade*" in *UG266: Silicon Labs Gecko Bootloader User's Guide*.
2. Follow procedures in section "*Enabling Secure Boot RTSL on EFR32xG21*" in *UG266: Silicon Labs Gecko Bootloader User's Guide*.
3. The code example below is used to install the Public Sign Key through mailbox interface. The Public Sign Key here is copied from `sign_pubkey.txt` generated in 2.1.1 Using Simplicity Commander step 6.

```
#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
}
```

4. The code example below is used to enable the Secure Boot through mailbox interface. The `otpConfig` structure contains the desired Secure Boot settings described in section "*Secure Boot Enabling*" in *AN1222: Production Programming of Series 2 Devices*.

```
#include "em_chip.h"
#include "em_se.h"
#include "application_properties.h"

static SE_OTPInit_t otpConfig =
  {
    true,       // Enable secure boot
    false,      // No certificate
    true,       // Enable anti-rollback
    false,      // No lock
    false       // No lock
  };
static SE_Status_t status;

extern const ApplicationProperties_t applicationProperties;     // For secure boot

/***************************************************************************//**
 * @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
}
```

Note:

1. For mailbox interface, see section "*Secure Element Subsystem*" in *AN1190: Series 2 Secure Debug*.
2. The functions in code examples are fully described in the Secure Element Subsystem `emlib` online documentation located at https://docs.silabs.com/mcu/latest/efr32mg21/group-SE.

3. Related Documents

- UG162: Simplicity Commander Reference Guide
- UG266: Silicon Labs Gecko Bootloader User's Guide
- AN1190: Series 2 Secure Debug
- AN1222: Production Programming of Series 2 Devices

4. Revision History

Revision 0.2

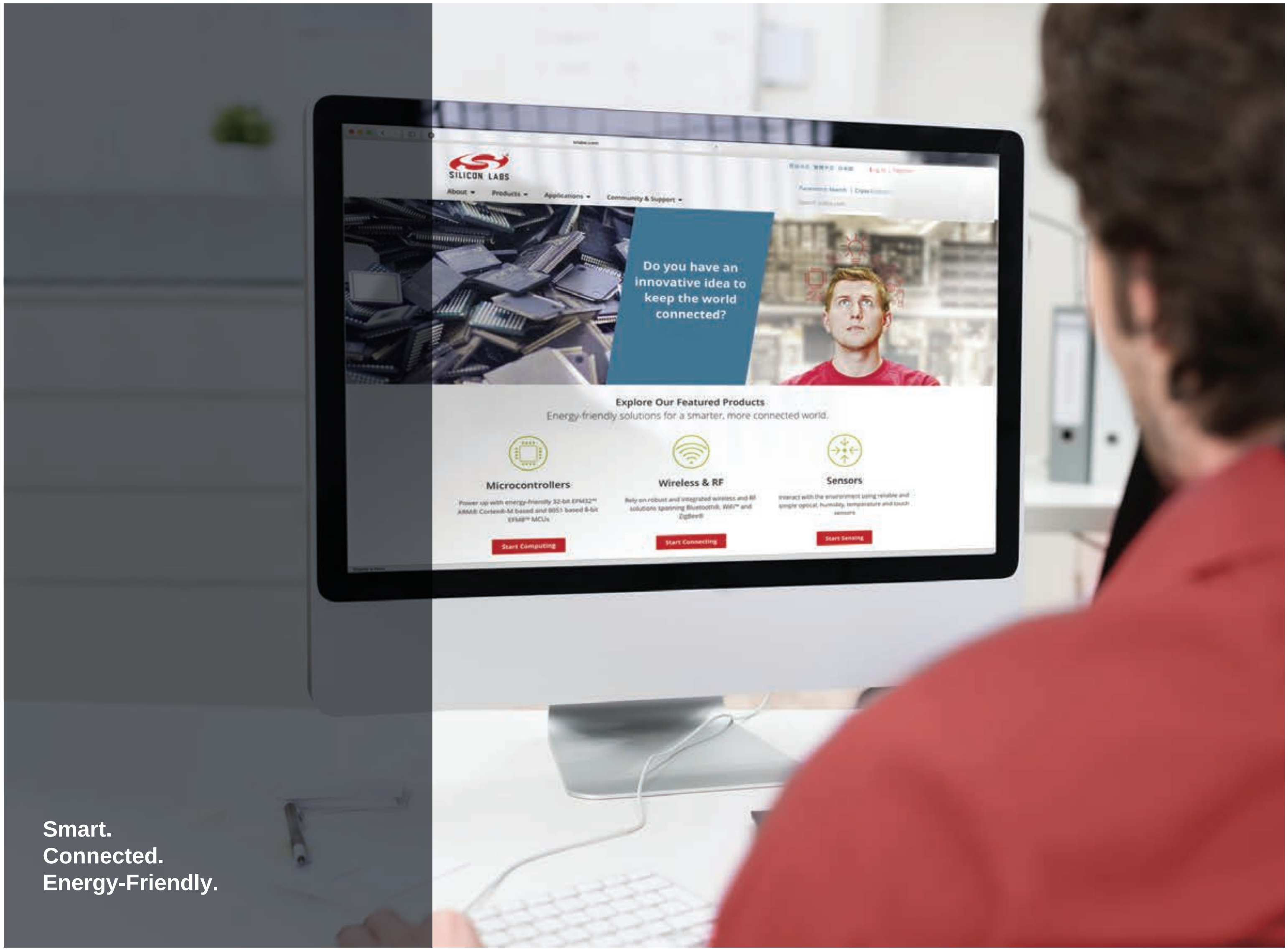
March 2020

- Added figure to Secure Boot (ECDSA) in Series 1 Devices section.
- Added Secure Element (SE) and Virtual Secure Element (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 Laboratories Inc.
400 West Cesar Chavez
Austin, TX 78701
USA

http://www.silabs.com